Dissertation

an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

HELENA

Handling massively distributed systems with ELaborate ENsemble Architectures

vorgelegt von

Annabelle Teresa Klarl

München, den 24. August 2016

Erstgutachter: Prof. Dr. Rolf Hennicker, LMU München
Zweitgutachter: Prof. Rocco De Nicola, PhD, IMT Lucca
Tag der mündlichen Prüfung: 15. November 2016

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.2011, §8, Abs. 2 Pkt. 5.)

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, ohne unerlaubte Beihilfe angefertigt ist.

München, den 24. August 2016

.....

Annabelle Teresa Klarl

Abstract

Ensemble-based systems are software-intensive systems consisting of large numbers of components which can dynamically form goal-oriented communication groups. The goal of an ensemble is usually achieved through interaction of some components, but the contributing components may simultaneously participate in several collaborations. With standard component-based techniques, such systems can only be described by a complex model specifying all ensembles and participants at the same time. Thus, ensemble-based systems lack a development methodology which particularly addresses the dynamic formation and concurrency of ensembles as well as transparency of participants.

This thesis proposes the HELENA development methodology. It slices an ensemble-based system in two dimensions: Each kind of ensemble is considered separately. This allows the developer to focus on the relevant parts of the system only and abstract away those parts which are non-essential to the current ensemble. Furthermore, an ensemble itself is not defined solely in terms of participating components, but in terms of roles which components adopt in that ensemble. A role is the logical entity needed to contribute to the ensemble while a component provides the technical functionalities to actually execute a role. By simultaneously adopting several roles, a component can concurrently participate in several ensembles.

HELENA addresses the particular challenges of ensemble-based systems in the main development phases: The domain of an ensemble-based system is described as an ensemble structure of roles built on top of a component-based platform. Based on the ensemble structure, the goals of ensembles are specified as linear temporal logic formulae. With these goals in mind, the dynamic behavior of the system is designed as a set of role behaviors. To show that the ensemble participants actually achieve the global goals of the ensemble by collaboratively executing the specified behaviors, the HELENA model is verified against its goals with the model-checker Spin. For that, we provide a translation of HELENA models to PROMELA, the input language of Spin, which is proven semantically correct for a kernel part of HELENA. Finally, we provide the Java framework jHELENA which realizes all HELENA concepts in Java. By implementing a HELENA model with this framework, HELENA models can be executed according to the formal HELENA semantics. To support all activities of the HELENA development methodology, we provide the HELENA workbench as a tool for specification and automated verification and code generation. The general applicability of HELENA is backed by a case study of a larger software system, the Science Cloud Platform. HELENA is able to capture, verify and implement the main characteristics of the system. Looking at HELENA from a different angle shows that the HELENA idea of roles is also well-suited to realize adaptive systems changing their behavioral modes based on perceptions. We extend the HELENA development methodology to adaptive systems and illustrate its applicability at an adaptive robotic search-and-rescue example.

Zusammenfassung

Ensemble-basierte Systeme sind software-intensive Systeme mit einer großen Anzahl von Komponenten, die sich dynamisch zu kleineren, ziel-orientierten Kommunikationsgruppen zusammenschließen können. Das Ziel eines Ensembles kann üblicherweise nur erreicht werden, indem mehrere Komponenten interagieren. Allerdings können die beitragenden Komponenten auch an mehreren Kollaborationen gleichzeitig teilnehmen. Mit den üblichen komponenten-basierten Entwicklungstechniken können solche Systeme nur durch ein komplexes Modell beschrieben werden, das alle Ensembles und Teilnehmer gleichzeitig spezifiziert. Deshalb bedarf es einer Entwicklungsmethodik für ensemble-basierte Systeme, die die dynamische Bildung und Nebenläufigkeit von Ensembles sowie die Transparenz der Teilnehmer adressiert.

Die vorliegende Arbeit schlägt die HELENA-Entwicklungsmethodik vor, die ein ensemble-basiertes System entlang zweier Dimensionen aufspaltet: Jede Art von Ensemble wird einzeln betrachtet. Dies erlaubt dem Entwickler, sich nur auf die relevanten Teile des Systems zu konzentrieren und von den Teilen zu abstrahieren, die nicht das aktuelle Ensemble betreffen. Ein Ensemble selbst wird nicht nur über seine teilnehmenden Komponenten definiert, sondern durch Rollen, die Komponenten in einem Ensemble annehmen können. Eine Rolle ist die logische Einheit, die zum Ensemble beiträgt, während eine Komponente nur die technischen Funktionalitäten bereitstellt, um eine Rolle auszuführen. Indem eine Komponente mehrere Rollen gleichzeitig annimmt, kann sie zur gleichen Zeit an mehreren Ensembles teilnehmen.

HELENA adressiert die speziellen Herausforderungen eines ensemble-basierten Systems in den Hauptentwicklungsphasen: Die Domäne eines ensemble-basierten Systems wird durch eine Ensemblestruktur von Rollen beschrieben, die auf einer komponenten-basierten Plattform aufbaut. Basierend auf der Ensemblestruktur werden die Ziele des Ensembles als Formeln in linearer temporaler Logik erfasst. Mit diesen Zielen vor Augen wird das dynamische Verhalten des Systems als eine Menge von Rollenverhalten modelliert. Um zu zeigen, dass die Ensemble-Teilnehmer die globalen Ziele des Ensembles tatsächlich durch kollaborative Ausführung der spezifizierten Verhalten erreichen, wird das HELENA-Modell mit dem Model-Checker Spin verifiziert. Dafür stellen wir eine Übersetzung von HELENA Modellen nach PROMELA, der Eingabesprache für Spin, bereit, die für eine Kernsprache von HELENA als semantisch korrekt bewiesen wird. Schließlich stellen wir die Java-Bibliothek jHELENA zur Verfügung, die alle HELENA-Konzepte in Java realisiert. Indem ein HELENA-Modell mit dieser Bibliothek implementiert wird, kann es anschließend gemäß der formalen HELENA-Semantik ausgeführt werden. Um alle Aktivitäten der HELENA-Entwicklungsmethodik zu unterstützen, bieten wir die HELENA Workbench an, die als Werkzeug zur Spezifikation und automatisierten Verifikation und Codegenerierung dient. Die allgemeine Anwendbarkeit von HELENA wird durch eine Fallstudie eines größeren Softwaresystems, der Science Cloud Plattform, gezeigt. HELENA erlaubt es die Hauptmerkmale des Systems zu erfassen, zu verifizieren und zu implementieren. Außerdem betrachten wir HELENA noch von einem anderen Blickwinkel aus: Wir zeigen, dass die HELENA-Idee von Rollen sich gut dafür eignet, adaptive Systeme zu realisieren, die ihre Verhaltensart aufgrund von Wahrnehmungen ihrer Umgebung ändern. Wir erweitern die HELENA-Entwicklungsmethodik für adaptive Systeme und veranschaulichen ihre Anwendbarkeit an einem adaptiven roboterzentrierten Rettungsszenario.

to my mother

—

my idol for life

Acknowledgments

My cordial thanks go to Rolf Hennicker, my supervisor and first reviewer. In 2011, I got to know him as an excellent lecturer at our group at LMU and worked with him as his teaching assistant. It was one of the best coincidences that one day in 2013, we happened to engage into a lively discussion about ensembles, although we never scientifically worked together until then. I deeply appreciate his excellent supervision: Our discussions were always fruitful and at eye level. We shared the same love for the detail. His interested and insistent questions as well as his constructive feedback and honest appreciation of my work pushed me to cover each topic in depth. He always showed me the light at the end of the tunnel, may it be scientifically or personally. I thank him for his close supervision and all the moral support he provided to me.

I also cordially thank Rocco De Nicola, my second reviewer. He crossed my scientific way at many pleasant occasions. In the EU project ASCENS, I experienced him as an excellent discussion partner about ensembles and their formal models. As a guest lecturer, he enriched our group with his bright and friendly personality as well as with his open scientific interest. Personally, I pleasantly remember our joint participation at the Dagstuhl seminar "Collective Adaptive Systems: Qualitative and Quantitative Modelling and Analysis". I appreciate him as a very dedicated and warm person who always has a joyful twinkle in the eye.

Without Martin Wirsing, the head of the PST group during my PhD thesis, I would never have had the opportunity to meet all the people who scientifically supported my work. I thank Martin Wirsing for the faith he had in me and my abilities when hiring me back in 2010. He introduced me to the world of science and especially the ASCENS project. He encouraged me to build up my scientific knowledge by guiding collaborative work with my colleagues Matthias Hölzl, Christian Kroiß and finally Rolf Hennicker. It is due to his enviable talent of bringing people together that I had such wonderful co-workers. Among them, I especially thank Andreas Schroeder, Philip Mayer, Marianne Busch, Lenz Belzner and Joschka Rinke for their moral support during my PhD thesis; they persuaded me that I can do it. Furthermore, I thank Anton Fasching with all my heart for every chat we had during the mornings when everybody else was still asleep at home while we already started working. His open-heartedness and honest personal appreciation helped me through many hard times sitting in my office.

I also wish to thank all my friends who supported me in lots of tearful and terrified, but also joyful and enthusiastic moments of my thesis. A special word of thanks is due to Benedikt Hauptmann. As my "Diss Buddy", he supported me with the necessary pressure to work on my thesis, motivated me in times of procrastination and helped me out of my panic when opposed with new, seemingly unbearable challenges. He also came up with the wonderful idea to occasionally work at the offices of the group of Manfred Broy at TUM. I am very grateful for the warm welcome from everybody of the group and the buzzing and motivating work atmosphere which I enjoyed there. Furthermore, I thank Chrisi Gerlach who invested his spare time into proofreading parts of my thesis.

My last thanks are directed to my family, my mother, my father and my sister. They always believed in me, never lost the faith in my abilities and encouraged me to do the same. They sensed when I was totally out of my mind and stressed and calmed me down with their love and extensive hiking and cooking sessions.

Thank you all!

Contents

Chapter 1

Introduction

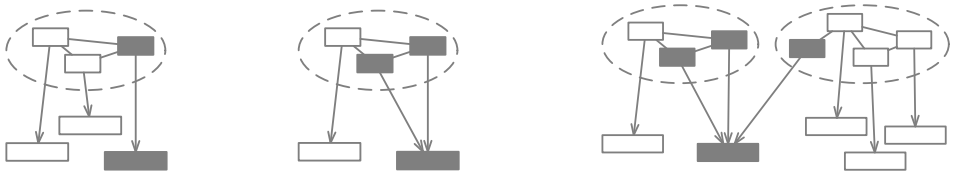
Ensemble-based systems are software-intensive systems consisting of a large number of components which can dynamically form goal-oriented communication groups and accomplish a common goal in collaboration. In general, the intended goal of an ensemble can only be achieved by interaction between its constituent members and requires certain local functionalities to be offered by the participants of the ensemble. As soon as appropriate components join the collaboration and contribute the desired functionalities, the ensemble becomes effective. Conversely, this means that an ensemble affects only a part of the global system, which may be a large distributed system, since only particular participants have to chip in. When performing a goal-oriented task in an ensemble-based system, only the components participating in the ensemble are of interest and need to be coordinated. But at the same time, components may be employed in several collaborations simultaneously allowing the ensemble-based system to concurrently work on several goals.

Ensemble-based systems especially target two new trends in computing: ubiquitous computing and autonomic computing. *Ubiquitous computing* [Wei99] introduces more and more computing devices into our systems. The contributing components are no longer restricted to powerful work-stations. They are augmented with small and cheap mobile devices like laptops, handhelds, and mobiles. Wide heterogeneity is an inherent property of such systems and the number of globally interconnected components grows daily. Mobile devices also introduce mobility into ensemble-based systems. That means that on the one hand moving components like mobiles are faced with continuously changing and possibly unknown environments. On the other hand, the overall system is subject to changes of participating components and cannot rely on a fixed set of contributors. Performing distributed goal-oriented tasks in such a volatile and pervasive system calls for dynamically formed collaboration groups as we envision them by ensembles.

Autonomic computing [KC03] introduces a certain degree of autonomy to the individual components of the ensemble-based system. The paradigm advocates that we no longer create systems which are thoroughly administrated by hand. Once employed the system should rather manage itself and keep itself alive and running. The contributing components have the autonomy to coordinate themselves and their interactions according to high-level objectives without any external supervision. This is especially helpful in ubiquitous systems since the responsibility for coordination is distributed among the participants of the system.

Ensembles appear in the human world in a similar way as we described them for software systems. In the domain of soccer, the underlying platform of distributed components consists of the players, trainers, medical staff, managers etc. On top, different ensembles are formed like national teams or soccer club teams. Each member of such a team can dynamically join and leave the team, takes over responsibility for different tasks and can changes his tasks. For example, in the German national team, the player Manuel Neuer is goal keeper, the trainer Joachim Löw is head coach, and the medical staff member Dr. Hans-Wilhelm Müller-Wohlfahrt is medical doctor. We say that each component like the player Manuel Neuer participates in a certain ensemble by *adopting a particular role* like goal keeper. Fig. 1.1a illustrates this situation. The boxes in the lower part represent the components contributing to the ensemble (note that there might be more components in the overall system which do not contribute to the ensemble under consideration and are thus not shown here). The dashed ellipse encompasses all roles which are needed for the functionality of the ensemble. Most importantly, we represent the adoption relation between role and component by an arrow from role to component. Connections between the roles denote which roles interact.

However, it is not unusual that one component (sequentially or concurrently) adopts different roles in the same ensemble as shown in Fig. 1.1b. For example, in the German national team, the player Bastian Schweinsteiger was center midfield, but also captain. One component can even (concurrently) adopt several roles in different ensembles as shown in Fig. 1.1c. For example, a game of the German national team could be organized as a charity game. Then, two ensembles exist in parallel, the German national team as well as the charity ensemble. The latter consists of organizers, donors and charity players such that a player like Bastian Schweinsteiger not only adopts the roles of center midfield and captain in the German national team, but at the same time also the role of a charity player in the charity ensemble.



(a) One component adopts one role

(b) One component adopts two roles

(c) A component adopts several roles in the same and different ensembles.

Figure 1.1: Participation of components in ensembles

1.1 Challenges of Ensemble-Based Systems

Ensembles are characterized by the goal-oriented collaboration of participants, building on top of a common component-based platform. These characteristics bear challenges for the development of ensemble-based systems which are partially known from distributed systems [CDKB11], but ensembles add a new layer of difficulty to them. The following list presents important challenges and illustrates some of them on the basis of the domain of soccer:

Concurrency: Concurrent execution has to be managed on two levels. Within an ensemble, the goal-oriented work has to be appropriately distributed among the participants of the ensemble and collaboration must be coordinated. In the do-

main of soccer, defense and offense have to be assigned to the different positions in the team like defenders and strikers. On system level, several ensembles must be able to concurrently perform their goal-oriented tasks possibly sharing the same participating components and taking care of conflicting participations. In the domain of soccer, national tournaments have to be coordinated with international tournaments to allow different players to participate in both.

Heterogeneity: In the ubiquitous computing paradigm, heterogeneity is increased by including small and cheap components like mobiles into the system. In such a setting, ensembles have to be formed unaffected or even have to exploit heterogeneity of available components in the system. They must be able to assign work according to the individual properties of each contributing component. In the domain of soccer, players show different strengths for abilities like dribbling, heading, corner kicks, free kicks etc. Depending on their particular abilities, they have to be assigned to the different positions of a soccer team.

Extensibility: On the system level, mobility of mobile devices requires that the overall system is able to cope with new component instances and it can make them available to the currently running ensembles. In the domain of soccer, scouts should continuously discover new talents which organizations like UEFA or Bundesliga trade between the different clubs. On the ensemble level, an ensemble has to be able to specify when new participants are allowed to join in and how they are integrated in the collaboration. This dynamism has to be reflected in the system's architecture to allow for connections between component instances to be set up dynamically. In the domain of soccer, tournaments rules have to define when a player can join the team on the field in exchange for another player leaving the field.

Dynamism: Ensembles are dynamically formed collaboration groups. Not only can new members join the collaboration during the life-time of an ensemble, but also the participation of a component can change. A component may communicate with different partners during operation of the ensemble such that communication links are dynamically reconfigured. Furthermore, a component may take over different tasks for the accomplishment of the global goal of the ensemble such that the contribution of a single component to the overall ensemble dynamically changes. In the domain of soccer, the task and preferred communication of a midfield player may change depending on whether his team is currently attacking or defending. When attacking, the midfield player may interact more frequently with the strikers; conversely, and when defending, the same player may be integrated into the line of defense. It may even happen that a player adopts an additional role in the team during a tournament, e.g., if Bastian Schweinsteiger leaves the field, the role of the captain is assigned to Manuel Neuer.

Transparency: It should be transparent from the user which components form an ensemble. It is only relevant that the ensemble achieves its goal while the concrete participants do not matter as long as they provide the required capabilities. This special kind of transparency allows dynamic composition of ensembles since they do not rely on particular components, but on their provided capabilities. In the domain of soccer, different moves can be planned and trained only based on the positions of those involved. Different players can transparently fill in on these positions without changing the overall plan of the move.

Goal-Orientation: Despite heterogeneity, and transparent and dynamic composition, an ensemble must always work towards the intended goal. It should be guaranteed that the goal is actually achieved. In the domain of soccer, a team should always be able to compete in a tournament despite different abilities of the current players in the squad (e.g., because of players changing affiliations, current assembly of the team, injuries etc.).

Autonomy: Due to the large number of components in the system and their mobility, global supervision of the whole ensemble-based system must be avoided. The organization of ensembles has to be coordinated locally such that each ensemble determines its composition and behavior itself. The individual components have to be granted with a certain kind of autonomy to self-organize their contribution to goal-oriented collaborations. In the domain of soccer, each club should organize itself independently from other clubs. However, internally, each club is mostly centrally coordinated. In an ideal ensemble-based system, those central coordinators should be avoided for the benefit of self-managed collaborations.

1.2 Shortcomings of Existing Approaches

A central approach to describe the structure and behavior of interacting systems is component-based software engineering (CBSE) [Szy02, RRMP08]. It aims at developing self-contained, reusable components. For each component, its structure, interface to communication partners and behavior is precisely specified to allow integration of several components to larger interacting systems. A wide variety of architecture description languages (ADLs) like Wright [AG94, AG97], Darwin [MK96], ACME [GMW97], and PADL [BCD00, BCD02] allow description of the high-level structure of a system in terms of components and their interactions and to reason about system properties at this high level of abstraction. Frameworks like Fractal [BCL+04, BABC+09], SOFA [BHP06], ArchJava [ACN02], and Java/A [BHH+06] do not only provide formal component-based architecture models, but also implement them in Java. ADLs and their frameworks are mainly used for the specification and implementation of component-based architectures via appropriate tool support. Their specification power is often limited and verification is only supported for a fixed set of properties [BvVZ06]. In contrast, a set of formal approaches like team automata [tEKR03], interaction automata [BvVZ06], assemblies [HK11] or multiparty session types [CDPY15] focuses on the precise specification of component interactions with less attention to the system's architecture. These approaches aim at automated verification, but are often too abstract to serve as implementation models [BvVZ06].

All of these approaches share a similar understanding of components and their composition to larger systems. An atomic component is thought of as a self-contained computational unit of the system. Its interface to the outside world is described by ports which handle communication with other components[1]. Such atomic components can be composed to hierarchical components and finally whole systems. The main architectural element are connectors which handle interaction and coordination between components. A connector specifies properties (often by a communication protocol,

[1]For communication, we only consider bi-directional message exchange in this thesis as opposed to broadcast messaging, remote procedure call or knowledge exchange via shared knowledge bases. We assume that the behavior of a component is specified by a single sequence of actions which conversely means that an atomic component is not able to simultaneously execute different behaviors.

e.g., [AG94, AG97]) which the ports of components interacting through this connector have to fulfill. That means that sets of interacting components, like we consider them in ensemble-based systems, are already a main characteristic of CBSE and its different proposed component models. However, standard component models are not able to express some of the distinguishing features of ensemble-based systems:

Explicit Notion of Ensembles: The notion of an ensemble as a group of collaborating components is central to ensemble-based systems. In standard component models like Wright [AG94, AG97] or ACME [GMW97], such ensembles are only implicitly defined by sets of connected components. Hierarchical component models like Fractal [BCL+04, BABC+09] or SOFA [BHP06] allow to simulate the explicit notion of an ensemble by encapsulating the interacting components in a single composite component. As soon as we envision several ensembles which overlap in terms of participants, hierarchical composition of components cannot express the notion of ensembles anymore. Fractal [BCL+04, BABC+09] thus introduces the concept of shared components. They are able to contribute to different collaboration groups, but while the concept of sharing is made explicit, groups which share a component are again only implicitly determined by the sets of connected components. We still miss an explicit notion of ensemble. Such a notion would allow focusing design and analysis on the relevant participants of an ensemble only.

Explicit Notion of Active Roles: If a component is shared between several ensembles, it has to provide all communication capabilities and behaviors which it needs to contribute to the different ensembles. To describe a component with different tasks distributed over several ensembles, standard component-based techniques propose to design a one-fits-all component which combines all required capabilities and behaviors for such tasks. Some capabilities might be needed in several ensembles, but others are specific for only one ensemble. Nevertheless, they are all pooled together into one large component. Which interface is needed in which ensemble is only implicitly given, e.g., by the involved communication ports. Furthermore, such a one-fits-all component must be able to work on every task which an ensemble requires from it. Therefore, it has to offer and execute a large parallel behavior. The design of such a behavior is complex and error-prone due to the large number of possible interleavings.

This design suffers from two disadvantages: Firstly, the shared one-fits-all component cannot be reused to perform other tasks. If we want the component to perform a different task, the component has to be broken up and appropriate capabilities and behaviors have to be included. Secondly, we lack an explicit concept to define which functionalities need to be contributed by the participants of an ensemble and which components of the underlying component-based platform are able to provide those functionalities. Ensembles do not require a particular component to contribute to the collaboration. They only require certain capabilities to be offered and tasks to be fulfilled. In general, these requirements could be offered by different components, but only one will later on contribute to the ensemble. Thus, the functionalities needed by the ensembles should be described separately from the capabilities offered by components.

Dynamic Architecture: In ensemble-based systems, components should dynamically collaborate towards a global goal. Therefore, they need to establish communication

links between each other on demand to allow dynamic interaction. Additionally, they should be able to dynamically request and integrate new participants into the ensemble if the current ensemble composition is not appropriate to achieve the global goal anymore. Reconfiguration of the communication structure between components is already supported by dynamic architecture approaches. For example, SOFA [BHP06] introduces specific reconfiguration patterns to allow controlled modification of the system's architecture. Wright [ADG98] proposes a special configurator component which takes care of establishing connections between components on demand. Other dynamic architecture approaches like Darwin [MK96] allow the dynamic instantiation of components at runtime such that collaborations can be extended by new participants on demand. Those techniques can be transferred to ensemble-based systems, but a developer can soon get lost in the reconfiguration options if the system grows large and consists of many different ensembles. Indeed, software development could benefit from the introduction of the notion of ensembles to only focus on one collaboration group when reconfiguring and from the introduction of the notion of roles to describe communication partners separately from the components currently serving as communication partners.

Some dedicated techniques and tools like the SCEL/PSCEL [DLPT14, MPT13], DEECo [BGH+13], DCCL [BBvP13] and BIP/Dy-BIP [BBB+12, BJMS12] already address the specific characteristics of ensemble-based systems. Apart from BIP/Dy-BIP, they all use membership predicates to express ensembles. A membership predicate determines through properties of components which components are currently considered part of an ensemble. Since the predicate is dynamically evaluated, an ensemble is a volatile snapshot of a system. This adds flexibility, but the conceptual structure of an ensemble describing potential participants and their interaction relationships cannot be explicitly specified. Besides that, all ensemble-based approaches do not propose the concept of roles. Components are thought of as one-fits-all entities which combine all functionalities needed in the different ensembles into one large extensive component. The ensemble-based approaches do neither provide a concept which helps to structure a component according to the different roles it can adopt nor a concept which allows to explicitly describe the participants of an ensemble separately from the underlying components. One exception is PSCEL [MPT13] which integrates FACPL policies [MMPT14] into SCEL specifications. These policies define when certain actions of a component are allowed or how behaviors of components have to be adapted to cope with the current situation. This resembles the idea of roles which are dynamically adopted according to the current task which a component has to fulfill. However, all ensemble-based approaches heavily focus on dynamic communication structures although they rely on different communication paradigms: In SCEL/PSCEL, components explicitly exchange knowledge by putting data items to and retrieving them from knowledge repositories. These knowledge repositories can be dynamically selected by variables or predicates analogously to membership predicates. DEECo and DCCL support implicit knowledge exchange by periodically updating the knowledge of all members of an ensemble. Since the members of an ensemble are dynamically determined from the evaluation of the membership predicate, knowledge is exchanged between varying partners. In BIP/Dy-BIP, components send and receive messages via connectors which can dynamically be bound. Interaction constraints guide the possible connection of communication partners. Finally, SCEL/PSCEL even supports the creation of new components which can dynamically be integrated into ensembles. In conclusion, all ensemble-based approaches

make a big step towards the dynamics of ensemble-based systems. However, they lack structural concepts like ensembles and roles as first-class entities which reduce the complexity of ensemble models on the one hand and support the dynamics of ensembles on the other hand.

1.3 Problem Statement

The development of ensemble-based systems should be supported by an appropriate development methodology aimed at the particular challenges of ensemble-based systems. Special focus has to be directed towards the most prominent development phases, goal specification, system modeling, verification, and implementation: We must provide abstractions to describe ensembles, their participants and relationships as first-class entities. Ensemble goals should be specified on the level of participants of the ensemble. Verification should allow to check goal satisfaction for a single ensemble to reduce complexity of the state space to be searched, but should nevertheless consider the ensemble in the context of the underlying component-based platform. The abstractions used to describe ensemble-based systems should be transferred to the implementation to allow a transparent participation of components in goal-directed ensembles and to allow management of each ensemble independently from all other ensembles.

Ensemble Description: The structure of distributed systems is in general defined by the contributing components of the system and their interaction patterns. In such structural models, the collaborating components are assigned with responsibilities and interaction capabilities to represent the purpose of the system. However, the system structure of an ensemble-based system is inherently complex since several ensembles and transitively even more participants have to be captured in one structural model. With standard component-based techniques, we would create a large and extensive model which specifies all ensembles at the same time.

To reflect the requirements and the dynamics of ensemble-based systems, the standard component-based models must be extended. We want to be able to describe the structure of an ensemble separately from, but on top of a standard component-based platform. This allows to model each ensemble and its participants as self-contained entities where other ensembles do not have to be considered. Contributing components should be able to participate in several ensembles simultaneously without predefining the composed behavior. Ensembles themselves should support dynamic composition in terms of contributing components such that the collaboration is formed on demand. At the same time, the ensemble – or rather the participants – should be able to decide themselves which components should contribute to the ensemble to allow autonomic self-management of the system.

Goal Specification and Verification: Ensembles are dynamically formed goal-directed communication groups. Thus, the question arises how goals, which directly aim at these dynamically formed communication groups, can be specified and verified. Goal specification should focus on the ensemble under consideration within the context of the underlying component-based platform.

When specifying goals in a standard-component-based design with one-fits-all components for an ensemble-based system as described in the first paragraph, the goals would have to be specified on the level of these all-mighty components. It would not be possible to focus only on the relevant parts for the ensemble under consideration.

Similarly, when verifying the standard component-based design against its goals, the model would have to be abstracted to the relevant parts of the ensemble only. The full-blown model would be too large and would contain many parts which would not affect the ensemble at all. A more modularized ensemble model would help to facilitate specification and verification of goals for ensemble-based systems. Therefore, we look for an ensemble model which describes the structure of the ensemble-based system more modularized and which is grounded on a solid formal foundation allowing rigorous analysis. Appropriate goal specifications capture the purpose of an ensemble and can be verified in the formal ensemble model.

Implementation: Another question is how to realize an ensemble-based system in implementation. In principle, complexity issues similar to finding an appropriate abstraction for the system description arise, i.e., how to break down the overall ensemble-based system into groups of communicating entities which can be shared between ensembles. A modularized structural model, as envisioned before, already provides abstractions to describe the underlying component-based platform of an ensemble-based system and all its employed ensembles independently. By transferring these ideas to the implementation, we would be able to sustain complexity reduction on this level as well. The separate implementation of the participants of one ensemble from all other ensembles furthermore allows to add, change and remove ensembles detached from each other. However, special care has to be taken to avoid interferences between members of different ensembles if they access and change the data of the same underlying components.

1.4 Solution Idea

To tackle the development of ensemble-based systems, we introduce the HELENA development methodology to specify ensemble-based systems and their goals as well as to verify and to implement them. The acronym HELENA stands for *"Handling massively distributed systems with ELaborate ENsemble Architectures"*. HELENA's main idea is that an ensemble is not specified in terms of participating components, but in terms of *roles* a component can adopt. A role is the logical entity needed to contribute to the ensemble while a component provides the technical functionalities to actually execute a role. Like that, we separate the basic capabilities which a component offers from the goal-directed behavior an ensemble requires from its participants. Components take over responsibility for a certain task in the ensemble by adopting roles. They therefore no longer have to combine all capabilities in a one-fits-all implementation, but extend their behavior on demand by acting in a certain role. By adopting several roles in parallel, they are even able to participate concurrently in several ensembles without predefining the composed behavior. We also address the heterogeneity of components since we allow any component with appropriate capabilities to take over responsibility for a certain role in an ensemble.

HELENA's main features are best described by looking at a certain state of an ensemble-based system modeled with HELENA. Fig. 1.2 shows a state of a system with different components and roles. The bottom layer depicts the component-based platform building the foundation of the ensemble-based system. In this system, three components `c1`, `c2`, and `c3` exist (which are of two different types `CompType1` and `CompType2`). They are passive entities in the system and merely provide their capabilities to form ensembles on top of them. The upper layer represents ensembles of collaborating roles which are adopted by the underlying components. In this case, the system employs

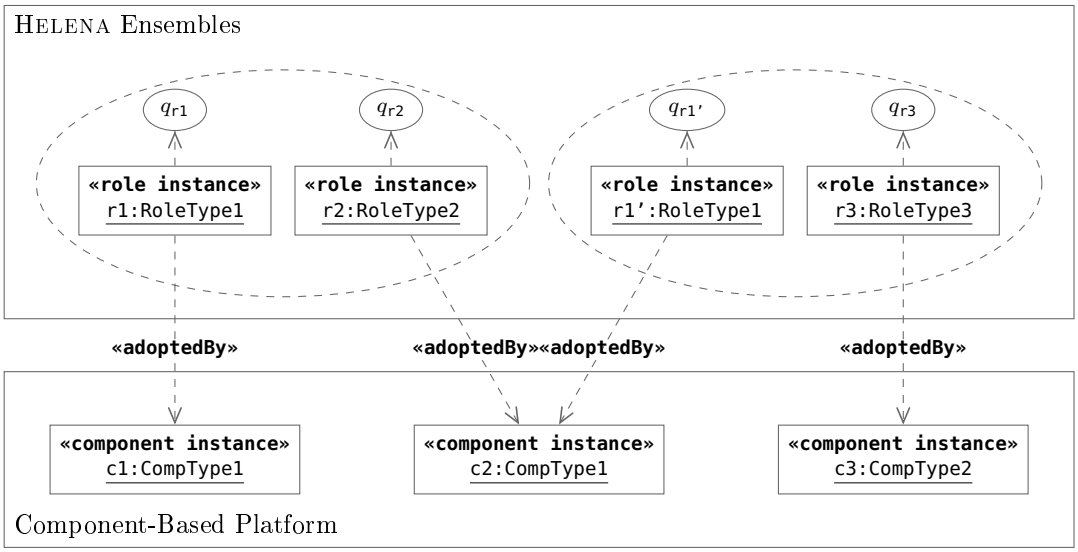


Figure 1.2: Ensemble state

two ensembles as shown by the dashed ellipses. In each ensemble, two roles (of different types `RoleType1`, `RoleType2`, and `RoleType3`) collaborate towards a common goal. These roles are the active entities in the system which actually work towards the goals. Therefore, a control state (indicated by ellipses in Fig. 1.2) is associated to each role instance which represents the current progress of the execution of the behavior of the role. To become active, a role has to be adopted by a component as shown by the dependency arrows linking to the underlying component-based platform. The components provide their resources to their adopted roles to persistently store data and to perform (possibly complex) computations.

From this picture of a state of an ensemble-based system in HELENA, the main differences between component-based design and role-based design become apparent.

Technical Functionalities vs. Goal-Oriented Behavior: The system is modeled by two layers. One layer represents the underlying component-based platform. Components merely serve as persistent data storage and offer services in the form of operations which can be called. They do not exhibit any active behavior and do not interact with each other (neither via message exchange nor via operation calls). Thus, in contrast to a component-based design, components in a role-based design are always passive objects which simply provide their resources, may it be storage or computing power, upon request from a role. The second layer represents goal-oriented ensembles of collaborating roles. Roles only exist within a certain ensemble and have to be adopted by an appropriate component. In contrast to components, they have a volatile nature in the sense that they only exist as long as their comprising ensemble has not yet reached its goal. During their lifetime, they allow to store data; however, and differently from components, the data is lost as soon as the embracing ensemble terminates since the role does not persist beyond the ensemble. Furthermore, roles are the active and interacting entities in the system, i.e., they execute a dynamic behavior contributing to achieving the ensemble goal in collaboration. In their behaviors, they use three different types of actions: The first type of actions purely refers to the role layer. Roles can (synchronously or asynchronously) exchange messages between each other and store data on themselves. The second type of actions allows the role to access

its owning component and thus fuses the two layers of roles and components together. The role can store data on its owning component persistently or call operations offering a particular service. The latter action is the only way to involve a component into the goal-oriented behavior of the ensemble. By calling an operation on the component, a role requests a particular service offered by the operation from the component and temporarily hands over the control flow to the component. Finally, the third type of actions supports management operations for roles. Roles can create other roles and retrieve references to them in order to communicate with them. This kind of dynamism stands out since it provides an autonomic and flexible way of managing the composition of an ensemble.

Ensemble Views: The upper goal-directed layer is organized in ensembles. This allows to model each collaboration from the viewpoint of a single ensemble only and independently from other ensembles. Each ensemble merely has to focus on its constituent participants in the form of roles. Thereby, each role represents one particular behavior required in the collaboration of an ensemble. In contrast to a component-based design, the component is therefore not a collection of all offered behaviors, but rather is structured by its adopted roles.

The HELENA development methodology is based on a rigorous typing discipline which distinguishes between types and instances. Fig. 1.2 is only concerned with the instance level. It shows a snapshot of the currently existing component instances of the underlying component-based platform and shows how ensembles are formed and executed from active role instances on top of it. However, these instances are rigorously typed and are only allowed to interact according to the conceptual relationships between their types defined on type level. On the one hand, we specify the properties and capabilities of types on this type level. For example, a component instance of a certain component type is able to persistently store certain data items in attributes and can offer particular services through operations. A role instance of a certain role type is able to store volatile data items in attributes and can communicate with instances of other role types through certain messages. On the other hand, the type level determines the conceptual relationships between those types in an ensemble structure. For example, the ensemble structure specifies which component types can actually adopt a certain role type and which role types must interact to form a goal-oriented ensemble.

1.5 Contributions

The contributions of this thesis lead to the HELENA development methodology for ensemble-based systems (cf. Fig. 1.3). The methodology addresses the particular challenges of ensemble-based systems in the most prominent development phases: domain modeling, goal specification, design, verification, and implementation.

Let us first summarize the HELENA development methodology in general before we introduce the particular contributions which this thesis provides to each of the phases. In the first step, the domain of an ensemble-based system is described as an ensemble structure building on top of a component-based platform. The ensemble structure captures the properties and capabilities of all participants of an ensemble and the structural relationships between them. Based on the domain model, the goals of the ensemble are specified as linear temporal logic (LTL) formulae. The goal specification thereby formalizes in terms of formulae over the participants' properties which state each participant should reach or maintain. With these goals in mind, the dynamic behavior

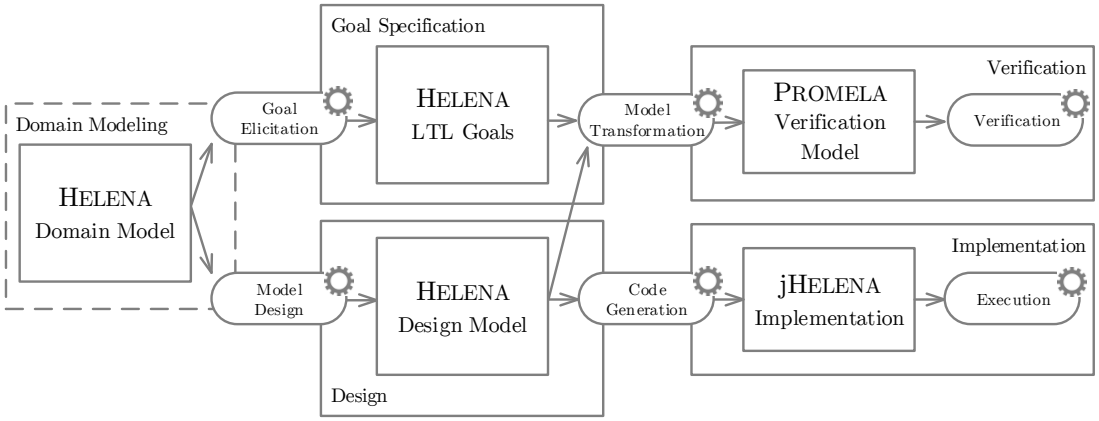


Figure 1.3: The HELENA development methodology for ensemble-based systems

of the system is designed as a set of behaviors for the participants. Together with the ensemble structure as the system's structural model, these behaviors form the HELENA design model. To show that the specified behaviors of the ensemble participants actually achieve the global goals of the ensemble (in the context of the ensemble structure), the HELENA design model is then verified against the goals. For that purpose, we translate the HELENA LTL goals and the HELENA design model to PROMELA, the input language of the model-checker Spin, and check goal satisfaction in the translated PROMELA verification model. On the other hand, we also execute those HELENA models after verification. We provide the Java framework jHELENA which realizes all HELENA concepts in Java. By implementing the HELENA design model with this framework, we are able to realize the model using the HELENA abstractions. Both translations, from HELENA to PROMELA and to jHELENA are supported by automated code generators.

The Modeling Approach of HELENA: Syntax and Semantics

In HELENA, we employ a formal modeling approach building on a component-based model of the underlying system. The basic entities of the model are components specified by their type which provide common capabilities available across all the roles the component can adopt. On top of that, we define roles, more precisely role types, which are able to take over responsibility for a certain part of the ensemble task. Each role type must be supported by at least one component type whose instances are able to adopt that role. The role types add role-specific attributes and communication abilities. To define the structural characteristics of collaborations, we use ensemble structures. They define which role types are needed in a collaboration and determine which role types may exchange which message types. Besides the structural relationships in an ensemble, we specify the dynamic behavior of each role type in a special kind of process algebra. We use standard process constructs like termination, action prefix, guarded choice, and process invocation to form process expressions. However, specific actions address the particular properties of an ensemble-based system. We support role creation and retrieval on top of a component-based platform, communication between roles by (synchronous or asynchronous) message exchange, but also communication with the underlying components via operation calls. The complete ensemble specification defines an ensemble structure together with behavior specifications for all involved roles, thus determining the collaboration needed to solve a specific task. Although all behaviors of the contributing roles are predefined, the collaboration remains adaptive since role

behaviors allow to start new roles on demand and therefore to change the composition of the ensemble at runtime.

The evolution of an ensemble (and therefore its semantics) is described by labeled transition systems. The states of the semantic labeled transition system describe the currently existing role instances and which component instances currently adopt which roles. Transitions between ensemble states are triggered by role instance creation or retrieval, (synchronous or asynchronous) message exchange between roles, and access to the underlying components. The semantic labeled transition system follows general preconditions for firing transitions and postconditions determining the effects of dynamically changing ensemble compositions. Structural operational semantic rules define the allowed transitions.

Goal Specification and Verification for HELENA Models

Exploiting the formal semantics of HELENA, we propose how goals can be specified and checked for a given HELENA model. A goal can be an "achieve goal" such that the ensemble will terminate when the goal is reached, or a "maintenance goal" such that a certain property is maintained while the system is running. We specify those goals with linear temporal logic (LTL). The basic atomic propositions for HELENA LTL goals can refer to data stored on roles or components as well as states reached in the behavior of roles. Representing the HELENA semantics in PROMELA, the input language for the explicit state model-checker Spin, we are able to verify those LTL properties in a given HELENA model. For that, we define a formal translation from HELENA to PROMELA. Furthermore, we formally prove for a subset of HELENA that its PROMELA translation satisfies the same set of LTL formulae such that model-checking results can be transferred from PROMELA to HELENA.

Implementing HELENA Models with Object-Orientation

To express role-based HELENA models with object-orientation, we have to realize the two key principles underlying HELENA. For each ensemble view, we introduce a separate container composing all participants of the collaboration and allowing communication between the members. To separate the technical functionalities from goal-oriented behavior when contributing to an ensemble, we slice the realization of an ensemble into components and roles. To make roles active, they are implemented as Java threads on top of a component. Role instances are bound to specific ensemble containers while components can adopt many roles in different concurrently running ensembles. As a proof of concept, we provide the jHELENA framework, a Java implementation of the HELENA syntax and semantics following the ideas for expressing role-based model elements with object-oriented concepts. The framework consists of two layers, a `metadata` layer and a `developer interface`, and an orthogonal system manager as shown in Fig. 1.4. The `metadata` layer allows to define ensemble specifications in terms of component types and ensemble structures (and thus role types etc.). The `developer interface` provides the basic functionality to realize an actual ensemble-based application and implements the execution semantics of HELENA. The system manager is responsible to instantiate ensemble structures, to create the underlying component-based platform, and to create and run ensembles on top of it.

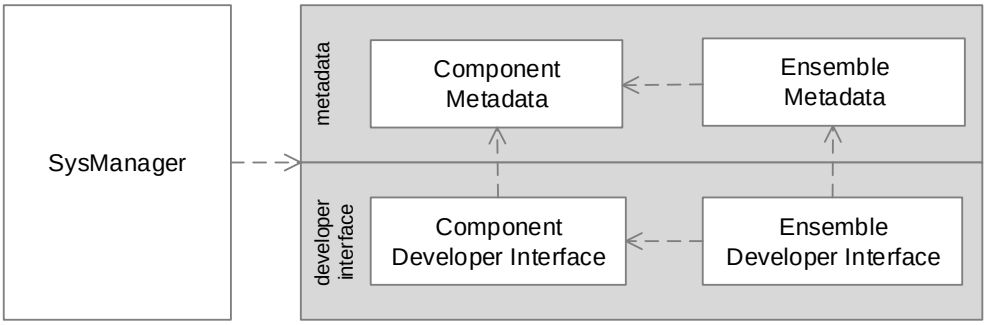


Figure 1.4: Architecture of the jHELENA framework

Tool Support for HELENA

To support the whole development cycle of ensemble-based systems, we provide an Eclipse plug-in for specification, verification and execution of ensembles. The domain-specific language HELENATEXT provides a concrete syntax for ensemble specifications supporting roles and ensembles as first-class citizens. Building on XTEXT, it is fully integrated into Eclipse providing a HELENATEXT editor with syntax highlighting, content assist, and validation. From a HELENATEXT specification either a verification model in PROMELA or an execution model in Java can be generated. The model transformation from HELENATEXT to PROMELA realizes the rules introduced in the formal translation from HELENA to PROMELA in XTEND. The generated PROMELA verification model can be used directly for verification with the model-checker Spin. The code generation from HELENATEXT to Java, also written in XTEND, generates executable code for the jHELENA framework.

Case Study

As a proof of concept, the HELENA development methodology is illustrated by a small peer-to-peer example for distributed file sharing throughout the thesis. However, to show general applicability, we provide a larger case study in the field of voluntary peer-to-peer cloud computing. The case study builds on a platform of distributed, voluntarily provided computing nodes. The nodes interact in a peer-to-peer manner to execute, keep alive, and allow use of user-defined software applications. Starting from the description of the case study, we derive an ensemble structure describing the domain and formulate HELENA LTL goals to capture the main purpose of the system. Based on the ensemble structure and the goals, we develop an ensemble specification with goal-directed behaviors for all employed roles. Goal satisfaction is checked for the HELENA model using the model-checker Spin and we report on experiences of model-checking in terms of usability, performance and memory consumption. Finally, we realize the model with HELENA concepts gaining a clear and easy to understand implementation from the encapsulation of responsibilities in roles.

Modeling Awareness and Adaptation with HELENA

We show that the concept of roles is also well-suited to model self-adaptive systems situated in an environment. A self-adaptive component keeps track of its individual and shared goals, perceives its internal state as well as its environment, and adapts its behavior accordingly. Such adaptations result in changing the behavioral mode in response to perceptions. Following the idea of roles, different behavioral modes

can be realized as roles. Changing the behavioral mode then means to adopt another role. Therefore, we propose a rigorous methodology to develop self-adaptive systems extending our HELENA development methodology (cf. Fig. 1.5).

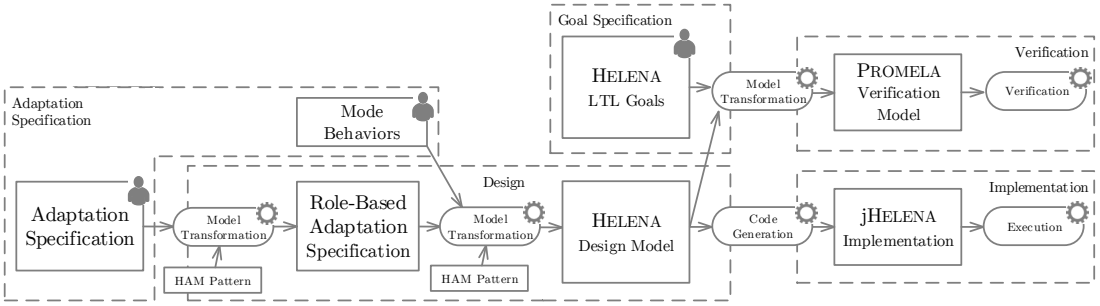


Figure 1.5: The HELENA development methodology for self-adaptive systems

We start from an adaptation specification. It defines the problem domain by employing a so-called signature of the self-adaptive system and the triggers (*when*) and actions (*what*) for self-adaptation by employing a so-called adaptation automaton. To realize the self-adaptive system, we propose the HELENA Adaptation Manager (HAM) pattern as a blueprint for *how* the system adapts and *how* the application logic is performed. By applying this pattern, the adaptation specification is transformed to a role-based adaptation specification. The key idea of the transformation is to express different behavioral modes of a component by roles, thus encapsulating independent parts of the application logic in self-contained roles. Being aware of the environment is realized monitoring awareness data via dedicated sensor roles. Furthermore, an adaptation manager – yet another role of the component – controls switching between different mode roles based on the adaptation automaton. Relying on the role-based adaptation specification, the application logic executed in each behavioral mode is specified by a set of mode behaviors for each mode role. Finally, we integrate the role-based adaptation specification with the mode behaviors by a second model transformation and gain a full HELENA design model with an ensemble structure describing the proposed role-based architecture of the self-adaptive system and a set of role behaviors for all contributing roles, behavioral modes, sensors and the adaptation manager. This model can then be analyzed and executed with the verification and implementation techniques and tools of HELENA as described above. All steps in the HELENA development methodology are illustrated at an adaptive robotic search-and-rescue scenario.

1.6 Outline

The remainder of this thesis is structured as follows:

Chap. 2: Syntax – Speaking HELENA: The formal syntax for the specification of HELENA models is introduced in this chapter. An ensemble structure describes the structural relationships between participants of an ensemble and a particular process algebra is used to specify the dynamic behavior of participants.

Chap. 3: Semantics – Understanding HELENA: This chapter defines the formal semantics for ensemble specifications. The semantic domain is labeled transition systems which evolve ensemble states through role creation and retrieval, message exchange between roles, and access to the underlying component-based platform.

Chap. 4: Goal Specifications – Being Successful with HELENA: Ensembles are formed to collaborate towards a global goal. In this chapter, we introduce HELENA LTL, a logic based on *LTL*with HELENA specific atomic propositions. With this logic, global goals can be specified for a HELENA ensemble specification.

Chap. 5: Verification – Being Sure about Goal Satisfaction: In this chapter, analysis techniques are proposed to verify goals in HELENA models. We represent the HELENA model in PROMELA and verify the resulting model against the given properties by employing the explicit state model-checker Spin.

Chap. 6: Correctness Proof – Allowing HELENA to Rely on Spin: To be able to transfer model-checking results from PROMELA to HELENA, we prove semantic equivalence between the two specifications. In this chapter, we formally show that the semantic labeled transition systems of an ensemble specification in a simplified version of HELENA and its PROMELA translation are stutter trace equivalent. They therefore satisfy the same set of LTL formulae such that our approach of model-checking with Spin is correct.

Chap. 7: Implementation – Vivifying HELENA with jHELENA: In this chapter, we show how the role-based modeling elements in HELENA can be expressed through object-oriented concepts. The jHELENA framework serves as a proof of concept of the translation from HELENA to Java.

Chap. 8: HELENA Workbench – Working with HELENA: To support the development with HELENA in practice, we present the HELENA workbench in this chapter. The workbench is fully included into Eclipse and allows to specify HELENA models in the dedicated domain-specific language HELENATEXT. For verification, HELENATEXT models are automatically transformed to PROMELA to serve as input for the model- checker Spin. For execution, Java code is generated which relies on the jHELENA framework to express all HELENA concepts as first-class entities in Java.

Chap. 9: HELENA Development Methodology – Developing with HELENA: In this chapter, we combine all techniques from the previous chapters together resulting in the rigorous HELENA development methodology for ensemble-based systems which has already been sketched before.

Chap. 10: HELENA@Work – Applying HELENA to the Science Cloud Platform: We apply HELENA to a larger case study from the field of voluntary peer-to-peer cloud computing. The HELENA development methodology is exemplarily exercised at the case study and we present a real-life implementation. The implementation combines the HELENA concepts and their object-oriented realization with underlying network and communication technologies.

Chap. 11: Role-Based Adaptation – Being Adaptive with HELENA: We propose a rigorous methodology to develop self-adaptive systems from specification to design. We specify the system's adaptation logic through adaptation automata. The design is realized in HELENA refining the specification by providing a role-based architecture and adding application logic in terms of role behaviors for different modes of the system.

Chap. 12: Conclusion: Lastly, we conclude by providing a résumé of our experiences with HELENA. We summarize advantages, challenges, and limitations of applying it and suggest further research directions in order to progress HELENA.

The appendices contain the detailed correctness proof which was only outlined in Chap. 6, a user guide for the HELENA workbench and the specification and generated source codes of the p2p example, the science cloud case study and the search-and-rescue scenario.

Before the bibliography, the author's publications are listed separately including statements about her contributions to joint works used in this thesis. Moreover, in each chapter, the section on "publication history" explains which publications provided the basis for the content of the chapter and how they were extended for this thesis. Related work, a summary and future perspectives are discussed for each chapter separately.

Chapter 2

Syntax
Speaking HELENA

The role-based modeling approach HELENA provides concepts to describe systems of a large number of *components* which dynamically team up in possibly concurrently running *ensembles* to perform global goal-oriented tasks. To participate in an ensemble, a component adopts a certain *role*. This role adds role-specific behavior to the component and allows collaboration with other components playing roles. By switching between roles, a component changes its currently executed behavior. By adopting several roles in parallel, a component concurrently executes different behaviors.

HELENA ensemble specifications describe the structural and dynamic properties of an ensemble on type-level. To define the structure of an ensemble, an *ensemble specification* determines which types of roles have to participate in an ensemble, which types of components are allowed to adopt the roles, and how many instances of each role type can collaborate in the ensemble. To define the dynamic behavior of an ensemble, each role type is equipped with a *role behavior* which is later on executed by each existing instance of the role type possibly interacting with other role instances.

In Sec. 2.1, we introduce a peer-to-peer file sharing scenario which is used as a running example throughout this thesis. Afterwards, we outline the syntax of HELENA ensemble specifications. We start by describing the specification of the underlying component-based platform in Sec. 2.2. In Sec. 2.3, we introduce ensemble structures which determine the structural relationships in a collaboration of components by roles. To gain a complete ensemble specification, each role type is equipped in Sec. 2.4 with a dynamic role behavior specified as a process term. We conclude with related work on the concepts used in HELENA in Sec. 2.5 and a short summary in Sec. 2.7. All subsections are illustrated with the p2p example introduce in the previous chapter. The full ensemble specification of the p2p example can be found in Appendix C.1.

Notation: Whenever we work with tuples $t = (t_1, \ldots, t_n)$, we may use the notation $t_i(t)$ to refer to the value t_i of t. If t_1 denotes a name, we often write t synonymously for the name t_1.

Notation: $\vec{z}$ denotes a list of z.

2.1 P2P Example

Throughout this thesis, we illustrate the HELENA concepts and tools based on the example of a peer-2-peer (p2p) network supporting the distributed storage of files which

can be retrieved upon request. Several peers form the underlying component-based platform. They are connected in a ring structure and are able to store files and to print their contents. Whenever a file is requested, some peers work together to find the file in the network and to transfer it to the requester. To this end, an appropriate ensemble is formed whose goal is to finally deliver the requested file to the requesting peer. Three different roles are necessary in the ensemble: One peer plays the role of the requester of the file, other peers act as routers and the peer storing the requested file adopts the role of the provider of the file. All these roles can be adopted by peers and rely on their capabilities to store files and print them. In the following chapters, we will derive the full specification of this example in HELENA. We will formally define the goal of file transferal and check it in the ensemble specification. Finally, we provide an executable implementation of the example based on the jHELENA framework.

2.2 Component-Based Platform

Ensembles are built on top of a *component-based platform*. The component-based platform describes the architectural and functional properties of the underlying target system on which ensembles are dynamically formed. Hence, the components provide the persistent foundation of ensemble-based systems. Components store data and maintain associations to other components. Both, data and associations, persist across the life-time of different dynamically evolving ensembles. They additionally provide their computing resources in the form of operations which can be called by the roles which components adopt in ensembles.

Therefore, components are pure data containers and computing resources without any active behavior. They are just passive objects on which the active entities, the roles, rely on for storing persistent data or performing computations. Components are not meant to exchange any messages or data between each other which the adopted roles are responsible for.

To represent data in HELENA, we assume given a set $\mathcal{D}$ of *data types* (like `bool` for boolean values, `int` for integer values, or `double` for floating-point numbers). The semantic domain of a data type $dt \in \mathcal{D}$ is given by $[\![dt]\!]$. To store persistent data, each component type offers a set of *attributes*. Attributes are characterized by their name and typed by a data type from $\mathcal{D}$.

Def. 2.1: Attribute

An attribute *is of the form atnm:dt such that atnm is the name of the attribute and $dt \in \mathcal{D}$ its data type.*

To store connections to other components, each component type offers a set of directed *component associations*. An association is identified by its name and the type of the component to which this association refers to.

Def. 2.2: Component Association

A component association *is of the form assocnm:ct such that assocnm is the name of the association and ct the component type referred to by this association.*

To provide computing resources, each component type offers a set of *operation types* which can be called by role instances later on. An operation is identified by its name

and a list of typed formal data parameters for inputs. The effect of the operation is not further specified by the operation type (e.g., by pre- and post-conditions), that means the operation could have side-effects. An operation can just use the resources of a component to compute something.

Def. 2.3: Operation Type

An operation type *op is of the form* $opnm(\overrightarrow{x : dt})$ *such that opnm is the name of the operation type and* $\overrightarrow{x : dt}$ *is a list of formal parameters for data with type* $\overrightarrow{dt}$.

To classify components, we use *component types*. Each component type is identified by its name and offers a set of attributes to store data, a set of component associations, and a set of operation types which can be called on and executed by (instances of) the component type. Note that by using sets, we consider all attributes, component associations, and operations unique per component type.

Def. 2.4: Component Type

A component type *ct is a tuple* $(ctnm, ctattrs, ctassocs, ctops)$ *such that ctnm is the name of the component type, ctattrs a set of attributes, ctassocs a set of component associations, and ctops a set of operation types.*

Example: In the p2p example introduced in Sec. 2.1, we employ a set of components connected in a ring structure. They support the distributed storage of files and allow to retrieve the files upon request. For simplicity, we only consider one single file which is stored and exchanged in the network. All components in the network are of the same type *Peer*. Formally, the component type for a peer is given by

$$Peer = (\text{"}Peer\text{"}, \{hasFile{:}\texttt{bool}, content{:}\texttt{int}\}, \{neighbor{:}Peer\}, \{printFile()\}).$$

The component type *Peer* has the name "*Peer*". The attribute *hasFile* of type `bool` indicates whether the peer has the file independently from the file's content information represented by the attribute *content* of type `int` (we assume that the content can be stored as an integer). A peer is furthermore connected to its neighboring peer given by the association *neighbor*:*Peer*. Lastly, a peer can print the content of the file by executing the operation *printFile* which does not have any parameters. Note that the effect of the operation is not specified by the operation type, i.e., printing the file is just the intuitive meaning of the operation *printFile*.

For visualization, we use a graphical representation inspired by UML class diagrams which is depicted in Fig. 2.1. It consists of three parts: the name of the component annotated with the stereotype «`component type`», the component attributes, and the component operations. Component associations are shown by arrows pointing to component types.

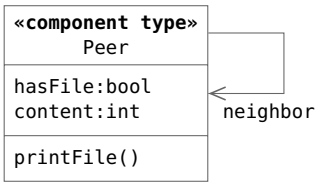


Figure 2.1: Component type *Peer* in graphical notation

2.3 Ensemble Structures

For becoming active and performing certain tasks, components team up in ensembles. Each participant in the ensemble contributes specific functionalities to the collaboration, we say, the participant plays a certain *role* in the ensemble. Roles are the active entities in the system and collaborate with each other to reach a goal, possibly commonly shared in the ensemble.

2.3.1 Role Types

For collaboration, roles exchange messages which are classified by their type. Each *message type* has a name, a list of formal parameters to pass references of role instances for further communication, and a list of formal parameters to pass ordinary data.

Def. 2.5: Message Type

A message type *msg is of the form* $msgnm(\overrightarrow{X : rt})(\overrightarrow{x : dt})$ *such that msgnm is the name of the message type,* $\overrightarrow{X}$ *is a list of formal role instance parameters with type* $\overrightarrow{rt}$*, and* $\overrightarrow{x : dt}$ *is a list of formal data parameters with type* $\overrightarrow{dt}$*.*

Roles themselves are also classified by their type. The definition of a *role type* always builds upon a given component-based platform specified by a set of component types. A role type determines the types of the components that are able to adopt this role. It also defines a set of role-specific attributes to store data which is only relevant while performing the role (and hence is volatile just as role-playing), and sets of message types for outgoing and incoming messages for interaction and collaboration with other roles. Note that by using sets, we assume that all attributes and messages types as well as adopting component types are unique per role type.

Def. 2.6: Role Type

Given a set CT of component types, a role type *rt over CT is a tuple* $rt = (rtnm, rtcomptypes, rtattrs, rtmsgs_{out}, rtmsgs_{in})$ *such that*

- *rtnm declares the name of the role type*[a]*,*
- $rtcomptypes \subseteq CT$ *is a finite, non-empty set of component types whose instances can adopt the role,*
- *rtattrs is a set of role-specific attributes,*
- $rtmsgs_{out}$ *and* $rtmsgs_{in}$ *specify sets of message types for outgoing and incoming messages resp. supported by the role type.*

[a] In the following, we often write *rt* synonymously for the name *rtnm* of the role type.

Example: In the context of our p2p network, we consider the task of requesting and transferring a file. To perform this task, we envision three role types: requester, router, and provider. The requester wants to download the file from the network. Thus, it requests the address of the peer storing the file from the network, while using the routers as forwarding peers of its request. Once the requester knows the address, it directly requests the file from the provider for download. Each role can be adopted by instances of component type *Peer*, but exhibits different capabilities to take over

responsibility for the transfer task: the requester must be able to request the address of the provider from a router and receive the reply. Afterwards, it must be able to request the file from the provider and receive the content. The router must be able to receive a request for the address, to send a reply back to the requester or to forward it to another router. The provider of the file must be able to receive a request for the file and send the content back to the requester.

The formal specification of the three role types is given in Fig. 2.2. A *Requester* has the name "*Requester*" and can be adopted by a component instance of type *Peer*. It has a role-specific attribute *hasFile* of type `bool` to store whether it already received the requested file or not. To request the address of the provider, it supports the message *reqAddr*(*req*:*Requester*)() as outgoing message. The parameter *req* is used to transfer the reference of the requesting peer to the receiver of this message such that an answer can be sent back to the requesting peer via this reference. To receive the address of the provider, the requester supports the message *sndAddr*(*prov*:*Provider*)() as incoming message. The parameter *prov* thereby holds the reference of the provider of the file. Downloading the file is requested with the message *reqFile*(*req*:*Requester*)() supported as outgoing message by the requester. The actual content is transferred to the requester with the incoming message *sndFile*()(*content*:`int`) where the parameter *content* of type `int` holds the actual content. The other two role types support the corresponding messages, e.g., the *Router* supports receiving a request for the address of the provider, replying to the request and forwarding the request while the *Provider* supports receiving a request for the content of the file and transferring the file's content.

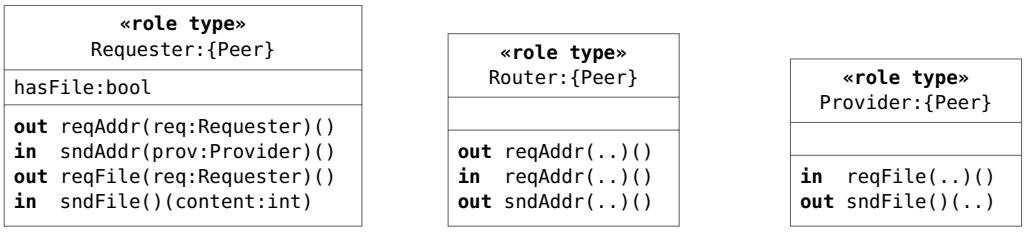
$$
\begin{aligned}
&\textit{Requester} && = (\text{"}\textit{Requester}\text{"}, \{\textit{Peer}\}, \{\textit{hasFile}\text{:}\texttt{bool}\}, \textit{msgs}_{out}(rq), \textit{msgs}_{in}(rq))\\
&\quad \text{with } \textit{msgs}_{out}(rq) && = \{\textit{reqAddr}(\textit{req}\text{:}\textit{Requester})(), \textit{reqFile}(\textit{req}\text{:}\textit{Requester})()\}\\
&\quad \text{and } \textit{msgs}_{in}(rq) && = \{\textit{sndAddr}(\textit{prov}\text{:}\textit{Provider})(), \textit{sndFile}()(\textit{content}\text{:}\texttt{int})\}\\
&\textit{Router} && = (\text{"}\textit{Router}\text{"}, \{\textit{Peer}\}, \emptyset, \textit{msgs}_{out}(ro), \textit{msgs}_{in}(ro))\\
&\quad \text{with } \textit{msgs}_{out}(ro) && = \{\textit{reqAddr}(\textit{req}\text{:}\textit{Requester})(), \textit{sndAddr}(\textit{prov}\text{:}\textit{Provider})()\}\\
&\quad \text{and } \textit{msgs}_{in}(ro) && = \{\textit{reqAddr}(\textit{req}\text{:}\textit{Requester})()\}\\
&\textit{Provider} && = (\text{"}\textit{Provider}\text{"}, \{\textit{Peer}\}, \emptyset, \textit{msgs}_{out}(pv), \textit{msgs}_{in}(pv))\\
&\quad \text{with } \textit{msgs}_{out}(pv) && = \{\textit{reqFile}(\textit{req}\text{:}\textit{Requester})()\}\\
&\quad \text{and } \textit{msgs}_{in}(pv) && = \{\textit{sndFile}()(\textit{content}\text{:}\texttt{int})\}
\end{aligned}
$$

Figure 2.2: All role types for the p2p example

A graphical representation of all three role types is given in Fig. 2.3. Similarly to the graphical representation of component types, it again consists of three parts inspired by UML class diagrams: The first compartment specifies the name of the role type annotated with the stereotype «`role type`». The notation `RoleType:{Peer}` indicates that any component instance of type `Peer` can adopt the role `RoleType`. The second compartment specifies role-specific attributes and the third compartment lists all supported messages together with the modifiers `in` and `out`.

2.3.2 Ensemble Structures

To define the structural characteristics of a collaboration, an *ensemble structure* determines the type of an ensemble which is able to perform a certain task. It specifies the



(a) Role type *Requester* **(b)** Role type *Router* **(c)** Role type *Provider*

Figure 2.3: All role types for the p2p example in graphical notation

set of role types which are needed in the collaboration and how many instances of each role type may (or must) contribute (note that by using a set of role types, we assume that role types are unique per ensemble structure). The roles contributing to the ensemble can then exchange messages which are outgoing at the source role and incoming at the target role. Interacting role instances can use synchronous or asynchronous communication via input queues. An ensemble structure specifies, for each role type, the (finite) capacity of the input queue of each role instance of that type where the value 0 expresses synchronous communication. An ensemble structure is always built on top of a given set CT of component types whose instances can adopt roles as specified in the ensemble structure.

Def. 2.7: Ensemble Structure

Let CT be a set of component types. An ensemble structure Σ over CT *is a tuple* $\Sigma = (nm, roletypes, roleconstraints)$ *such that*

- *nm is the name of the ensemble structure,*
- *$roletypes$ is a set of role types over CT and for each $rt \in roletypes$,*
- *$roleconstraints(rt)$ is a triple of*
  - *the minimal number $min \in \mathbb{N}^+$ of role instances of rt,*
  - *the maximal number $max \in \mathbb{N}^+ \cup \{*\}$ of role instances of rt (* refers to the UML multiplicity of arbitrarily many instances), and*
  - *the finite capacity $cap \in \mathbb{N}$ of the input queue of rt.*

An ensemble structure Σ is called *closed ensemble structure* if each outgoing message type supported by some role type of Σ is matched by an incoming message type supported by any (possibly the same) role type of Σ, i.e.,

$$\forall rt \in roletypes(\Sigma), msg \in rtmsgs_{out}(rt)\ .\ \exists rt' \in roletypes(\Sigma)\ :\ msg \in rtmsgs_{in}(rt') \wedge$$
$$\forall rt \in roletypes(\Sigma), msg \in rtmsgs_{in}(rt)\ .\ \exists rt' \in roletypes(\Sigma)\ :\ msg \in rtmsgs_{out}(rt').$$

Otherwise the ensemble structure is called *open*. In the following, we only consider closed ensemble structures.

Example: The ensemble structure for the p2p example consists of the three role types *Requester*, *Router*, and *Provider* where we now associate a minimal and maximal number for the allowed instances per role type and capacities for their input queues. Its formal representation is given in Fig. 2.4

For instance, exactly one instance of the role type *Requester* is required in a file transfer ensemble while arbitrarily many instances of the role type *Router* might be

$$\Sigma_{transfer} = (\text{"}\Sigma_{transfer}\text{"}, \{Requester, Router, Provider\}, roleconstraints)$$
$$\text{with } roleconstraints(Requester) = (1, 1, 2),$$
$$roleconstraints(Router) = (1, *, 2),$$
$$roleconstraints(Provider) = (0, 1, 1).$$

Figure 2.4: Ensemble structure $\Sigma_{transfer}$ for the p2p example

necessary to route messages through the network. The input queue of an instance of the role type *Requester* or *Router* can store up to two messages, of an instance of the role type *Provider* only one message. Which messages can be exchanged between each of the role types is implicitly given by the intersection of outgoing and incoming message types of two role types.

Fig. 2.5 shows a graphical representation of the ensemble structure $\Sigma_{transfer}$ for the p2p example. This graphical representation makes some of the implicitly specified properties more explicit. Firstly, it depicts by dependency arrows with the stereotype «adoptedBy» that each role type in the p2p example can be adopted by the component type `Peer`. Secondly, minimal and maximal numbers, and capacities are explicitly shown in a separate compartment of each role type. Lastly, arrows between role types denote sets of messages which can be exchanged between the roles, i.e., which are outgoing for the source role and incoming for the target role. For instance, the `Requester` can send the message `reqAddr(req:Requester)()` to a `Router`. This message will be used for requesting the address of a `Provider` for the requested file such that the file can be directly downloaded afterwards using the messages between `Requester` and `Provider`.

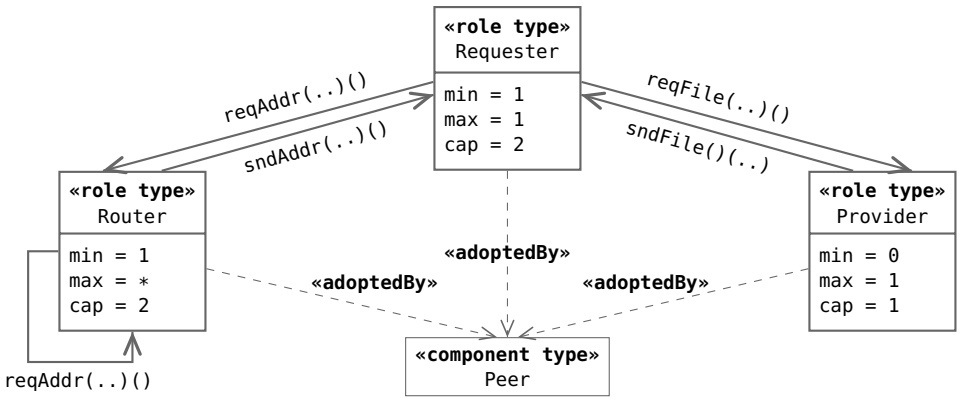


Figure 2.5: Ensemble structure $\Sigma_{transfer}$ for the p2p example in graphical notation

2.3.3 Well-Formedness of Ensemble Structures

An ensemble structure has to satisfy two conditions to be well-formed. Firstly, the ensemble structure determines the minimal number and maximal number of instances for each role type. Thus, we must take care that those two numbers are ordered. Secondly, in the context of an ensemble structure, only the role types named in the ensemble structure are known. Thus, the messages exchanged between role types of the ensemble structure can only refer to those known role types as parameters. These conditions are expressed by well-formedness of ensemble structures.

Def. 2.8: Well-Formedness of Ensemble Structures

An ensemble structure $\Sigma = (nm, roletypes, roleconstraints)$ is well-formed *if for all role types* $rt \in roletypes(\Sigma)$,

(1) the minimal number of allowed role instances is less or equal to the maximal number of allowed role instances,
i.e., $min(roleconstraints(rt)) \leq max(roleconstraints(rt))$, *and*

(2) all types rt' *of role instance parameters of all message types* $msg \in rtmsgs_{out}(rt) \cup rtmsgs_{in}(rt)$ *are part of the ensemble structure* Σ,
i.e., $rt' \in roletypes(\Sigma)$.

Example: The ensemble structure $\Sigma_{transfer}$ for the p2p example shown in Fig. 2.4 resp. Fig. 2.5 is well-formed according to Def. 2.8. Firstly, the minimal and maximal number are ordered. Secondly, from Fig. 2.2 resp. Fig. 2.3, we know that only the message types *reqAddr*(*req*:*Requester*)(), *sndAddr*(*prov*:*Provider*)(), and *reqFile*(*req*:*Requester*)() have role instance parameters. Their role types *Requester* and *Provider* are part of the underlying ensemble structure $\Sigma_{transfer}$.

2.4 Ensemble Specifications

After having modeled the structural aspects of ensembles, we focus on the specification of behaviors for each role type of an ensemble structure. A role behavior declaration is given by a process expression and a set of process declarations which can be called in that process expression.

2.4.1 Process Expressions

In HELENA, *process expressions* are built from termination, action prefix, nondeterministic choice, if-then-else, and process invocation. Note that HELENA process expressions do not support parallel composition. In HELENA, a role is always responsible for a specific task which we consider to be achieved by a (possibly branching) sequential process. If a component is meant to perform two tasks in parallel, it has to adopt two roles in parallel and therefore to concurrently execute their sequential role behaviors. The HELENA semantics presented in the next chapter will support this kind of concurrency since it formalizes the concurrent execution of all roles of a component in the context of an ensemble.

There are eight different kinds of actions which can be used in action prefix: creation and retrieval of role instances, sending (!) and receiving (?) a message, operation calls on the owning component, setting attribute values for the role instance or the owning component[1], and state labels. Intuitively, these actions must fit to the declared ensemble structure, e.g., messages can only be sent by roles which declare them, as we discuss it more formally in the following subsection about well-formedness of process expressions.

Guards used in if-then-else constructs are the boolean primitives **true** and **false**, data variables (e.g., from a message reception or the result value of an operation call), access of component attributes or role attributes, the predefined query **plays**(rt, C) to request whether the component instance identified by C currently plays the role rt,

[1] Getters are not provided as distinct actions, but the values of attributes can be retrieved and accessed in data expressions as we will present in the following.

data expressions (e.g., summation and subtraction or relations comparing expressions and constants with the usual relational operators) or any boolean expression built from these primitives using propositional operators like ! and &&. In the following subsection, we will formalize well-formedness of guards to determine that they are actually boolean.

Notation: We use the special, predefined role instance variable **self** to refer to the current role instance and the special, predefined component instance variable **owner** to refer to the owning component instance. Furthermore, the notation **owner**.*assocnm* is used to refer to the name of the component association *assocnm*:*ct* of the owning component instance.

In the following definition, P, P_1, and P_2 are process expressions, a is an action, *guard*, $guard_1$, and $guard_2$ are guards, N is the name of a declared process, rt is the name of a role type, C is either the special, predefined variable **owner** or **owner**.*assocnm* referring to the name of a component association of the owning component instance, *msgnm* is the name of a message type, *opnm* is the name of an operation type, *attr* is the name of an attribute (either of a component type or a role type), X and Y are role instance variables (including the special, predefined variable **self**), x is a data variable[2], and e is a data expression.

Def. 2.9: Process Expression

A process expression P *is built from the following grammar:*

$P ::=$	***quit***	*(termination)*
	$\mid a.P$	*(action prefix)*
	$\mid P_1 + P_2$	*(nondeterministic choice)*
	$\mid$ ***if*** $(guard)$ $\{P_1\}$ ***else*** $\{P_2\}$	*(if-then-else)*
	$\mid N$	*(process invocation)*
$a ::=$	$X \leftarrow$***create***(rt, C)	*(role instance creation)*
	$\mid X \leftarrow$***get***(rt, C)	*(role instance retrieval)*
	$\mid Y!msgnm(\overrightarrow{X})(\overrightarrow{e})$	*(sending a message)*
	$\mid ?msgnm(\overrightarrow{X : rt})(\overrightarrow{x : dt})$	*(receiving a message)*
	$\mid [x =]$ ***owner***$.opnm(\overrightarrow{e})$	*(component operation call)*
	$\mid$ ***owner***$.attr = e$	*(component attribute setter)*
	$\mid$ ***self***$.attr = e$	*(role attribute setter)*
	$\mid label$	*(state label)*
$guard ::=$	***true*** $\mid$ ***false***	*(boolean primitives)*
	$\mid x$	*(data variable)*
	$\mid$ ***owner***$.attr \mid$ ***self***$.attr$	*(attribute access)*
	$\mid$ ***plays***(rt, C)	*(plays query)*
	$\mid e$	*(data expression)*
	$\mid !guard \mid guard_1 \;\&\&\; guard_2 \mid \ldots$	*(propositional operators)*

[2]We distinguish between role instance variables and data variables since role instance variables can be used as recipients for messages later on, for instance for callbacks.

In the following, we give an intuitive meaning for each of the process constructs and actions in the context of an ensemble structure. The explanations anticipate the formal semantics in Chap. 3.

Termination is expressed by the process construct **quit**. When using a process expression to describe the behavior of a role, **quit** first advises the owning component of the role to quit playing the role and then terminates the execution of the role behavior.

Action prefix is of the form $a.P$. This process construct first executes the action a and then behaves like the remaining process P.

Nondeterministic choice is of the form $P_1 + P_2$. Nondeterministic choice in HELENA realizes external (nondeterministic) choice: The executability of the first action of either branch, the process expression P_1 and the process expression P_2, is checked. If both branches are executable, one branch is selected nondeterministically for execution. If only one is executable, this branch is selected for execution. If none is executable, the execution of the whole nondeterministic choice process construct is blocked until at least one of the branches becomes executable. Importantly, the question of executability just arises in the context of an ensemble. A single role instance can always execute its actions, but in collaboration with other role instances, some actions might not be executable, e.g., one role instance wants to send a message to another role instance which is currently not able to receive it.

If-then-else is of the form **if** (*guard*) $\{P_1\}$ **else** $\{P_2\}$. This process construct allows to choose between two process expressions P_1 and P_2 based on the value of the boolean guard *guard*. The if-then-else process construct can only be executed if either the guard evaluates to **true** and the first action of the first process expression P_1 is currently executable or the guard evaluates to **false** and the first action of the second process expression P_2 is currently executable. Otherwise the whole if-then-else process construct blocks until one of the aforementioned execution conditions holds.

Process invocation is of the form N. This process construct simply invokes the process with the name N and instantaneously continues by executing this process.

A create action is of the form $X \leftarrow \mathbf{create}(rt, C)$. It creates a new role instance of type rt which is adopted by the component instance identified by C, and referenced by the variable X of type rt in the sequel.

A get action is of the form $X \leftarrow \mathbf{get}(rt, C)$. It retrieves an existing role instance of type rt already adopted by the component instance identified by C and binds a reference to the found role instance to the role instance variable X. This actions blocks if the component instance does not currently adopt such a role. Note that there can only be one active instance of each role type per component in an ensemble.

A send action is of the form $Y!msgnm(\overrightarrow{X})(\overrightarrow{e})$. It expresses that a message with name *msgnm* and actual parameters $\overrightarrow{X}$ and $\overrightarrow{e}$ is sent to a role instance referenced by Y. The first parameter list $\overrightarrow{X}$ consists of role instances to be passed to the receiver; with the second parameter list $\overrightarrow{e}$, data is passed to the receiver. This action blocks if the message queue of the receiving role instance is full.

A receive action is of the form $?msgnm(\overrightarrow{X : rt})(\overrightarrow{x : dt})$. It expresses the reception of a message with name *msgnm*. The values received on the parameters are bound to the variables $\overrightarrow{X}$ for role instances and to the variables $\overrightarrow{x}$ for data. This action blocks if the message queue of the receiving role instance is empty.

A component operation call is of the form $[x =]$ **owner**.$opnm(\overrightarrow{e})$. It calls the corresponding operation with the name *opnm* on the owning component of the role instance with the given parameters $\overrightarrow{e}$ and binds the retrieved value to the data variable x if the operation returns a value in one atomic step. Note that in HELENA, we do not specify any effect of operations in an ensemble specification (i.e., when defining a component type with its operations). That means that semantically, an operation call can change the state of an ensemble other than progressing a role behavior; more specifically, it can have side-effects on the state of the owning component instance as we will see in the next chapter about the semantics of HELENA.

A component attribute setter is of the form **owner**.$attr = e$. It sets the value of the attribute *attr* of the owning component of the current role to the value e in one atomic step.

A role attribute setter is of the form **self**.$attr = e$. It sets the value of the attribute *attr* of the current role to the value e.

A state label is of the form *label*. It can semantically be seen as a silent non-communication action. We will only use state labels during verification to explicitly refer to a particular state in a role behavior. State labels do not contribute to the actual goal-directed behavior of a role except labeling a certain state in the role behavior.

The variables $\overrightarrow{X}, \overrightarrow{x}$ used in message reception and operation call, and the variable X for role instance creation and retrieval open a scope which binds the open variables with the same names in the successive process expression. The bound variables receive a type as declared by the role types $\overrightarrow{rt}$ and rt or $\overrightarrow{dt}$ resp.

2.4.2 Well-Formedness of Process Expressions

In the context of an ensemble structure $\Sigma = (nm, roletypes, roleconstraints)$, a process expression has to satisfy some conditions to be well-formed. First of all, actions and guards employed in the process expression have to be well-formed, i.e., they must conform to the underlying ensemble structure.

We do not allow mixed states in nondeterministic choice such that the first actions of both branches have either to be incoming messages or any other actions than an incoming message. Otherwise, it would be possible for example to nondeterministically choose between sending and receiving a message. Intuitively, sending a message can internally be decided, but receiving a message is externally triggered if a message from another role is available. Due to this different kind of decision it does not make sense to allow selecting between those two options.

Furthermore, state labels are not allowed as first actions of any branch in nondeterministic choice or if-then-else. State labels mark a certain progress of execution in a process expression. If used as a first action in nondeterministic choice, it is unclear when the designated point of progress is reached. Before executing nondeterministic

choice, the point is not yet reached since we can still decide for either of the branches. When choosing one of the branches, we immediately execute the first action such that after executing the first action the designated point is not reached either since the state label was already passed. Therefore, we cannot allow state labels as first actions of branches in nondeterministic choice. The same argumentation applies to state labels as first actions of branches in if-then-else.

Lastly, a process expression must not immediately invoke itself. An immediate recursive process invocation would not progress the process since process invocation is not a separate step in the semantics (as we will see in the next chapter), but immediately executes the first action of the invoked process.

Def. 2.10: Well-formedness of a Process Expression

A process expression P is well-formed *for a role type* $rt \in \mathit{roletypes}(\Sigma)$ *w.r.t.* Σ, *if*

(1) in any action prefix construct of P, all actions are well-formed for rt with respect to Σ,

(2) in any if-then-else construct of P, all guards are well-formed for rt with respect to Σ,

(3) in any nondeterministic choice construct of P, the first actions of the two branches are either incoming messages or any other action than an incoming message,

(4) in any nondeterministic choice construct or if-then-else construct, state labels are not the first action of any branch,

(5) a process expression does not immediately invoke itself, also not by a chain of process invocations being the first and last invocation the same.

An action is well-formed for a role type $rt \in \mathit{roletypes}(\Sigma)$ w.r.t. Σ, if

(1) for $X \leftarrow \mathbf{create}(rt', C)$ and $X \leftarrow \mathbf{get}(rt', C)$ resp.,

  (a) $rt' \in \mathit{roletypes}(\Sigma)$, i.e., rt' is a role type in the ensemble structure Σ,

  (b) C is either the special, predefined variable **owner** or **owner**.*assocnm* referring to the name of a component association $\mathit{assocnm}{:}ct'$ which is defined for all component types $ct \in \mathit{rtcomptypes}(rt)$ which can adopt a role of type rt,

  (c) the component instance identified by C is of a type $ct \in \mathit{rtcomptypes}(rt')$, i.e., the instances of the component type ct can adopt a role of type rt',

  (d) the variable X has not been declared before,

(2) for $Y!\mathit{msgnm}(\overrightarrow{X})(\overrightarrow{e})$,

  (a) $\mathit{msgnm}(\overrightarrow{X' : rt'})(\overrightarrow{x' : dt'}) \in \mathit{rtmsgs}_{out}(rt)$, i.e., the role type rt supports the message type $\mathit{msgnm}(\overrightarrow{X' : rt'})(\overrightarrow{x' : dt'})$ as outgoing message,

  (b) the type of the role instance variable Y supports the message type $\mathit{msgnm}(\overrightarrow{X' : rt'})(\overrightarrow{x' : dt'})$ as incoming message,

(c) the actual parameters $\overrightarrow{X}$ and $\overrightarrow{e}$ fit in number, ordering, and type to the formal ones $\overrightarrow{X'}$ and $\overrightarrow{x'}$,

(d) the expressions Y, $\overrightarrow{X}$, and $\overrightarrow{e}$ only name the predefined constant **self**, concrete values, or variables or parameters which have been declared before,

(3) for $?msgnm(\overrightarrow{X : rt'})(\overrightarrow{x : dt})$,

(a) $msgnm(\overrightarrow{X : rt'})(\overrightarrow{x : dt}) \in rtmsgs_{in}(rt)$, i.e., the role type rt supports the message type $msgnm(\overrightarrow{X : rt'})(\overrightarrow{x : dt})$ as incoming message,

(b) $\overrightarrow{X}$ and $\overrightarrow{x}$ have not been declared before,

(4) for $[x =]$ **owner**.$opnm(\overrightarrow{e})$,

(a) $\forall ct \in rtcomptypes(rt)\ .\ opnm(\overrightarrow{x : dt}) \in ctops(ct)$, i.e., all component types which can adopt a role of the type rt support the operation type $opnm(\overrightarrow{x : dt})$,

(b) the actual parameters $\overrightarrow{e}$ fit in number and ordering to the formal ones $\overrightarrow{x}$,

(c) $\overrightarrow{e}$ only names concrete values or variables and parameters which have been declared before,

(d) the variable x has not been declared before,

(5) for **owner**.$attr = e$ and **self**.$attr = e$ resp.,

(a) $\forall ct \in rtcomptypes(rt)\ .\ attr \in ctattrs(ct)$, i.e., all component types which are able to adopt a role of type rt provide the attribute $attr$,

(b) $attr \in rtattrs(rt)$, i.e., the role type rt provides the attribute $attr$ resp.,

(c) e must be an expression with the same type as $attr$,

(d) e only names concrete values or variables and parameters which have been declared before,

(6) all state labels are unique.

A guard is well-formed for a role type $rt \in roletypes(\Sigma)$ w.r.t. Σ, if the guard is

(1) a data variable x which was declared before and is of type `bool`,

(2) an attribute access **owner**.$attr$ with $\forall ct \in rtcomptypes(rt)\ .\ attr \in ctattrs(ct)$, i.e., all component types which are able to adopt a role of type rt provide the attribute $attr$, and $attr$ is of type `bool`,

(3) an attribute access **self**.$attr$ with $attr \in rtattrs(rt)$, i.e., the role type rt provides the attribute $attr$, and $attr$ is of type `bool`,

(4) a plays query $\mathbf{plays}(rt', C)$ which only refers to a role type $rt' \in roletypes(\Sigma)$ and C is either the special, predefined variable **owner** or **owner**.$assocnm$ referring to the name of a component association $assocnm{:}ct'$ which is defined for all component types $ct \in rtcomptypes(rt)$ which can adopt a role of type rt,

(5) a data expression e in which operators (e.g., summation, subtraction or relational operators like $<$, $==$, ...) are applied to expressions with matching data types and where type of the complete data expression is `bool`.

2.4.3 Role Behavior Declarations

Building on process expressions, we first introduce the notion of process declarations which the specification of role behavior declarations relies on.

Def. 2.11: Process Declaration

A process declaration *is of the form* ***process*** *procnm* $= P$ *such that procnm is the name of the declared process and* P *is a process expression.*

Relying on process expressions and process declarations, we can now define role behavior declarations which specify the dynamic behavior of instances of a role type.

Def. 2.12: Role Behavior Declaration

Let Σ *be an ensemble structure and* $rt \in roletypes(\Sigma)$ *be a role type in* Σ*. A* role behavior declaration for rt *has the form*

$$\textbf{\textit{roleBehavior}}\ rt = P$$

where P *is a process expression which is well-formed for* rt *with respect to* Σ.[a] *Additionally, a set* $procdecls(rt)$ *of (local) process declarations can be associated to the role behavior* rt *such that*

- *the expressions of all process declarations in* $procdecls(rt)$ *are well-formed for* rt *w.r.t* Σ *and*
- *all process invocations in* P *and in the expressions of all process declarations in* $procdecls(rt)$ *only invoke processes in* $procdecls(rt)$*.*

[a] Note that we use rt also as a process name for the role behavior of the role type rt.

Notation: The notation $rivars(rt)$ denotes all role instance variables from message reception, role instance creation, and role instance retrieval in the role behavior declaration (and the associated process declarations) for role type rt. Similarly, the notation $datavars(rt)$ denotes all data variables from message reception and operation call.

Example: The three role types *Requester*, *Router*, and *Provider* in the p2p example are equipped with a role behavior to fulfill their responsibilities. To download the file from the network, the role behavior of the *Requester* is given in Fig. 2.6. The requester first creates a router on the neighboring peer of its owner and sends a request for the address of the provider to the newly created router. The message *reqAddr* thereby includes the **self**-reference to the requester itself such that a reply can be sent back via this reference. Afterwards, the requester waits for the message *sndAddr* which transmits a reference of the provider. Via this reference, it sends a request for the file to the provider (again equipped with a **self**-reference to itself). Lastly, it waits to receive the content of the file with the message *sndFile*, stores the content in the attribute *content* of its owner, sets the attributes *hasFile* of its owner and of itself to **true**, prints the file by calling the operation *printFile* on its owning component, and finally quits its execution.

To retrieve the address of the provider by forwarding the request for it through the network, the role behavior of a router is given in Fig. 2.7. The set $procdecls(Router)$ is here implicitly given by the three process declaration $P_{provide}$, P_{fwd}, and P_{create}. Initially, a router is able to receive a request for an address of the provider of the requested file.

```
roleBehavior Requester = router←create(Router, owner.neighbor) .
                         router!reqAddr(self)() .
                         ?sndAddr(prov:Provider)() .

                         prov!reqFile(self)() .
                         ?sndFile()(content:int) .

                         owner.content = content .
                         owner.hasFile = true .
                         self.hasFile = true .
                         owner.printFile() .
                         quit
```

Figure 2.6: Role behavior of a *Requester* for the p2p example

```
roleBehavior Router = ?reqAddr(req:Requester)() .
                      if (owner.hasFile) {P_provide}
                      else {P_fwd}

P_provide = provider←create(Provider, owner) .
            req!sndAddr(provider)() . quit

P_fwd = if (plays(Router, owner.neighbor)) {quit}
        else {P_create}

P_create = router←create(Router, owner.neighbor) .
           router!reqAddr(req)() . Router
```

Figure 2.7: Role behavior of a *Router* for the p2p example

Depending on whether the router's owner has the file or not, it either provides the file to the requester in the process $P_{provide}$ or forwards the message to another router P_{fwd}. To provide the file in $P_{provide}$, the router creates a new role instance of type *Provider* on its owning component and sends the reference of the newly created provider back to the requester before it quits its execution. To forward the message in P_{fwd}, the router checks whether the neighboring component of its owner already plays the role *Router*. If so, the neighboring component does not have the file (since it already forwarded the message in its role as a router) and the router can stop to forward the message (represented by **quit**). That means for the whole ensemble that the file does not exist in the p2p network. If the neighboring component does not play the role *Router*, a new router is created on the owner's neighbor and the request is forwarded to this new router (cf. process P_{create}). Afterwards, it resumes its behavior from the beginning. Note that a router only terminates if its owning component has the file, i.e., the owning component serves as provider of the file, or the neighboring component of its owning component already plays the role of a router, i.e., the message traversed the whole chain of routers and none of the owning components had the file. The extension to always terminate the router is easily done by a second round-trip throughout the p2p network, but not presented here to keep the example simple.

To provide the file to the requester, the role behavior of the provider is given by Fig. 2.8. The provider waits for a request for the file. Via the received reference to the requester it sends back the content of the file which is stored in the attribute *content* of its owner.

roleBehavior *Provider* = ?*reqFile*(*req*:*Requester*)() .
req!*sndFile*()(**owner**.*content*) .
quit

Figure 2.8: Role behavior of a *Provider* for the p2p example

2.4.4 Ensemble Specifications

The full ensemble specification in HELENA consists of two parts: an ensemble structure describing the structural composition of the ensemble and a set of role behavior declarations describing the interaction behavior of the ensemble by introducing a dynamic behavior for each role type occurring in the ensemble structure.

Def. 2.13: Ensemble Specification

An ensemble specification over CT *is a pair* $EnsSpec = (\Sigma, behaviors)$ *such that* Σ *is an ensemble structure over* CT *and* $behaviors$ *is a set of role behavior declarations which contains exactly one role behavior declaration* ***roleBehavior*** $rt = P$ *for each role type* $rt \in \Sigma$.

Example: The ensemble specification for the p2p example consists of the ensemble structure $\Sigma_{transfer}$ in Fig. 2.4 and the behaviors for the three role types *Requester*, *Router*, and *Provider* in Fig. 2.6, Fig. 2.7, and Fig. 2.8 resp. For a complete overview, we refer the reader to Appendix C.1.

2.5 Related Work

HELENA as we presented it in this chapter is related to different fields of research: It builds on a component-based platform and therefore shares concepts with component-based software engineering (CBSE). However, it enhances standard component models by the dynamic formation of ensembles from a set of components, similarly to works in the field of ensemble-based systems. In contrast to these ensemble-based approaches, HELENA furthermore introduces the notion of roles to model participants of ensembles independently from the actual components adopting the roles in ensembles. Role models in general already have a long history. Our understanding of roles is highly influenced by existing work in this field, but HELENA is the first to apply roles to ensemble-based systems. However, in multi-agent systems roles are also already used to describe participants of interacting agent organizations similarly to participants of ensembles. Finally, we introduce a dedicated process algebra for the description of roles which shares ideas with other common process algebras. In the following, we will consider each of the fields separately.

2.5.1 Component-Based Models

As we already discussed in the introduction, component-based software engineering (CBSE) [Szy02, RRMP08] is concerned with the development of self-contained components, their composition and interaction. For each component, its interface to the outside world is described by a set of ports. Connectors allow and coordinate interaction between components whose ports fit to the required properties of the connector.

Component models like Wright [AG94, AG97], Darwin [MK96], ACME [GMW97], and PADL [BCD00, BCD02], component frameworks like Fractal [BCL+04, BABC+09], SOFA [BHP06], ArchJava [ACN02], and Java/A [BHH+06] or formal component approaches like team automata [tEKR03], interaction automata [BvVZ06], assemblies [HK11] or multiparty session types [CDPY15] already consider sets of interacting components. However, CBSE approaches are not sufficient for the description of ensemble-based systems in three main points: They do not introduce an explicit notion of ensembles such that they do not allow to focus design and analysis on the participants of an ensemble only. They lack an explicit notion of active roles which would help to structure the different functionalities needed to contribute to an ensemble and the different behaviors which a component offers. They do not handle reconfiguration and dynamic instantiation transparently from components participating in ensembles which HELENA allows due to the introduction of the two levels of components and roles.

Let's take a closer look how standard component-based techniques like Wright [AG94, AG97, ADG98] or Darwin [MK96] could be used to model an ensemble-based system like the p2p example which we used for illustration throughout this chapter. We only focus on the architectural description of an ensemble leaving the behavior aside for now. Fig. 2.9 shows a modular component model of the p2p example. Each contributing role

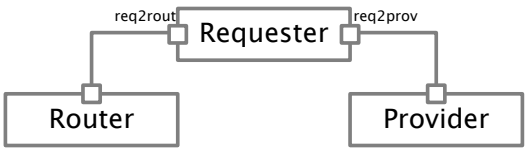


Figure 2.9: P2P example described by a modular component model

of the ensemble is specified by a separate component, i.e., a `Requester` component, a `Router` component, and a `Provider` component. Ports expose the interface for binary interactions. For example, the port `req2rout` handles the communication of the `Requester` with the `Router` while the port `req2prov` handles the communication of the `Requester` with the `Provider`. The components thus offer a port for each type of binary communication which they can be involved. Some components like `Requester` expose several ports while others like `Router` or `Provider` only expose one port. All ports of a component together describe its communication facilities. This modular component model of the p2p example has two disadvantages: Firstly, the model does not show that depending on the actual peer network, a chain of routers has to forward the request for the file through the network. If we wanted to specify that, we would have to use techniques like lazy or dynamic instantiation in Darwin [MK96]. It allows to describe structures evolving according to a fixed pattern. However, most component models do not support this kind of dynamism. Secondly, the p2p example is not sufficiently described by this modular component model. The underlying peer network does not consist of dedicated `Requester`, `Router`, or `Provider` components. It rather is composed of general purpose components which can serve in the role of a `Requester`, `Router`, or

`Provider`. A more suitable model would therefore combine all these sub-components to a larger composite `Peer` component.

Fig. 2.10 shows a component model where the `Peer` component is modeled as a one-fits-all component. It combines all functionalities and behaviors needed in all of the

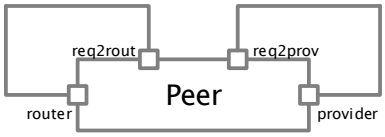


Figure 2.10: P2P example described by a one-fits-all component model

three roles discussed before. The four ports of the one-fits-all component expose the different communication facilities: The port `req2rout` exposes the interface of the `Peer` component acting in the role of a requester when communicating with another `Peer` component acting in the role of a router (and vice versa for the port `router`). Similarly, the port `req2prov` exposes the interface of the `Peer` component acting in the role of a requester when communicating with another `Peer` component acting in the role of a provider (and vice versa for the port `provider`). However, this one-fits-all component suffers from two disadvantages. Firstly, the functionalities and behaviors for all possible roles of the `Peer` component are mixed together into one large component without any structure depending on the different tasks the component has to fulfill in the different roles. Secondly, ports only expose a binary communication interface such that a role like the requester has to be represented by two different ports, one for each communication partner of the requester. Already crammed, this one-fits-all component model gets even more overloaded if we imagine a second type of ensemble built on top of the same `Peer` component. Even more goal-directed behaviors would be introduced to the large one-fits-all component and even more ports and connections to form the new ensemble would be needed. To overcome this overly large component model, we propose a separation between component and role in Helena. A role describes the functionalities needed in an ensemble and an adoption relation between role and component allows the component to provide its capabilities to the adopted role.

The notion of roles already appears in CBSE techniques in a different meaning: A connector describes the partners required for binary communication by roles [AG94, AG97] as shown in Fig. 2.11. Each role of a connector normally prescribes a certain

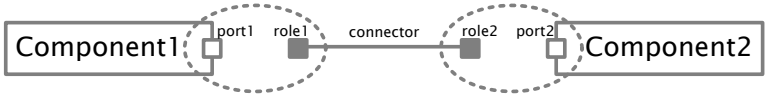


Figure 2.11: The role concept used in binary communication via connectors

interaction protocol. A component which is later on bound via its port to this role has to fulfill this interaction protocol (cf. compatibility of protocols [AG94]). The connector itself coordinates the interaction between the two communication partners by prescribing a certain sequence of interactions. In contrast, in Helena, roles are not restricted to one part of binary communication only, but rather which part a participant of an ensemble contributes to the overall ensemble. Furthermore, roles are active instances in Helena which define a goal-directed behavior relying on the capabilities of the underlying component. In CBSE, a role at a connector only constrains the behavior of a component which adopts the role at the connector.

Standard CBSE techniques focus on some of the aforementioned aspects of ensemble-based systems like sharing components between several collaboration groups [BCL+04, BABC+09], reconfiguration [ADG98, BHP06], and dynamic instantiation [MK96]. However, none of them combines these aspects with an explicit concept for ensembles and their participants which allows dynamic reconfiguration and instantiation as we do it in HELENA. For a more detailed comparison of component-based approaches we refer the reader to [BHP06] or [Jan10].

2.5.2 Ensemble-Based Models

Approaches from the field of ensemble-based systems take into account the notion of ensembles, but do not yet explicitly separate the functionalities needed for a particular goal-oriented ensemble from the components actually contributing the functionalities. We focus on four works from this field.

SCEL [DLPT14] provides a kernel language for abstract programming of autonomic systems. The entities of the systems are autonomic components which rely on knowledge repositories to store knowledge items and interact via explicit knowledge exchange on these repositories.

In SCEL, ensembles are understood as group communications. Such groups are defined by predicates over properties of components, e.g., a predicate describes all components in range of the acting component. With these predicates, an action to put or retrieve knowledge items can be directed to a set of knowledge repositories all satisfying the given predicate. Like that, all components which can access the involved repositories are included in this group communication. An ensemble is thus a very dynamic concept which is evaluated at runtime and might only exist for a single interaction. Opposed to ensemble structures in HELENA, these predicates do not specify the conceptual structure of an ensemble collaborating to achieve a certain global goal.

SCEL components are in principle modeled as one-fits-all components. However, it is possible to simulate the idea of roles by starting parallel processes on top of a component, even dynamically at runtime.

The behavior of a component is also described by a dedicated process algebra. On the level of process constructs, SCEL goes beyond HELENA by providing a controlled composition construct. This allows to compose parallel processes which is not allowed in HELENA since parallel behaviors should be expressed by different roles on top of a component. In contrast to HELENA, SCEL relies on knowledge exchange instead of message exchange. As already mentioned before, knowledge exchange allows group communications which are not yet part of HELENA. Furthermore, components can dynamically create other components. In HELENA, we do not create new component instances during the run of an ensemble because we assume them to be already given by an overall system management when an ensemble is started. However, component instances can dynamically join and leave an ensemble by dynamically creating new role instances and letting components adopt these new roles during ensemble execution.

An interesting extension of SCEL is PSCEL [MPT13]. It integrates FACPL policies [MMPT14] into SCEL specifications to define adaptation actions of ensemble-based systems. With these adaptation actions, an ensemble-based system is able to react to changes in the environment, can allow or forbid certain actions, and can switch to behaviors more suitable to the current situation. In HELENA, this can be expressed by components which adopt roles depending on the current state of the environment. A de-

velopment methodology for adaptive systems based on HELENA roles will be presented in Chap. 11.

DEECo [BGH+13] is a component-based framework which introduces an explicit specification artifact for ensembles. An ensemble is a dynamic group of components characterized by a membership predicate similarly to a former variant of SCEL. DEECo's runtime infrastructure dynamically determines the current members of an ensemble according to this membership predicate. Thus, in contrast to SCEL, ensembles are more explicit since DEECo provides an explicit specification artifact, but in contrast to HELENA, it still does not support any structural description of members and their interactions like in HELENA's ensemble structures.

On the level of components, DEECo offers a structural element to describe the different tasks which a component can perform. For each task, a separate process can be defined which manipulates the local knowledge appropriately. Depending on the given scheduling policy, this process is executed periodically or whenever a certain trigger condition is met. Although the behavior of a component is thus structured into separate processes (which can be started on demand), these processes are not directly associated to particular ensembles where the component contributes to. Furthermore, DEECo does not allow to instantiate new members of an ensemble by component instance creation. The set of components which can contribute to an ensemble is fixed during runtime, only the membership predicate controls when a component joins or leaves an ensemble. In HELENA, we achieve the clear assignment of goal-directed behavior (represented by processes on components) to ensembles by structuring ensembles in roles. These can dynamically be instantiated and adopted by components.

Instead of direct communication between the components of an ensemble, DEECo's runtime infrastructure manages implicit knowledge exchange between the participants of an ensemble which can then operate on their local knowledge. A computational model for DEECo is defined in terms of automata [ABG+13] which express knowledge exchange by buffered updates of the components' knowledge.

DCCL [BBvP13] is a similar approach to DEECo. It also defines an ensemble by a membership predicate which is dynamically evaluated during runtime of the ensemble. Components do also not communicate directly via message exchange, but implicitly via knowledge exchange among the participants of an ensemble. Components only operate on their local knowledge. While DEECo focuses on the implementation and execution of ensemble models, DCCL is a verification-oriented modeling language. It allows to verify the modeled ensemble-based system against properties over the local state of a component or the global state of the system. Compared to HELENA, it comes with the same restrictions as DEECo: no explicit structure for ensembles, no explicit structure for roles, no instance creation, and only implicit knowledge exchange.

BIP [BBB+12] and DyBIP [BJMS12] are component-based frameworks which focus on a clear separation between components and interactions while DyBIP additionally supports change of interaction models. The BIP framework can be viewed as an architecture description language with a rigorous formal semantics. Atomic components are finite-state automata which basically interact with other components via shared actions called ports (they should not be confused with ports in standard component models which expose an interface). The concept of shared actions is extended by connectors: They define which actions are synchronized between components even if not

sharing the same name. They define the type of interaction, e.g., whether the actions are executed synchronously or asynchronously. Finally, priorities at connectors specify which interaction is taken if several interactions are possible. DyBIP extends the BIP framework by dynamic connectors between components. These dynamic connectors are defined by interaction constraints on the transitions of the finite state-automaton of an atomic component. Dynamic connectors nicely support the dynamic architecture of ensemble-based systems where components dynamically form ensembles and therefore new connections to collaborate for some global goal. However, compared to HELENA, BIP and DyBIP do not introduce any notion of ensemble or role. Components have to be designed again as one-fits-all components; there does not exist any structuring element which allows to separate the different behaviors which a component offers. Furthermore, it is not possible to create new instances for a particular task on demand like HELENA allows by role instance creation.

2.5.3 Role-Based Models

HELENA is centered around the notion of roles. Bachmann and Daya [BD77] were the first to propose role-based modeling. They observed that an item in a data model often only represents a certain subset of properties of a real-world entity. According to them, this is "in contrast with integrated database theory which has taught that each record should represent all aspects of some one entity in the real world"[BD77]. They therefore suggested to define roles which allow to consider a real-world entity from a certain viewpoint or in a certain context. In this context, only some particular properties of the entity are relevant.

The role models which emerged starting from this vision all focus on different aspects such that no common understanding of roles was formed. Thus, Steimann [Ste00b] identified a set of features according to which the different approaches can be evaluated. Most prominently among these are whether a role has its own properties, state and behavior, whether it shares identity with the underlying real-world entity, and whether an entity can play several roles simultaneously. Later on, Kühn et al. [KLG+14] observed that most role models nowadays especially focus on the context in which the roles exist. Dedicated concepts are introduced in many approaches which particularly specify this context, called "compartments" in [KLG+14]. HELENA is an approach where roles are represented by volatile instances which can only exist bound to an owner, but which have their own properties, state and behavior. A HELENA component can adopt several roles simultaneously, but not of the same type in the same ensemble. Furthermore, HELENA particularly considers ensemble structures, i.e., compartments in which several roles collaborate for some global goals.

Apart from these features, Steimann [Ste00b] concluded that existing role models mainly fall into three categories which we extend by one category contributed by [KLG+14]: roles as named places of a relationship, roles and real-life entities as conglomerates, roles as adjunct instances, roles as participants of collaborations. In the following, we review each of the categories together with some representative approaches. For a more detailed review, we refer the reader to [Ste00b] and [KLG+14].

Roles as Named Places: Relational models mainly describe types and their relationships. However, as soon as there exist more than one relationship between two types or one type participates several times in one relationship, it is not enough to just name the related types. They have to be distinguished by the contribution which they

bring to the relationship. Such contribution is identified by a meaningful name which expresses the role of the type in the relationship.

This notion is for example used in UML [HKKR05]; other representatives of this category are the well-known ER model [Che76] or the ORM model [Hal09]. In a UML class diagram, the ends of an association between two classes are provided with a role name; in sequence diagrams, interaction partners are identified by a role name. In both cases, the role is just a name expressing the contribution which the entity offers to the relation or interaction. It does not support the characteristic that roles have their own properties and behavior as we consider it central for the notion of roles in HELENA.

Roles and Real-Life Entities as Conglomerates: The main characterization of this category is that roles are mapped onto the type hierarchy of the underlying real-world entities. Roles can be seen as subtypes of the types of real-world entities. They specialize a real-world entity by focusing only on particular properties. However, with roles as specialization, we experience problems if several different types of real-life entities can adopt the same role. The role then has to specialize both supertypes although a concrete instance of the role at runtime will never specialize both supertypes. Taking an example from [Ste00b], a person as well as an organization can adopt the role of a customer, i.e., the role customer has to specialize both supertypes, person and organization. However, at runtime, a customer will always be either a person or an organization and not both.

Therefore, roles can also be considered as supertypes of types of real-world entities. A role only provides a certain set of role-specific properties and operations while the whole real-life entity combines all these. With roles as generalizations, we are able to express that either subtype can adopt the role, but we also force each subtype to actually adopt the role. Taking the same example as before, a person as well as an organization can adopt the role of a customer if the role of a customer is the supertype of persons and organizations. However, consequentially each person and organization must automatically always be a customer and can never drop the role.

Most approaches in this category [SD96, BO98, JSHS96, GWGvS04] solve the problem whether a role is a subtype or a supertype of the type of a real-world entity by using dynamic specialization. They define roles as a dynamic type which comprises all objects which are currently engaged in a certain relationship.

A further example for this category which we want to consider more closely are UML composite structure diagrams for classes as presented in [HKKR05, Chap. 3.5.1]. This type of diagram allows to refine the classes and their relationships already structured in a UML class diagram from the viewpoint of a single context class. The underlying class diagram specifies the general associations and interaction relationships between classes. In the context of a certain class, only some of these classes and relationships might be used and some of them might even be restricted. The composite structure diagram for a context class therefore provides a view on the modeled system from the perspective of this context class. In the manner of speaking of UML, the composite structure diagram determines the roles in which classes of the complete system act if considered from the viewpoint of the context class. This category of roles thus requires that a role and its underlying real-world entity share identity and state, but the role serves as a view on particular properties of the real-world entity. Therefore, it is not allowed that roles have their own properties and behavior as we consider it essential in HELENA.

Roles as Adjunct Instances: Instead of mapping roles directly onto the type hierarchy of the underlying real-life entities, this category introduces roles as own types. Role types define role-specific properties and behavior, instances of those types have their own state. However, role instances cannot exist without a real-life entity adopting it. Therefore, a role instance must always be bound to its owner. The owner can adopt several roles in parallel and can dynamically pick up or drop them. Simultaneous role adoption naturally allows the underlying owner to simultaneously be in several role-specific states. By using roles as facades for the underlying real-life entities, interaction partners only communicate via their adopted roles such that their mutual view is restricted to the role-specific properties only.

Kristensen and Østerbye [KØ96] as well as Gottlob et al. [GSR96] are the most prominent authors to transfer this idea of roles to conceptual modeling and programming. Their understanding of roles builds the foundation for HELENA: Intrinsic properties describe the core information about a real-world entity. For example, for a person, these are properties like date of birth. Extrinsic properties describe the properties which are only relevant in a certain context. They are defined as a role which can temporarily be bound to a real-world entity. For example, a person can adopt the role of a teacher whose main property is the subject the person teaches. HELENA is influenced by the characteristics proposed in this role model: A role has its own state and its own operations to manipulate that state, a role is dependent on a real-world entity and can only exist bound to an entity currently adopting the role, a role is dynamically bound to an entity and is therefore of a temporary nature. Opposed to HELENA, the role model by Kristensen, Østerbye and Gottlob does neither consider active entities nor active roles. Furthermore, the context in which the roles exist is not considered which is in contrast to HELENA where we explicitly model ensemble structures consisting of roles. Kristensen, Østerbye and Gottlob mainly focus on the dynamism of role playing and interferences between concurrent roles.

The modeling language LODWICK by Steimann [Ste00b] is one important formal approach to modeling roles as adjunct instances sharing ideas with HELENA. A "model specification" in LODWICK consists of three parts: a signature, a static model and a dynamic model. The signature relates types and roles which the types can adopt. The static model determines all instances of types and roles as well as their adoption relationships which may potentially exist during the lifetime of the system. The dynamic model consists of sequences of sets of concrete (type and role) instances and their adoption relationships over time. HELENA follows the same idea of roles and their dynamic adoption by components (as we will see in Chap. 3 about HELENA's semantics). However, it goes one step further by allowing to dynamically create new role instances on demand while in LODWICK all instances are predefined. Furthermore, LODWICK is only designed as a rudimentary modeling language. The properties and capabilities of types and roles is not part of the modeling language while it is in HELENA. LODWICK does not contain collaboration specifications opposed to ensemble structures in HELENA. Most importantly, it does not support interactions in the dynamic models as we consider it central to describe ensembles and the goal-oriented behaviors of their participants.

Roles as Participants of Collaborations: According to Kühn et al. [KLG+14], more recent role-based approaches do not only model roles and their relationships to real-world entities, but also the context in which they exist. The "context represents a collaboration or container of a fixed, limited scope" [KLG+14]. In some approaches, the context is called team [Her03], institution [BSI07], role model [Ree96], or compart-

ment [KLG+14]. In HELENA, we call the context *ensemble*. Following Kühn et al., roles operate in a certain context or collaboration, but can also be shared between different collaborations. By introducing an explicit notion of such collaborations, they can be handled as first-class modeling concepts which may have their own identity, may play roles themselves, may be hierarchically composed etc. [KLG+14]. In HELENA, collaborations are represented by ensemble structures. The participants of an ensemble structure are specified by role types which can be shared between several ensemble structures. It is considered future work in HELENA to allow hierarchical composition of ensembles or to let several ensembles collaborate for a higher shared global goal.

UML [HKKR05, Chap. 3.5.2] already proposes composite structure diagrams for collaborations to model the structure of a system in the context of a collaboration. The composite structure diagram provides a structural view on the whole system which only shows the relationships between elements of the collaboration. Thereby, the elements do not need to be actual structural elements of the system, but can be aggregations or abstractions of them. Each element of the collaboration contributes under a certain role which describes the function which the element fulfills in the collaboration similar to role names in class diagrams or sequence diagrams. However, UML composite structure diagrams are very limited compared to other role-based approaches. They only describe the structure of a collaboration. Since their elements are not integral parts of the system, behavioral UML diagrams cannot directly refer to these elements to specify their behavior. Furthermore, collaborations cannot be instantiated such that they are mostly used to specify structural patterns.

Other approaches which include collaborations as first-class entities mostly reside in the area of programming languages. For example, ObjectTeams/Java [Her03] introduces aforementioned "teams" as a first-class programming construct, powerJava [BSI07] "institutions", and OOram [Ree96] "role models". Like in HELENA, they all define the structural model of a collaboration by participating roles. However, in ObjectTeams/Java and powerJava, roles are not active, but only react to operation calls. In the OOram method, roles are autonomic entities which start to exchange messages for collaboration upon external stimulus (like a file being requested from the outside). In contrast to HELENA, all role models representing collaborations are composed to a single composite role model. Thus, the OOram method only considers composite behaviors while HELENA explicitly considers the parallel execution of role behaviors as we will see in Chap. 3.

Apart from programming languages, role modeling is also incorporated as a central part in methodologies for analysis and design of (multi-)agent systems. For instance, the GAIA methodology [WJK00] and its extensions [CJSZ04] consider a multi-agent system as a computational organization consisting of various interacting roles; this is very similar to our interpretation of ensembles. Most specifications in these methodologies are, however, rather informal or at most semi-formal, like the UML-based notation Agent UML [BMO01]. Agent UML models collaborations by interaction protocols which combine sequence diagrams with state diagrams. Another approach has been pursued in the ROPE project [BGKM99], which proposes to use "cooperation processes" represented by Petri nets for the specification of collaborative behavior. A model-driven approach to the development of role-based open multi-agent software is presented in [XZP07]. It uses Object-Z notation and focuses merely on structural properties of role organizations and agent societies and not on interaction behavior. The structural concepts involve, however, specifications of role spaces as containers of role instances (that can be taken by agents), which resembles ensemble states in HELENA. All these methods are not

based on a formal semantics and do not provide verification techniques which we will present in the next chapters. In particular, they do not formalize concurrent executions which is built-in in HELENA's semantics as shown in Chap. 3.

2.5.4 Process Calculi

HELENA also shares ideas with process algebras and multiparty session types.

Process Algebras

In HELENA, the behaviors of roles are described by process expressions. Leading process algebras like CSP [Hoa78], CCS [Mil82], ACP [BK89] or the π-calculus [Mil99] already propose the main process constructs which we use similarly in the HELENA syntax: termination, action prefix, nondeterministic choice, the if-then-else construct and process invocation. Apparently, HELENA does not support a parallel composition operator which is contrast to standard process algebras. The main reason is that a role provides a behavior which is responsible for exactly one task in the collaboration of an ensemble. If we allowed parallel composition in role behaviors, we would be able to specify several processes which are all executed in parallel in one single role behavior. Thus, a role would be able to simultaneously work on several tasks. This would contradict the idea to extract each task to a separate role type and its behavior. As we will see in Chap. 3 about the semantics of HELENA, termination is also not formalized in a standard way. Since we introduce two layers in HELENA, components and roles which are adopted by components, termination does not only terminate the execution of the role behavior, but also advises the owning component of the role to quit playing the role. More details on that can be found in Chap. 3.

On the level of actions, HELENA provides the standard message passing actions for sending and receiving messages. These messages are directed towards one particular receiver only and are sent and received synchronously or asynchronously. Compared to leading process algebras, the following two types of actions in HELENA stand out: Firstly, HELENA allows to create and retrieve role instances. Therefore, new role instances can dynamically be created by another role instance during runtime of the ensemble and references to existing role instances can be retrieved on demand. Secondly, since HELENA introduces two layers, components and roles adopted by the underlying components, HELENA also introduces a special action to bridge these two layers. A role can call an operation on its owning component to exploit the capabilities of the underlying component. A deeper comparison of HELENA and its communicating roles is left to Sec. 3.5 since differences mainly become apparent on the level of semantics.

Multiparty Session Types

Finally, we want to highlight a process calculus which particularly focuses on groups of interacting components, multiparty asynchronous session types [CDPY15]. This calculus allows to model and reason about interactions between several components within a certain scope of a distributed system. This scope is called a multiparty session since several different parties communicate within a private session. Outsiders cannot interfere in this private session since communication relies on a private session channel. Multiparty session types are used to describe the interaction protocol between the participants of the multiparty session. The goal of the formalization of the interaction protocol is to guarantee communication safety, protocol fidelity and progress [CDPY15].

An interaction protocol is first designed at global level. A global type describes the communication from a neutral global viewpoint and gives an abstract overview about the whole interaction structure. From the global type, local types are derived by a strict projection operation. Each local type abstractly describes the interaction from the viewpoint of a single participant of the multiparty session. Finally, processes following a dedicated process calculus describe the concrete implementation of the behavior of a single participant. A strict type system takes care that each process respects its derived local type. The main advantage of this approach is that the global properties of communication safety, protocol fidelity and progress can be checked on the local types only. Due to the type system relating a local type and its concrete process implementation, their satisfaction is preserved by the concrete implementation.

Compared to HELENA, the first thing to note is that multiparty session types do not define the architecture of the underlying system. They only consider the abstract and concrete specification of behaviors without the underlying component types. HELENA goes beyond that by introducing component types which define the capabilities of the underlying component-based platform and by introducing role types which allow a component to dynamically join a certain ensemble contributing a certain behavior.

Leaving the point of architecture aside, multiparty session types focus on the global description of the interaction protocol of a group of components. Local types are only derived for analysis and implementation proposes. HELENA, however, supports to specify local role behaviors and the interaction is treated on ensemble level. The systematic derivation of local role behaviors from a global interaction specification is future work for HELENA.

When we look into type and process description of multiparty session types more closely, we observe that HELENA is more flexible. A multiparty session is always started by an initiation action which makes all participants known to each other. Compared to HELENA, multiparty session types employ just one single global channel where all messages are put and retrieved from one single message queue. The message itself contains sender and receiver to distribute the message to the correct participant. In HELENA, role instances directly communicate with each other and each role instance manages its own (asynchronous) message queue, as we will see in Chap. 3. This might not preserve message ordering on global level, but is more flexible and does not require a central message queue which is not efficient in dynamically evolving ensembles.

Another difference is the dynamic participation of components in the collaboration. In multiparty session types, new participants cannot be created on demand, but they can dynamically join a multiparty session by channel delegation. Dynamic multirole session types [DY11] furthermore allow participants to join a collaboration under a certain role. However, participants under certain roles are also not created on demand. The notion of roles is only used to dynamically communicate with all participants of a session which currently adopt the role, similarly to the concept of broadcasting. HELENA is more flexible here since it supports the dynamic creation of new role instances which can join the ensemble on demand, albeit leaving broadcast messaging still open for future work.

2.6 Publication History

The notion of an ensemble specification as presented in this chapter is based on [HK14] and [KMH14]. In [HK14], ensemble specifications still rely on role connectors for message exchange between roles and on labeled transition systems to describe role behaviors. To simplify ensemble specifications, role connectors have been removed from HELENA

in [KMH14]. Thus, messages can now be exchanged between two role types which are outgoing for one role type and incoming for the other. Additionally, [KMH14] proposes a process algebra for the description of role behaviors as a more compact notation than labeled transition systems.

Compared to these publications, ensemble specifications as presented in this chapter include a more sophisticated notion of data. Types are introduced for all attributes of component types and role types and parameters of messages. The process algebra introduced in [KMH14] is extended and improved, in particular to include data and to allow branching according to guards. Furthermore, we add component associations to component types, rely on operations provided by component types instead of internal operations or messages provided by role types, and all role types of an ensemble structure are equipped with a capacity of their input queues. Specific criteria define the well-formedness of ensemble structures and process expressions.

2.7 Present Achievements and Future Perspectives

Present Achievements: The main concept for describing ensemble-based systems is the notion of roles. Components team up in concurrently running ensembles by adopting particular roles. While the components are just pure data containers and computing resources, the adopted roles are the active entities exhibiting a goal-directed behavior and collaborating with each other in ensembles.

The whole system is structured into independently running ensembles. They are goal-oriented communication groups of roles which do not rely on any other ensembles or any components not participating in the very same ensemble.

The syntax for the description of HELENA ensembles is following a model-based specification technique. An ensemble specification consists of two parts: the structural relationships between participants of an ensemble are specified by an ensemble structure and its collaborating roles; the dynamic behavior is given as a set of role behaviors, one for each contributing role. The behaviors describe how an ensemble evolves via message exchange between the participating roles.

Future Perspectives: On the level of syntax, several extensions of the HELENA approach come to mind:

Broadcasting: The set of actions could be extended by broadcast messages. Those messages could either be used to broadcast information to a whole ensemble, to instances of a particular role type, or even to a set of role instances specified by a logical predicate.

Component Interfaces: It might be useful to introduce component interfaces which component types implement. With that extension, a role type does no longer have to name all component types whose instances are allowed to adopt the role, but rather just indicates the required component interface. The component interface exposes all attributes and operation types the implementing component types share. On the one hand, this facilitates the criteria for well-formedness of process expressions. On the other hand, this allows to extend the underlying component-based platform by new component types independently from an ensemble-based system since the ensemble specifications do not have to be adapted.

Open Ensembles: Lastly, the use of open ensembles would allow to interact with components or actors outside of an ensemble. Open ensembles could communicate with other ensembles or roles inside of other ensembles which allows information exchange between ensembles. An ensemble could rely on another ensemble to perform a task divided into several subtasks. Open ensembles could also communicate with actors outside the ensemble, e.g., humans. Thus, humans could employ an ensemble with a certain task, contribute to the completion of the task, and retrieve the results of work from the ensemble.

Chapter 3

Semantics
Understanding HELENA

The semantic domain of ensemble specifications are labeled transition systems describing the evolution of ensembles. Informally, a labeled transition systems consists of a set of states and a set of labeled transitions between states. In the context of ensemble specifications, the states are the states which an ensemble can be in. Structured operational semantics (SOS) rules define the allowed transitions between those states. We pursue an incremental approach, similarly to [HL93] and [Wei97], by splitting the semantics into two different layers. The first layer describes how a single role behavior evolves according to the constructs for process expressions of the last section. The second layer builds on the first one by defining the evolution of a whole ensemble from the concurrent evolution of its constituent role instances.

3.1 Notations

In the definition of the semantics of HELENA, we will make use of functions to describe the state of a HELENA ensemble. To facilitate the notation of the following subsections, we define some preliminary properties of functions.

We assume given a function $f : D \to V$ which maps elements d of its *domain* given by the set D to values v of its *range* given by the set V.

- The notation $f(d) = \perp$ expresses that the value for the item d is *undefined*.
- The function f is *partial* if there exists $d \in D$ such that $f(d) = \perp$; otherwise, f is *total*.
- The function f for which holds that $f(d) = \perp$ for all $d \in D$ is denoted by $\emptyset$.
- The maximal set $D' \subseteq D$, for which holds $f(d) \neq \perp$ for all $d \in D'$, is called the *definition domain of* f and is denoted by $dom(f)$.
- If the value for an element $d \in D$ is either newly defined or redefined to the value $v \in V$ for a function f, we denote the redefined function by $f[d \mapsto v]$.
- The function f is *non-extensible* if the definition domain is not allowed to be extended by any new element; otherwise, it is *extensible*.

For the special case of a function $f : \mathbb{N}^+ \mapsto V$ which maps positive natural numbers $i \in \mathbb{N}^+$ to values $v \in V$, we introduce some additional notations.

- The function f is *finite* if there exists $n \in \mathbb{N}^+$ such that $f(i) \neq \perp$ for all $0 < i \leq n$ and $f(i) = \perp$ for all $i > n$.

- The index $n \in \mathbb{N}^+$ for which holds $f(i) \neq \perp$ for all $0 < i \leq n$ and $f(i) = \perp$ for all $i > n$ describes the *size of a finite function* f and is denoted by $size(f)$.
- The *first free index* whose value is undefined in a finite function f is $size(f) + 1$ and is denoted by $next(f)$.
- The definition domain of a finite function f can also be extended by defining a value v for the first free index $next(f)$. The extended function is denoted by $f[next(f) \mapsto v]$.

Furthermore, the semantic domain of a HELENA ensemble specification are labeled transition systems. They consist of a set of states and labeled transitions between those states. A traversal of the labeled transition systems starts in some initial state and follows a path of transitions which are labeled with actions.

Def. 3.1: Labeled Transition System

A labeled transition system (LTS) *T is a tuple $(S_T, I_T, A_T, \rightarrow_T)$ such that*

- *S_T is a set of states,*
- *$I_T \subseteq S_T$ is a set of initial states,*
- *A_T is a set of actions such that the silent action $\tau \notin A_T$, and*
- *$\rightarrow_T \subseteq S_T \times (A_T \cup \tau) \times S_T$ is a labeled transition relation.*

3.2 Ensemble States

For the semantics of a HELENA ensemble specification, we describe the states which an ensemble can be in and how the ensemble evolves from one ensemble state to another. In this section, we consider ensemble states and their formal description. Intuitively, an ensemble state captures the local states of all components composing the underlying component-based platform and the local states of all roles currently participating in the ensemble. Components are considered as persistent data storage which can be accessed from their owning roles while roles only store data which is needed for the execution of their goal-directed behavior. In the following, we first explain the local states of components and roles before we merge them together to describe the state of a complete ensemble.

Assumption: For simplification, we assume that role instance parameters as well as data parameters of messages are unary during the remainder of this chapter, i.e., we do not consider lists of parameters, but only a single role instance parameter and a single data parameter. The extension to lists of parameters is straightforward.

3.2.1 Component Instance State

The local state of a component instance represents all information locally stored on the component: the (unmodifiable) type of the component instance, the values of all its attributes and the references to all its associated components and to itself.

Def. 3.2: Component Instance State

Let V be the domain of data values. The local state of a component instance *is a tuple (ct, at^c, as) which stores the following information:*

- *the unmodifiable component type* $ct = (ctnm, ctattrs, ctassocs, ctops)$ *of the component instance,*
- *a total attribute value function* $at^c : ctattrs \to V$ *mapping attributes of the component type* ct *to values in* V*,*
- *a (possibly partial) association value function* $as : ctassocs \cup \{\boldsymbol{owner}\} \to \mathbb{N}^+$ *mapping component associations of the component type* ct *and the special, predefined variable* ***owner*** *to component instances identified by natural numbers.*

The set $\mathcal{L}_{comps}$ *denotes all local states of component instances.*

The tuple (ct, at^c, as) contains three parts: the (non-modifiable) type of the component instance and two value functions at^c and as.

The type ct of the component instance is unmodifiable and therefore fixed throughout the lifetime of the component instance.

The function at^c maps attributes to data values. Therefore, its domain encompasses the set *ctattrs* of all attributes of the component type ct and its range all data values in V. Most importantly, we require the function at^c to be total, i.e., for each attribute $attr \in ctattrs$, there must exist a value $v \in V$ such that $at^c(attr) = v$. Intuitively, that means that each attribute of a component instance must always have a defined value.

The function as maps component associations and the special, predefined variable **owner** to component instances. Therefore, its domain encompasses the set *ctassocs* of all component associations of the component type ct and the variable **owner**. Component instances are identified by natural numbers such that the image of the function as is the natural numbers $\mathbb{N}^+$. We do not require the function as to be total, i.e., a component association can be unspecified. For all other component associations and the special, predefined variable **owner** we require that they must link to an existing component instance identified by a natural number. In the formal definition of the local state of a component instance, we cannot guarantee that these natural numbers always represent an existing component instance, but we formalize this restriction as a well-definedness condition for global ensemble states in Def. 3.5.

3.2.2 Role Instance State

The local state of a role instance represents the connection to its owning component, all information locally stored on the role, and all information related to the progress of execution of its role behavior. Thus, the local state of a role instance is composed of the (unmodifiable) type of the role instance, the reference to its owning component, the values of all its attributes, the values of all declared role instance variables including the special, predefined variable **self** and data variables used in its role behavior (resulting from role creation and retrieval, message reception and operation call), all messages which were sent to the role, but not yet received (i.e., the current content of its message input queue), the current progress of executing its role behavior.

Def. 3.3: Role Instance State

Let V be the domain of data values. The local state of a role instance *is a tuple $(rt, ci, at^r, v, w, q, P)$ which stores the following information:*

- *the unmodifiable role type $rt = (rtnm, rtcomptypes, rtattrs, rtmsgs_{out}, rtmsgs_{in})$ of the instance,*
- *the owning component instance $ci \in \mathbb{N}^+$ of the role instance or $ci = \perp$ if the role instance terminated its role behavior,*
- *a total attribute value function $at^r : rtattrs \to V$ mapping attributes of the role type rt to values in V,*
- *a (possibly partial) extensible local environment function $v : rivars(rt) \cup \{\textbf{\textit{self}}\} \to \mathbb{N}^+$ mapping local role instance variables to values, i.e., role instances identified by natural numbers,*
- *a (possibly partial) extensible local environment function $w : datavars(rt) \to V$ mapping local data variables to values in V,*
- *the current content q of the input queue of the instance (the empty queue is denoted by ϵ, the length of q is denoted by $|q|$), and*
- *a process expression P representing the current control state of the instance or $\perp$ representing termination.*

The set $\mathcal{L}_{roles}$ denotes all local states of role instances.

Let us explain the individual parts of the local state $(rt, ci, at^r, v, w, q, P)$ of a role instance in more detail:

The type rt of the role instance is unmodifiable and therefore fixed throughout the lifetime of the role instance.

The component instance ci adopts the role instance. It is identified by a natural number (similarly to associations in the local state of components). Well-definedness conditions of ensemble states in Def. 3.5 formalize that this number must always represent an existing component instance. However, if the role instance terminated execution of its role behavior, the adopting component can also quit adopting the role meaning the role does no longer have an owner. This is represented by the value $\perp$ for the owning component instance. Again, we formalize in Def. 3.5 about well-definedness of ensemble states that $\perp$ is only allowed as a value for the owning component instance if the role instance terminated execution of its role behavior.

The function at$^{\mathbf{r}}$ maps attributes to data values. Therefore, its domain encompasses the set *rtattrs* of all attributes of the role type rt and its range all data values in V. The function at^r is total, i.e., the values of all attributes have to be specified. Like in Java, we therefore assume that all attributes are initialized to default values at the beginning.

The function v maps role instance variables to role instances. Therefore, its domain encompasses the set $rivars(rt)$ of all role instance variables of the role behavior declaration of the role type rt and the special, predefined variable **self**. The variables originate from role creation, role retrieval, and message reception (see

notation associated to Def. 2.12 on page 30). Similarly to component instances, role instances are identified by natural numbers such that the image of the function v is $\mathbb{N}^+$. The function v can be partial and is extensible, i.e., the value of some role instance variable might be unspecified at first, but can be changed to a specific value during the lifetime of the role instance. However, in the formal definition of the local state of the role instance, we cannot guarantee that the natural numbers always represent an existing role instance. Therefore, we formalize this restriction as a well-definedness condition for global ensemble states in Def. 3.5.

The function w maps data variables to data value. Therefore, its domain encompasses the set $\mathit{datavars}(rt)$ of all data variables of the role behavior declaration of the role type rt and the image all data values in V. The data variables originate from message reception and operation call (see notation associated to Def. 2.12 on page 30). Like the previous two functions at^r and v, the function w can be partial and is extensible, i.e., the value of some data variable might be unspecified at first, but can be changed to a specific value during the lifetime of the role instance.

The content of q represents the input queue of the role instance. The queue lists all messages, which were sent to the role but not yet received, in order of reception. If the role instance communicates synchronously, the input queue is always empty since messages have to be received synchronously with sending the message from another role. Thus, the input queue only contains elements if the role instance communicates asynchronously with other roles. The number of elements in the input queue (and therefore asynchronous or synchronous communication) is restricted by the maximal capacity of the input queue which is given in an ensemble structure. We formalize this restriction as a well-definedness condition for global ensemble states in Def. 3.5.

The process expression P represents the current progress of execution of the corresponding behavior of the role instance. If the role instance terminated the execution of its role behavior, the remaining behavior is represented by $\perp$. The process construct $\perp$ is a semantic extension of the syntax of process expressions to describe that the role finished its role behavior and the owning component quit playing the role. In contrast, the process construct **quit** expresses that the role instance reached the end of its role behavior, but still the owning component has to quit playing the role. In the formal definition of the local state of a role instance, we cannot guarantee the role instance is no longer adopted by a component (denoted by $\perp$ for the owning component) if the remaining process expression is $\perp$. We formalize this restriction as a well-definedness condition for global ensemble states in Def. 3.5.

3.2.3 Ensemble State

The state of an ensemble is characterized by the component instances which provide computing and storage resources and by the role instances which execute their goal-directed behavior and therefore participate in the ensemble. The ensemble state manages two functions, one mapping component instance identifiers to local states of component instances and another one mapping role instance identifiers to local states of role instances.

Def. 3.4: Ensemble State

The global state σ of an ensemble *is a pair* $(comps, roles)$ *such that*

- $comps : \mathbb{N}^+ \rightarrow \mathcal{L}_{comps}$ *is a non-extensible finite function mapping each component instance identifier to a local state,*
- $roles : \mathbb{N}^+ \rightarrow \mathcal{L}_{roles}$ *is an extensible finite function mapping each role instance identifier to a local state.*

Component instances and role instances are both identified by identifiers taken from the natural numbers. These identifiers are unique if considering component instances and role instances separately, but might overlap if considering both at the same time. Therefore, the ensemble state manages two separate functions: *comps* maps a component instance identified by a natural number to a local component instance state and similarly, *roles* maps a role instance identified by a natural number to a local role instance state. In the following, we focus on the specific features of both functions:

The function *comps* is non-extensible, i.e., it has a fixed definition domain which cannot be extended throughout the lifetime of the ensemble. Intuitively, that means that the component-based platform building the foundation of the ensemble relies on a fixed set of components. Components cannot be added or removed during the execution of the ensemble. The current version of HELENA is focused on roles dynamically being created and adopted, apart from focused on components dynamically joining and leaving the underlying component-based platform.

The function *comps* is finite, i.e., the function *comps* maps local states to component instance identifiers beginning from the identifier 1 continuously up until a maximal identifier n. That means that we identify the fixed set of component instances by a set of monotonously increasing natural number identifiers.

The range of component instance identifiers will never change throughout the lifetime of the ensemble since the function *comps* is non-extensible (and therefore the number of component instances is fixed) and finite. However, we strongly emphasize that the local state of a component instance can change nonetheless, e.g., by attribute setters.

The function *roles* is extensible, i.e., the definition domain can be extended throughout the lifetime of the ensemble. That means that new role instances can be created during the execution of the ensemble and their local states are managed in the global ensemble state as well. Though, role instances are never deleted from a global ensemble state, even when they have been terminated. A role instance remains in the global ensemble state after termination, but it quits its connection to its owning component by setting the value $\perp$ as its owning component.

The function *roles* is finite. That means similarly to the previous function *comps* that role instance identifiers are monotonously increasing beginning from the identifier 1 up until a maximal identifier m. Whenever a new role instance is created, the range of identifiers is increased by 1 and the new role instance is identified by the new maximal identifier.

The definition domain of the function *roles* can be monotonously extended by new role instances since the function *roles* is extensible and finite. However,

we want to highlight that we can also change the local state of already existing role instances, e.g., by any action in the executed role behavior.

As we already hinted in the description of local states of components and roles, an ensemble state has to satisfy some conditions to be well-defined, e.g., identifiers must point to currently existing components and roles resp. The following definition lists all those restrictions for an ensemble state in the context of a given ensemble specification.

Def. 3.5: Well-Definedness of an Ensemble State

A global ensemble state $\sigma = (comps, roles)$ is well-defined *w.r.t. an ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *over* CT *with* $\Sigma = (nm, roletypes, roleconstraints)$ *if*

(1) for all $i \in dom(comps)$ *and* $comps(i) = (ct, at^c, as)$*:*

- *(a)* $ct \in CT$,
- *(b)* $as(\textbf{\textit{owner}}) = i$,
- *(c) for any* $C \in dom(as) : as(C) \in dom(comps)$,

(2) for all $i \in dom(roles)$ *and* $roles(i) = (rt, ci, at^r, v, w, q, P)$*:*

- *(a)* $rt \in roletypes$,
- *(b) if* $P \neq \bot$*:* $ci \in dom(comps)$,
  if $P = \bot$*:* $ci = \bot$,
- *(c)* $v(\textbf{\textit{self}}) = i$
- *(d) for any* $X \in dom(v) : v(X) \in dom(roles)$,
- *(e) for* $roleconstraints(rt) = (min, max, cap)$*:* $|q| \leq cap$,
- *(f) for* $q = msgnm_1(k_1)(e_1) \cdot \ldots \cdot msgnm_m(k_m)(e_m)$:
  $k_1, \ldots, k_m \in dom(roles)$,
- *(g) if* $P \neq \bot$, P *is well-formed for* rt *w.r.t.* Σ *with the exception of all (local) variables* X *occurring in* $dom(v)$ *and all (local) variable* x *occurring in* $dom(w)$,

(3) for all $rt \in roletypes$ *and* $roleconstraints(rt) = (min, max, cap)$*:*
$min \leq \left|\{i \mid roles(i) = (rt, ci, at^r, v, w, q, P) \text{ and } ci \neq \bot\}\right| \leq max$.

Let us explain the specific conditions for well-definedness of ensemble states:

We first consider the restrictions on the local states of component instances stored in the function *comps*: Item (1a) expresses that every existing component instance must be of a component type ct which is part of the component-based platform CT building the foundation for the ensemble specification *EnsSpec*. Furthermore, we require that the association value function as of each component instance maps the variable **owner** to its own identifier in item (1b). This condition is necessary to be able to use the variable **owner** as a predefined constant in the specification of role behaviors. Lastly, since the function as is total, every association is mapped to a natural number identifier representing a component instance. However, item (1c) requires that every such identifier actually represents a currently existing component instance.

Similar conditions restrict the local state of role instances stored in the function *roles*: Item (2a) expresses that every existing role instance must be of a role type rt which is part of the ensemble structure of the underlying ensemble specification *EnsSpec*. As item (2b) states, the identifier of the owning component instance of a role instance must refer to an actually existing component instance if the role is not yet terminated. However, if the role is terminated (expressed by the process construct $\perp$ as the remaining process expression), the owning component instance must be unspecified (expressed by the value $\perp$ for the owning component instance). Similarly to the predefined constant **owner** for component instances, the predefined constant **self** must be mapped to the role's own identifier in the local environment function v, as stated in item (2c). Furthermore, item (2d) requires that every role instance variable in the definition domain of the local environment function v represents an actually existing role instance (which might already be terminated). For the input queue q, the number of possible entries is restricted to the maximal capacity given for that role type in the ensemble specification (item (2e)) and we require all role instance parameters of messages in the input queue to refer to actually existing role instances (item (2f))[1]. Finally, the process expression P describing the remaining behavior to be executed for this role must be well-formed (item (2g)) except that all variables occurring in the local environment functions v and w must not be declared before.

Apart from these individual restrictions for each role instance, a well-defined ensemble state must contain at least as many role instances per role type as the minimal multiplicity for the corresponding role type in the ensemble structure states and at most as many as the maximal multiplicity in the ensemble structure states (item (3)).

The semantics of a HELENA ensemble specification evolves ensemble states beginning from an admissible initial ensemble state. Such an admissible initial ensemble state must capture (1) at least one component instance such that there can be (2) at least one role instance being adopted by the former component instance. (3) All role instances existing in the initial state must be initial, in the sense that they must be at the beginning of their corresponding role behavior without having executed any actions so far. That means that the role instance is adopted by a component instance, but all values for attributes, role instance parameters and data parameters are unspecified (except for the predefined constant **self**) and the input queue is empty.

Def. 3.6: Admissible Initial Ensemble State

A well-defined ensemble state $\sigma = (comps, roles)$ *is an* admissible initial state *for the ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *over* CT *with* $\Sigma = (nm, roletypes, roleconstraints)$ *if*

(1) there exists $i \in dom(comps)$,

(2) there exists $i \in dom(roles)$,

(3) for all $i \in dom(roles)$: $roles(i) = (rt, ci, \emptyset, \emptyset[\textbf{\textit{self}} \mapsto i], \emptyset, \epsilon, P)$ *such that behaviors contains the declaration* ***roleBehavior*** $rt = P$, *i.e.,* P *is the process expression in the declaration of the role behavior for* rt,

[1] Note that we assumed that parameter lists are unary at the beginning of this section.

If we consider that the execution of an ensemble must start in an admissible initial ensemble state, well-definedness of ensemble states is not a real restriction. Any admissible initial state is well-defined per definition. Furthermore, the structural operational semantics rules to evolve an ensemble which are defined in the next section preserve well-definedness. This follows from the syntactic restrictions for well-formed process expressions, and therefore role behavior declarations. The most important restrictions result from send actions. A send action in a process expression in HELENA is only well-formed if (amongst others) the variables X and Y have been declared before (or refer to the predefined variable **self**). Declaration however is done via receive, create, or get actions such that each send action must be preceded by appropriate receive, create or get actions if a process expression is well-formed. That matches the requirements for well-definedness of the ensemble states described in Def. 3.5.

3.3 Structured Operational Semantic Rules

To evolve an ensemble from one state to another, we determine the allowed transitions by structured operational semantic (SOS) rules. Thereby, we split the rules into two different layers. The first layer describes the evolution of a single role behavior without taking into account interaction with other roles and components. The second layer describes the evolution of a complete ensemble by considering the communication between roles and between roles and components. It builds on the first layer to concurrently evolve the constituent role instances of an ensemble. In the following, we present both layers of rules.

Assumption: For simplification, we assume as in the previous chapter that role instance parameters as well as data parameters of messages are unary, i.e., we do not consider lists of parameters, but only a single role instance parameter and a single data parameter.

3.3.1 Evolution of Roles

On the first level, we formalize the progress of a single role and its role behavior. The progress only captures how a process expression can evolve according to its structure without taking into account any interaction with other roles and components. The SOS rules in Fig. 3.1 define this progress inductively over the structure of well-formed process expressions (cf. Def. 2.9 on page 25). The symbol $\overset{a}{\hookrightarrow}_{i,\sigma}$ describes a transition on the role level when executing an action a for the role instance with identifier i in the global ensemble state σ. The role instance identifier i and the global ensemble state σ are necessary to be able to evaluate the guards of if-then-else constructs.

Let us consider each process construct individually: Termination with the process construct **quit** cannot evolve at role level. Intuitively, termination requires the owning component instance to quit playing the role. However, this means that on the role level no action is possible and the role simply terminates.

Action prefix $a.P$ can always evolve on the role level by executing the action a. Restrictions on the execution of an action only result from the communication between roles and components, e.g., for sending a message the input queue of the receiving role must be able to store one more element. These restrictions are taken into account when we consider the second layer, i.e. the evolution of the whole ensemble and therefore all interactions and dependencies between roles and components.

(action prefix) $a.P \overset{a}{\hookrightarrow}_{i,\sigma} P$

(nondet. choice-1) $$\frac{P_1 \overset{a}{\hookrightarrow}_{i,\sigma} P_1'}{P_1 + P_2 \overset{a}{\hookrightarrow}_{i,\sigma} P_1'}$$

(nondet. choice-2) $$\frac{P_2 \overset{a}{\hookrightarrow}_{i,\sigma} P_2'}{P_1 + P_2 \overset{a}{\hookrightarrow}_{i,\sigma} P_2'}$$

(if-then-else-1) $$\frac{P_1 \overset{a}{\hookrightarrow}_{i,\sigma} P_1'}{\textbf{if } (guard) \ \{P_1\} \textbf{ else } \{P_2\} \overset{a}{\hookrightarrow}_{i,\sigma} P_1'} \quad \text{if } [\![guard]\!]_{i,\sigma}$$

(if-then-else-2) $$\frac{P_2 \overset{a}{\hookrightarrow}_{i,\sigma} P_2'}{\textbf{if } (guard) \ \{P_1\} \textbf{ else } \{P_2\} \overset{a}{\hookrightarrow}_{i,\sigma} P_2'} \quad \text{if not } [\![guard]\!]_{i,\sigma}$$

(process invocation) $$\frac{P \overset{a}{\hookrightarrow}_{i,\sigma} P'}{N \overset{a}{\hookrightarrow}_{i,\sigma} P'}$$ if **roleBehavior** $N = P$ or **process** $N = P$

Figure 3.1: SOS rules for the evolution of roles

Nondeterministic choice can evolve any branch for which the corresponding process expression can evolve. If both process expressions can evolve, one of the branches is nondeterministically selected for execution. Consequently, nondeterministic choice cannot evolve if none of the process expressions of the branches can evolve. On the first level of a single role instance, only **quit** cannot evolve, all other process constructs and especially all actions can always evolve. On the second level of roles collaborating in an ensemble, we will see that some actions cannot evolve, e.g., if a role wants to send a message to another role which currently cannot receive it. This means, nondeterministic choice is realized in HELENA as an external choice between its two branches. It does not internally decide for one of the branches, but decides based on the executability of the branches which one to execute. Nondeterministic choice is therefore externally triggered since, e.g., we cannot internally decide to receive a message, but have to wait for the external action of sending a message from the outside to be able to receive it.

The if-then-else construct can evolve its then-branch if the guard evaluates to **true** and the corresponding process expression of the then-branch can evolve. It evolves its else-branch if the guard evaluates to **false** and the corresponding process expression of the else-branch can evolve. If the process expression of the branch chosen based on the evaluation of the guard is not executable, the whole if-then-else process construct cannot evolve. Note again that on the first level of a single role instance, only **quit** cannot evolve, all other process constructs and especially all actions can always evolve. On the second level of roles collaborating in an ensemble, branches might become not executable. To evaluate the guards of guarded choice, we assume that guards are well-formed as presented in Sec. 2.4.2. Fig. 3.2 inductively defines the evaluation $[\![guard]\!]_{i,\sigma}$ of a guard *guard* based on its syntactic structure according to Def. 2.9 on page 25. The evaluation depends on the role instance i in whose behavior the guard occurs. Furthermore, it depends on the current state σ of the ensemble, e.g., when evaluating attributes of the role instance or its owner.

$[\![\textbf{true}]\!]_{i,\sigma}$	$=$ **true**	
$[\![\textbf{false}]\!]_{i,\sigma}$	$=$ **false**	
$[\![x]\!]_{i,\sigma}$	$= w(x)$	iff $\sigma = (comps, roles)$ and $roles(i) = (rt, ci, at^r, v, w, q, P)$
$[\![\textbf{owner}.attr]\!]_{i,\sigma}$	$= at^c(attr)$	iff $\sigma = (comps, roles)$, $roles(i) = (rt, ci, at^r, v, w, q, P)$, and $comps(ci) = (ct, at^c, as)$
$[\![\textbf{self}.attr]\!]_{i,\sigma}$	$= at^r(attr)$	iff $\sigma = (comps, roles)$ and $roles(i) = (rt, ci, at^r, v, w, q, P)$
$[\![\textbf{plays}(rt, C)]\!]_{i,\sigma}$	$=$ **true**	iff $\sigma = (comps, roles)$, $roles(i) = (rt_i, ci_i, at^r_i, v_i, w_i, q_i, P_i)$, $comps(ci_i) = (ct_i, at^c_i, as_i)$, and there exists j such that $roles(j) = (rt, as_i(C), at^r_j, v_j, w_j, q_j, P_j)$ and $as_i(C) \neq \perp$
	$=$ **false**	otherwise
$[\![e]\!]_{i,\sigma}$	$= \ldots$	(usual evaluation of data expression)
$[\![!guard]\!]_{i,\sigma}$	$=$ **true**	iff not $[\![guard]\!]_{i,\sigma}$
	$=$ **false**	otherwise
$[\![guard_1 \;\&\&\; guard_2]\!]_{i,\sigma}$	$=$ **true**	iff $[\![guard_1]\!]_{i,\sigma}$ and $[\![guard_2]\!]_{i,\sigma}$
	$=$ **false**	otherwise

...

Figure 3.2: Evaluation of guards

- The boolean primitives **true** and **false** directly evaluate to their corresponding semantic value.
- A data variable x is evaluated based on the current state σ of the ensemble. The value $w(x)$ of the data variable x is retrieved from the local environment function w of data variables stored for the role instance i in the ensemble state σ.
- Similarly, the guard **owner**.*attr*, i.e., the attribute *attr* of the owning component of the current role instance, is evaluated. We access the attribute value function at^c of the owning component ci of the role instance i and retrieve the value $at^c(attr)$ for the attribute *attr*. Thereby, it is important that the guard **owner**.*attr* is well-formed, in the sense that the attribute *attr* of the owning component must be of type `bool` to serve as a guard.
- For the guard **self**.*attr*, we access the attribute value function at^r of the role instance i and retrieve the value $at^r(attr)$ for the attribute *attr*. Again, the guard **self**.*attr* must be well-formed to retrieve a boolean value.
- Intuitively, a plays query **plays**(rt, C) evaluates to **true** if the component instance identified by C adopts a role of type rt. However, C is not a component instance identifier, but is either the special, predefined variable **owner** or **owner**.*assocnm* referring to the name of a component association of the owning component instance. Thus, we have to retrieve the corresponding component instance identifier $as_i(C)$ from the association value function as_i of the owning component instance ci_i of the current role instance i. If this identifier $as_i(C)$ is not $\perp$, we then deter-

mine whether there exists a role instance j in the current ensemble state σ which is of type rt and is adopted by the component with the identifier $as_i(C)$.

- Data expressions e are not further specified in the syntax of HELENA. We assume them to be built from constant values, data variables, and attribute getters combined with the usual arithmetic and relational operators. These expressions are evaluated as usual, e.g., **self**.*count* $< 1 + 2$ evaluates to **true** if the current value of the attribute *count* of the role instance i is smaller than 3.
- Finally, propositional operators like ! and && can be used in guards. In Fig. 3.2, the evaluation of negation with ! and conjunction with && is specified as usual. The evaluation of further propositional operators is omitted here, but is straightforwardly defined.

To come back to the evolution of roles, we finally consider the evolution of process invocation. In the context of a role behavior, only the role behavior itself or a process from the set of associated local process declarations can be invoked. In either case, we assume that the role behavior N or the process N is defined by the process expression P. Then, if the process expression P can evolve by an action a to the process expression P', the process invocation N can evolve by the same action a to the process expression P'. The semantics of HELENA does not prescribe a separate step for process invocation, but immediately executes the first action of the invoked process.

3.3.2 Evolution of Ensembles

On the second level, we formalize the concurrent evolution of roles based on the underlying component-based platform. This level takes into account the interaction between roles and components and therefore evolves the state of a complete ensemble. The SOS rules in Fig. 3.3, Fig. 3.4 and Fig. 3.5 define the evolution of an ensemble state in the context of an ensemble specification under the assumption of asynchronous communication. The symbol $\xrightarrow{i:a}_{\text{HEL}}$ describes a transition from one ensemble state to another when executing an action a for the role instance with identifier i on the ensemble level. For each rule, the transition between the two ensemble states is inferred from a transition of a process expression on the role level, denoted by $\overset{a}{\hookrightarrow}_{i,\sigma}$.

If a role terminates the execution of its role behavior by the process construct **quit** (rule quit), the role cannot execute any action on the role level, but on the ensemble level the owning component has to quit playing the role. Thus, the SOS rule for **quit** takes care that the owning component of a terminating role is set to the value $\perp$ (expressing that the role is no longer owned or played by any component) and likewise that the remaining behavior is set to $\perp$ (expressing that the execution of the role behavior is completely finished). However, the role is only allowed to terminate its execution if the number of adopted (i.e., not terminated) instances of its role type is greater than the minimal number of instances for its role type required in the ensemble specification (item (2)).

The SOS rule of a create action formalizes three conditions when a role instance i can issue the creation of a role instance of type rt_j on a component instance identified by C (rule create): Firstly, the component identified by C[2] must exist and be defined

[2]Similarly to plays queries, C is not a component instance identifier, but C is either the special, predefined variable **owner** or **owner**.*assocnm* referring to the name of a component association of the owning component instance. Thus, we have to retrieve the corresponding component instance identifier $as_i(C)$ from the association value function as_i of the owning component instance ci_i of the role instance

$$\text{(quit)} \quad \frac{}{(comps, roles) \xrightarrow{i:\mathbf{quit}}_{\text{HEL}} (comps, roles[i \mapsto (rt, \bot, at^r, v, w, q, \bot)])}$$

$$\text{if} \begin{cases} (1)\ i \in dom(roles), roles(i) = (rt, ci, at^r, v, w, q, \mathbf{quit}) \\ (2)\ roleconstraints(rt) = (min, max, cap), \\ \quad \left|\{k \mid roles(k) = (rt, ci_k, at^r_k, v_k, w_k, q_k, P_k) \text{ and } ci_k \neq \bot\}\right| > min. \end{cases}$$

$$\text{(create)} \quad \frac{P_i \xrightarrow{X \leftarrow \mathbf{create}(rt_j, C)}_{i,\sigma} P_i'}{(comps, roles) \xrightarrow{i:X \leftarrow \mathbf{create}(rt_j, C)}_{\text{HEL}} (comps, roles')} \quad \text{with } \sigma = (comps, roles)$$

$$\text{if} \begin{cases} (1)\ i \in dom(roles), roles(i) = (rt_i, ci_i, at^r_i, v_i, w_i, q_i, P_i), \\ \quad comps(ci_i) = (ct_i, at^c_i, as_i) \\ (2)\ as_i(C) \in dom(comps) \\ (3)\ \text{there does not exist } k \in dom(roles) \text{ such that,} \\ \quad roles(k) = (rt_j, as_i(C), at^r_k, v_k, w_k, q_k, P_k), \\ (4)\ roleconstraints(rt_j) = (min, max, cap), \\ \quad \left|\{k \mid roles(k) = (rt_j, ci_k, at^r_k, v_k, w_k, q_k, P_k) \text{ and } ci_k \neq \bot\}\right| < max \\ (5)\ \mathbf{roleBehavior}\ rt_j = P_j, \\ (6)\ roles' = roles[i \mapsto (rt_i, ci_i, at^r_i, v_i[X \mapsto next(roles)], w_i, q_i, P_i')] \\ \qquad\qquad [next(roles) \mapsto (rt_j, as_i(C), \emptyset, \emptyset[\mathbf{self} \mapsto next(roles)], \emptyset, \varepsilon, P_j)]. \end{cases}$$

$$\text{(get)} \quad \frac{P_i \xrightarrow{X \leftarrow \mathbf{get}(rt_j, C)}_{i,\sigma} P_i'}{(comps, roles) \xrightarrow{i:X \leftarrow \mathbf{get}(rt_j, C)}_{\text{HEL}} (comps, roles')} \quad \text{with } \sigma = (comps, roles)$$

$$\text{if} \begin{cases} (1)\ i \in dom(roles), roles(i) = (rt_i, ci_i, at^r_i, v_i, w_i, q_i, P_i), \\ \quad comps(ci_i) = (ct_i, at^c_i, as_i) \\ (2)\ as_i(C) \in dom(comps) \\ (3)\ \text{there exists } j \in dom(roles), roles(j) = (rt_j, as_i(C), at^r_j, v_j, w_j, q_j, P_j) \\ (4)\ roles' = roles\ [i \mapsto (rt_i, ci_i, at^r_i, v_i[X \mapsto j], w_i, q_i, P_i')]. \end{cases}$$

Figure 3.3: SOS rules for the evolution of ensembles (part 1)

(item (2)). Secondly, the new role instance of type rt_j can only be created on a component identified by C which does not yet play a role of the same type rt_j (item (3)). Intuitively, this restriction is reasonable since a component cannot play the same role twice in one ensemble. It can only play it twice when participating in two different ensembles. Thirdly, the new role instance of type rt_j can only be created if the maximal number of instances for this role types required in the ensemble specification has not yet been reached (item (4)). Other conditions, like that the component instance identified by C is of a type which is allowed to adopt a role of type rt_j, are already guaranteed due to well-formedness of process expressions (cf. Sec. 2.4.2). If the two aforementioned conditions are satisfied and the role instance i can execute the create action on role level, the ensemble can also evolve by the corresponding create action. In the resulting ensemble state (item (6)), a new role instance with the identifier $next(roles)$ has been added which is in its initial state. Furthermore, the role instance i which issued the

i which wants to create the new role instance. With this identifier $as_i(C)$, we then determine whether there currently exists a role instance k which is of type rt_j and is adopted by the component with the identifier $as_i(C)$.

creation stores a reference to the newly created role by setting the value of the role instance variable X in its local environment function v_i to $next(roles)$. Obviously, also the remaining behavior of the role instance i changed to P' after executing the create action.

To be able to execute a get action on ensemble level (rule get), two conditions have to be satisfied: the component identified by C must exist and be defined (item (2)) and in the current ensemble state, there must exist a role instance j of the desired role type rt_j which is currently adopted by the component instance identified by C, or rather the component instance with the identifier $as_i(C)$ as before (item (3)). If this condition is satisfied and a role instance i can execute the get action on role level, the ensemble can also evolve by the corresponding get action. In the resulting ensemble state (item (4)), the role instance i which issued the get action stores a reference to the retrieved role by setting the value of the role instance variable X in its local environment function v_i to j. Obviously, also the remaining behavior of the role instance i changed to P' after executing the get action.

$$\text{(send)} \quad \frac{P_i \xrightarrow{Y!msgnm(X)(e)}_{i,\sigma} P_i'}{(comps, roles) \xrightarrow{i:Y!msgnm(X)(e)}_{\text{HEL}} (comps, roles')} \quad \text{with } \sigma = (comps, roles)$$

$$\text{if} \begin{cases} (1)\ i \in dom(roles), roles(i) = (rt_i, ci_i, at_i^r, v_i, w_i, q_i, P_i), \\ (2)\ v_i(Y) = j \in dom(roles), roles(j) = (rt_j, ci_j, at_j^r, v_j, w_j, q_j, P_j), \\ \quad roleconstraints(rt_j) = (min, max, cap), |q_j| < cap, \\ (3)\ v_i(X) = k \in dom(roles) \\ (4)\ roles' = roles[i \mapsto (rt_i, ci_i, at_i^r, v_i, w_i, q_i, P_i')] \\ \qquad\qquad [j \mapsto (rt_j, ci_j, at_j^r, v_j, w_j, q_j \cdot msgnm(k)([\![e]\!]_{i,\sigma}), P_j)]. \end{cases}$$

$$\text{(receive)} \quad \frac{P_i \xrightarrow{?msgnm(X:rt_j)(x:dt)}_{i,\sigma} P_i'}{(comps, roles) \xrightarrow{i:?msgnm(X:rt_j)(x:dt)}_{\text{HEL}} (comps, roles')} \quad \text{with } \sigma = (comps, roles)$$

$$\text{if} \begin{cases} (1)\ i \in dom(roles), roles(i) = (rt_i, ci_i, at_i^r, v_i, w_i, msgnm(j)(e) \cdot q_i, P_i), \\ (2)\ j \in dom(roles), \\ (3)\ roles' = roles[i \mapsto (rt_i, ci_i, at_i^r, v_i[X \mapsto j], w_i[x \mapsto e], q_i, P_i'). \end{cases}$$

Figure 3.4: SOS rules for the evolution of ensembles (part 2)

The SOS rule of a send action formalizes one condition when a role instance i can send a message $msgnm(X)(e)$ to a role instance identified by the variable Y (rule send): The input queue of the role instance Y has not yet exceeded its maximal capacity determined in the underlying ensemble specification (item (2)). Thereby, the variable Y is not a role instance identifier, but refers to a variable whose value $v_i(Y) = j$ has to be retrieved from the local environment function v_i. Other conditions, like that the sending role must support the message as outgoing message and the receiving role as incoming message, are already guaranteed due to well-formedness of process expressions (cf. Sec. 2.4.2). If the aforementioned condition is satisfied and the role instance i can execute the send action on role level, the ensemble can also evolve by the corresponding send action. In the resulting ensemble state (item (4)), the message has been placed at the end of the input queue of the receiving role instance j. Obviously, also the remaining behavior of the role instance i changed to P' after executing the send action.

To receive a message $msgnm(X{:}rt_j)(x{:}dt)$ on a role instance i in an ensemble (rule receive), only one condition has to be satisfied: The first entry of the input queue of the role instance i must be an appropriate message (item (1)). Other conditions, like that the receiving role must support the message as incoming message, are already guaranteed due to well-formedness of process expressions (cf. Sec. 2.4.2). If the aforementioned condition is satisfied and the role instance i can execute the receive action on role level, the ensemble can also evolve by the corresponding receive action. In the resulting ensemble state (item (3)), the role instance i has received the message and has stored the received value j for the role instance parameter X in its local environment function v_i and the received value e for the data parameter x in its local environment function w_i. Obviously, also the remaining behavior of the role instance i changed to P' after executing the receive action.

When calling an operation from a role instance i on its owning component (rules op call 1 and op call 2) in an ensemble, no special conditions have to be satisfied. Whenever the role instance i can execute the operation call, the ensemble can also evolve by the corresponding operation call. Most interesting is the resulting ensemble state after the execution of the operation call. In HELENA, we do not specify any effects of an operation, e.g., by pre- and post-conditions. Thus, an operation can have arbitrary side-effects on the owning component instance where the operation is called. That means if an operation without return value is called (rule op call 1), it evolves the global ensemble state by locally evolving the remaining behavior of the calling role instance i to P' (item (2)) and by arbitrarily, but well-definedly changing the local state of the owning component instance ci_i while all other local states of component instances have to remain unchanged (item (3)). If an operation with a return value is called (rule op call 2), the variable x for storing the return value is set to an arbitrary value e in the local environment function w_i of the role instance i in the resulting ensemble state (item (2)). This reflects that in HELENA we do not specify the effect of an operation and therefore the return value is arbitrary. Furthermore, the operation can have side-effects on the local state of the owning component instance ci_i as before (item (3)).

If data should be changed on the owning component of a role instance i in a defined way, the role instance uses the component attribute setter i:**owner**.*attr* (rule comp attr). No special conditions have to be satisfied to execute that action on the ensemble level. Whenever the role instance i can execute the component attribute setter, the ensemble can also evolve by the corresponding action. In the resulting ensemble state, the value of the attribute *attr* in the attribute value function at_i^c of the owning component instance ci_i of the role instance i is set to the evaluation $[\![e]\!]_{i,\sigma}$ of the expression e (item (2)). Obviously, also the remaining behavior of the role instance i changed to P' after executing the receive action (item (3)).

Similarly, if data should be changed on the role instance i in an ensemble, the role instance uses the role attribute setter i:**self**.*attr* (rule role attr). No special conditions have to be satisfied to execute that action on the ensemble level. Whenever the role instance i can execute the role attribute setter, the ensemble can also evolve by the corresponding action. In the resulting ensemble state (item (2)), the value of the attribute *attr* in the attribute value function at_i^r of the role instance i is set to the evaluation $[\![e]\!]_{i,\sigma}$ of the expression e. Obviously, also the remaining behavior of the role instance i changed to P' after executing the receive action.

Finally, whenever a role instance i in the current ensemble state can execute a state label action *label*, the action can also be executed in the ensemble (rule label). In the resulting ensemble state (item (2)), only the remaining behavior of the role instance i

changed to P' after executing the label action. That means that state labels do not contribute to the goal-oriented behavior of a role. They are used for verification purposes only.

(op call 1)
$$\frac{P_i \xrightarrow{\mathbf{owner}.opnm(e)}_{i,\sigma} P_i'}{(comps, roles) \xrightarrow{i:\mathbf{owner}.opnm(e)}_{\text{HEL}} (comps', roles')} \quad \text{with } \sigma = (comps, roles)$$
$$\text{if } \begin{cases} (1)\ i \in dom(roles), roles(i) = (rt_i, ci_i, at_i^r, v_i, w_i, q_i, P_i), \\ (2)\ roles' = roles[i \mapsto (rt_i, ci_i, at_i^r, v_i, w_i, q_i, P_i')], \\ (3)\ (comps', roles') \text{ well-defined} \\ \quad \text{and for all } ci \in comps \text{ with } ci \neq ci_i \text{ holds } comps'(ci) = comps(ci). \end{cases}$$

(op call 2)
$$\frac{P_i \xrightarrow{x=\mathbf{owner}.opnm(e)}_{i,\sigma} P_i'}{(comps, roles) \xrightarrow{i:x=\mathbf{owner}.opnm(e)}_{\text{HEL}} (comps, roles')} \quad \text{with } \sigma = (comps', roles)$$
$$\text{if } \begin{cases} (1)\ i \in dom(roles), roles(i) = (rt_i, ci_i, at_i^r, v_i, w_i, q_i, P_i), \\ (2)\ roles' = roles[i \mapsto (rt_i, ci_i, at_i^r, v_i, w_i[x \mapsto e'], q_i, P_i')] \\ \quad \text{for an arbitrary value } e', \\ (3)\ (comps', roles') \text{ well-defined} \\ \quad \text{and for all } ci \in comps \text{ with } ci \neq ci_i \text{ holds } comps'(ci) = comps(ci). \end{cases}$$

(comp attr)
$$\frac{P_i \xrightarrow{\mathbf{owner}.attr=e}_{i,\sigma} P_i'}{(comps, roles) \xrightarrow{i:\mathbf{owner}.attr=e}_{\text{HEL}} (comps', roles')} \quad \text{with } \sigma = (comps, roles)$$
$$\text{if } \begin{cases} (1)\ i \in dom(roles), roles(i) = (rt_i, ci_i, at_i^r, v_i, w_i, q_i, P_i), \\ (2)\ comps' = comps[ci_i \mapsto (ct_i, at_i^c[attr \mapsto [\![e]\!]_{i,\sigma}], as_i)] \\ (3)\ roles' = roles[i \mapsto (rt_i, ci_i, at_i^r, v_i, w_i, q_i, P_i')]. \end{cases}$$

(role attr)
$$\frac{P_i \xrightarrow{\mathbf{self}.attr=e}_{i,\sigma} P_i'}{(comps, roles) \xrightarrow{i:\mathbf{self}.attr=e}_{\text{HEL}} (comps, roles')} \quad \text{with } \sigma = (comps, roles)$$
$$\text{if } \begin{cases} (1)\ i \in dom(roles), roles(i) = (rt_i, ci_i, at_i^r, v_i, w_i, q_i, P_i), \\ (2)\ roles' = roles[i \mapsto (rt_i, ci_i, at_i^r[attr \mapsto [\![e]\!]_{i,\sigma}], v_i, w_i, q_i, P_i')]. \end{cases}$$

(label)
$$\frac{P_i \xrightarrow{label}_{i,\sigma} P_i'}{(comps, roles) \xrightarrow{i:label}_{\text{HEL}} (comps, roles')} \quad \text{with } \sigma = (comps, roles)$$
$$\text{if } \begin{cases} (1)\ i \in dom(roles), roles(i) = (rt_i, ci_i, at_i^r, v_i, w_i, q_i, P_i), \\ (2)\ roles' = roles[i \mapsto (rt_i, ci_i, at_i^r, v_i, w_i, q_i, P_i')]. \end{cases}$$

Figure 3.5: SOS rules for the evolution of ensembles (part 3)

3.4 Semantic Labeled Transition System

The semantic rules given in the previous subsection generate a labeled transition system with ensemble states evolving by role instance creation and retrieval, communication actions of roles, and access to the owning component.

Def. 3.7: Semantics of an Ensemble Specification

Let EnsSpec be an ensemble specification over a set CT of component types. Given an admissible ensemble state σ_{init}, the semantics of the ensemble specification EnsSpec is the labeled transition system $T_{\text{HEL}} = (S_{\text{HEL}}, I_{\text{HEL}}, A_{\text{HEL}}, \rightarrow_{\text{HEL}})$ with $I_{\text{HEL}} = \{\sigma_{init}\}$ which is generated by the structured operational semantic rules in Fig. 3.3, Fig. 3.4 and Fig. 3.5.

The states S_{HEL} of the generated labeled transition system are all (well-defined) ensemble states of the ensemble specification *EnsSpec*, the set I_{HEL} of initial states only contains σ_{init}, the actions A_{HEL} are all actions on ensemble level, and the transitions in $\rightarrow_{\text{HEL}}$ are described by the SOS rules in Fig. 3.3, Fig. 3.4 and Fig. 3.5.

3.5 Related Work

In the literature, many process algebras of concurrent communicating processes exist. We focus on three representatives of them which are quite similar to our HELENA approach: The Fork Calculus by Havelund and Larsen [HL93], a process algebra using the special *fork* operator to put processes in parallel; PROMELA, a language for modeling systems of concurrent processes and the input language for the model-checker Spin [Hol03], and its incremental semantics by Weise [Wei97]; and SCEL by De Nicola et al. [DLPT14], a language to model systems of autonomic components.

First of all, they all share the idea of concurrently executing, communicating processes, but they employ different communication styles. In the Fork Calculus, two processes synchronously communicate on complementary actions. PROMELA allows synchronous and asynchronous message passing via global channels. Components in SCEL exchange knowledge via dedicated knowledge repositories, similarly to tuple spaces, where knowledge is asynchronously put and retrieved in the form of data tuples. HELENA is most similar to PROMELA in its communication style. Messages are asynchronously exchanged (the extension to synchronous message passing is straightforward). However, in contrast to PROMELA, messages in HELENA are received on a dedicated input queue per role which is not globally available like channels in PROMELA.

The three approaches also differ in the handling of data. The Fork Calculus does not include data. In PROMELA, data is allowed as global and local variables of processes, as parameters of processes and as content of messages. In SCEL, data is stored as tuples in knowledge repositories and put and retrieved from them via special knowledge repository manipulation actions. HELENA again resembles PROMELA in the treatment of data. Data is stored in attributes of components and roles similarly to local variables and parameters of processes. Furthermore, roles exchange data as content of messages.

Dynamic process creation as proposed by dynamic role creation in HELENA has not found much attention in the literature about concurrent communicating processes. Bergstra [Ber92] uses an environment operator which allows to place the newly created process in parallel with the process initiated the process creation. Thus, he mainly exploits parallel composition to express process creation. Baeten [BV92] proposes a more direct way to express process creation. He uses continuations which allow a process with dynamic process creation to continue with either process expression, the newly created process or the process which initiated the process creation. The three approaches, to which HELENA is mainly related, all allow to create a new process which is executed in parallel to all existing processes in the system. The Fork Calculus introduces the special

fork operator. The operator allows to start a separate evaluation of a process expression and runs it in parallel with all other processes. The *run* operator of PROMELA spawns a new process instance of a certain process type. Again, this new process is run in parallel with all other processes of the system. SCEL provides the *new* action to create a new component. This component executes its behavior modeled as a (possibly) parallel process concurrently to all other components in the system. Similarly, HELENA allows to dynamically create a new role instance from within another role behavior. The newly created role then executes its behavior in parallel to all other roles.

The main difference to all these approaches is that HELENA not only has the concept of active role instances executing their goal-directed behavior. The approach additionally employs components which serve as data storage and computing resources for the active roles. Components thereby provide their capabilities to their adopted roles such that in the semantics of a HELENA ensemble specification, roles and their progress of execution as well as components with their data state have to be handled.

In general, HELENA and the three approaches share the idea of describing their semantics by transition systems. They all split the semantics in two levels, the first level representing the evolution of a single entity and the second level representing the concurrent execution of all entities. In the Fork Calculus, structured operational semantics (SOS) rules are defined for the execution of a process in isolation. The SOS rules for programs are derived from the single process rules as an interaction between a multiset of processes. Similarly, in PROMELA, first the semantics of the behavior of a single process is defined before the behavior of the complete system is derived from a set of interacting processes. In SCEL, the execution of a process retrieves commitments for the process, i.e., actions which the process can perform and continuations how the process would proceed after executing the actions. With these commitments, a system configuration composed from several components executing the aforementioned processes is evolved. Similarly, in HELENA, we describe the evolution of a single role instance without considering the interaction with other roles and components on the first level. On the second level, we evolve all role instances of an ensemble in parallel and take into account communication. In contrast to the other three approaches, roles thereby do not only communicate with other roles in HELENA, but also with components. Therefore, we have to manage not only the local states of roles for an ensemble, but also the local states of components.

3.6 Publication History

The semantics of ensemble specifications has first been defined by ensemble automata in [HK14] and [KH14]. These publications verbosely describe ensemble states by families of functions and ensemble automata by the evolution of ensemble states by ensemble actions. In [HKW15], we propose a precise SOS semantics for a subset of HELENA ensemble specifications which is based on the previous notion of ensemble states and their evolution. This chapter extends the SOS semantics to full HELENA ensemble specifications. In particular, the simplified SOS semantics in [HKW15] does not consider the level of components which is an essential part of the full semantics. The simplified semantics omits any notion of data which is included in the full semantics as values of (component or role) attributes, content of messages, and guards in role behaviors. Finally, the semantics of the restricted set of process constructs and actions is extended to the full syntactic constructs of HELENA. This extension is particularly important for the treatment of guards, the if-then-else construct, and nondeterministic choice.

3.7 Present Achievements and Future Perspectives

Present Achievements: The semantics of HELENA ensemble specifications is described by labeled transition systems. These labeled transition systems follow structured operational semantics (SOS) rules to evolve the states an ensemble can be in according to the ensemble specification.

An ensemble state captures the states of the components and roles forming the system. Since a component is thought of as a pure data container and computing resource in HELENA, the state of a component is just characterized by information currently stored on the component. In contrast, a role is meant to execute a goal-directed behavior and to store information only relevant for that behavior. Thus, the state of a role is characterized by the information currently stored on the role and all information related to the progress of behavior execution (like the current content of the input queue or the remaining behavior to be executed). Furthermore, the role keeps a reference to its owning component to access the data stored on the component and to exploit the computing resources of the component by calling operations.

SOS rules determine the allowed transitions between those ensemble states according to the ensemble specification. Inspired by [HL93] and [Wei97], we split the rules into two layers: The first layer describes the evolution of a single role instance and its corresponding role behavior without considering the interaction with other roles and components. The second layer describes the concurrent evolution of all role instances of an ensemble. It formalizes the interaction between roles, like sending a message to a role by putting it into its input queue, and the communication with components, e.g., to access data stored on the components or to call operations on the components.

The main aspects of the semantics are that HELENA allows to dynamically create new role instances and employs asynchronous communication between roles. From the technical side, it is important to note that in the HELENA semantics neither the selection of a branch in guarded choice nor process invocation does take a separate step. Both directly execute the first action of the selected branch or the invoked process resp. This is in-line with other process algebras like SCEL [DLPT14], but is sometimes differently handled as in PROMELA, the input language for the model-checker Spin [Hol03]. In contrast, termination of a role behavior has to take an additional step to allow the owning component to quit playing the role before the role terminates its execution. Lastly, it remains to mention that no effects are specified for operations in HELENA and thus operations can have arbitrary side-effects on the owning component instance of the calling role instance.

Future Perspectives: For future work, there exist some interesting extension points:

Effect of Operations: So far, we cannot formalize any specific effects of operation calls. To be able to formalize specific effects, operations could be extended by pre- and post-conditions. These would allow to specify the particular tasks which a component can fulfill for a role by executing operations.

Interferences between Roles: According to the proposed semantics, an operation call is handled as an atomic step such that concurrent operation calls cannot happen. However, if we allowed an operation to consist of several steps and additionally allowed to specify precise effects of operations, it might happen that two concurrently called operations interfere which each other and their effects contradict each other. For example, one operation tells the component to move to a certain

location while another advises the component to remain at the current location. Solutions to this problem could be to only allow the execution of a single operation at a time (known as `synchronized` methods in Java) or to define which roles can only mutually exclusively be adopted.

Shared Components between Ensembles: In the semantics of HELENA, the local states of components are part of the state of an ensemble. However, the idea of HELENA is to employ several ensembles concurrently on the same underlying component-based platform. Hence, the local states of components should be shared between different ensembles, but this comes with the problems of interferences again. Whenever, the data on a component is changed during the execution of one ensemble, it is also changed for all other ensembles which rely on the same set of components. Problems similar to database reads and writes may arise, i.e., an ensemble reads data from a component which is immediately outdated due to a write from another ensemble.

Openness for New Components: Finally, we restricted the set of components to a fixed number. In the semantics, the local states of components can be changed, but new components cannot be added or removed. However, if we allow to dynamically extend or shrink the set of represented components, we come across the problem what happens to roles which are currently played by a leaving component. A new action to transfer a role to another owning component may solve the problem of a leaving component, but could also help to exploit new resources if a component joins.

Chapter 4

Goal Specifications
Being Successful with HELENA

Ensembles are formed to collaborate for some global goal. Following [vL09], a goal can be an *achieve goal*, such that the ensemble terminates when the goal (specified, e.g., by a particular state) is reached, or a *maintenance goal*, such that a certain property (specified, e.g., by an ensemble invariant) is maintained while the ensemble is running. Such goals are often described by linear temporal logic (LTL) formulae [DvLF93, DAC99] to allow formal verification of goal satisfaction in the underlying model. In this chapter, we introduce HELENA LTL, a logic based on LTL with HELENA-specific atomic propositions to be able to specify goals for a HELENA ensemble specification. We rely on the formal semantics of HELENA to define satisfaction of LTL formulae over a HELENA specification.

In the following, we first explain the general notion of goals and their representation in LTL in Sec. 4.1. Sec. 4.2 defines HELENA LTL formulae and their satisfaction over HELENA specifications. We conclude this chapter with a short outlook about future work in Sec. 4.4.

4.1 Goals and their Specification in LTL

The notion of goals has widely been used in the field of requirements engineering. KAOS [vL09] is one of the most famous frameworks of goal-oriented requirements engineering and we base our understanding of goals on their notions. In KAOS, the system and its environment is seen as a collection of active components or agents, i.e., some agents define the system while others define its environment. According to van Lamsweerde [vL03], a "*goal* is a prescriptive statement of intent about some system (existing or to-be) whose satisfaction in general requires the cooperation of some of the agents forming that system". That means that a goal is a high-level strategic objective which the system should achieve. Each agent plays a certain role towards achieving the goal. By refining a goal into a set of subgoals where each subgoal is realizable by a single agent, the high-level goal is operationalized into low-level requirements. A requirement thus represents a way of achieving a (part of a) goal. It defines the contribution of a single agent and therefore its role in the collaboration.

Ensemble-based systems as considered in the HELENA approach are large distributed systems where components dynamically collaborate for a high-level objective. Each component participates in a certain role in the ensemble. Similarly to KAOS, we describe the high-level objective of an ensemble by a global goal. Each role in the ensemble

intends to contribute a particular functionality to achieving the goal and is thus an operationalization of (a part of) the goal.

4.1.1 Goal Types

In the literature of goal-oriented requirements engineering, goals have been classified according to different axes [vL01, vL09].

Concern: The first axis describes the kind of properties the goal is concerned with. *Functional goals* are services or functions which the system should provide. Given a certain input, the system should execute a specific behavior to produce the desired output. Opposed to that, *non-functional goals* are concerned with the quality of the system. Thus, they describe the performance of the system and constraints on how the system provides the desired functionalities, e.g., time bounds or security levels.

Time: A taxonomy that is especially chosen when considering goals is the types of temporal behavior prescribed by the goal. *Achieve goals* describe a property or a system state that is established at some point in the future; analogously, *cease goals* describe a property which is ceased at some point in the future. *Maintain goals* require the system to maintain a certain property throughout the whole lifetime of the system; analogously, *avoid goals* require it to avoid a certain state.

Achievement: The third axis considers whether a goal can explicitly be achieved. A *hard goal* is a property of the system whose satisfaction can strictly be verified while a *soft goal* is not characterized by a clear-cut criterion. A soft goal is always only satisfied to a certain degree and complete achievement is not possible. Soft goals can be specialized to *optimize goals*. Optimize goals define an objective function according to which different behaviors are evaluated. The behavior maximizing the function is favored over the others.

In ensemble-based systems, components collaborate to perform goal-oriented tasks. Therefore, we focus on functional goals, i.e., the ensemble executes a specific behavior to provide a certain functionality. Furthermore, an ensemble can pursue two types of goals: either it strives to achieve a certain goal or to maintain a certain property throughout execution. In both cases, we only consider hard goals to be able to verify satisfaction.

Example: The transfer ensemble in the p2p example pursues two goals. On the one hand, the ensemble is formed to transfer a file from the providing peer to the requesting peer if the file is present in the p2p network. Hence, we formulate an achieve goal to reach a state in the running system where the requesting peer has the file. On the other hand, the ensemble has to guarantee that whenever the file is available in the system it should not accidentally be deleted. Thus, the system has to fulfill the maintain goal that one peer always has the file if it was present in the initial state.

4.1.2 Linear Temporal Logic

Besides giving an informal (but intuitive) description of goals as in the previous subsection, we formally specify them. Together with a formal specification of the system, this allows to check satisfaction of goals for the intended system.

A popular approach to specify goals is linear temporal logic (LTL). It is especially suitable to describe properties which have to be achieved at some point of time or

maintained throughout the whole lifetime of a system. LTL are formulae built from a set of atomic propositions and logic operators. The atomic propositions of LTL formulae are simple properties which have a specific boolean value in each state of the system.

Def. 4.1: Linear Temporal Logic [BK08]

Let AP be a set of atomic propositions. LTL formulae over *AP are then inductively defined by:*

$$\begin{aligned} \phi \quad &= p \in AP && \textit{(atomic proposition)} \\ &\mid \neg\phi \mid \phi \wedge \psi && \textit{(proposition logic operators)} \\ &\mid \boldsymbol{X}\phi \mid \Diamond\phi \mid \Box\phi \mid \phi \boldsymbol{U} \psi && \textit{(linear temporal logic operators)} \end{aligned}$$

The set of LTL formulae over AP is denoted by LTL(AP).

Disjunction, implication, and equivalence are given by the usual abbreviations:

$$\begin{aligned} \phi \vee \psi &\equiv \neg(\neg\phi \wedge \neg\psi), \\ \phi \Rightarrow \psi &\equiv \neg\phi \vee \psi, \text{ and} \\ \phi \Leftrightarrow \psi &\equiv (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi). \end{aligned}$$

LTL in Kripke Structures: To define when a given system satisfies an LTL formula, the system must be formally described. The first possibility is to use the notion of Kripke structures. They consist of a set of states connected by (unlabeled) transitions. The states are labeled by sets of atomic propositions which hold in the state and some states are marked as initial states.

Def. 4.2: Kripke Structure

Let AP be a set of atomic propositions. A Kripke structure K over *AP is a tuple* $(S_K, I_K, \rightarrow_K, F_K)$ *such that*

- S_K *is a set of states,*
- $I_K \subseteq S_K$ *is a set of initial states,*
- $\rightarrow_K \subseteq S_K \times S_K$ *is an (unlabeled) transition relation without terminal states (i.e.,* $\forall s \in S_K \exists s' \in S_K \,.\, s \rightarrow_K s'$*), and*
- $F_K : S_K \rightarrow 2^{AP}$ *is a labeling function associating to each state the set of atomic propositions that hold in it.*

For a Kripke structure $K = (S_K, I_K, \rightarrow_K, F_K)$, we further define:

- A *path of* K is an infinite sequence $p = s_0 s_1 s_2 \ldots$ (with $s_i \in S_K$ for all $i \in \mathbb{N}$) such that $s_0 \in I_K$ and $s_i \rightarrow_K s_{i+1}$. A *path fragment of* K is an infinite sequence $p = s_1 s_2 \ldots$ (with $s_i \in S_K$ for all $i \in \mathbb{N}$) such that $s_i \rightarrow_K s_{i+1}$.
- A *trace of* K is an infinite sequence $t = t_0 t_1 t_2 \ldots$ such that there exists a path $p = s_0 s_1 s_2 \ldots$ in K and $t_i = F_K(s_i)$ for all $i \in \mathbb{N}$. A *trace fragment of* K is an infinite sequence $t = t_1 t_2 \ldots$ such that there exists a path fragment $p = s_1 s_2 \ldots$ in K and $t_i = F_K(s_i)$ for all $i \in \mathbb{N}$.

When a system described by a Kripke structure satisfies a goal described by an LTL formula is defined by the usual inductive definition [BK08]. Note that satisfaction is always evaluated according to an infinite trace of the Kripke structure.

Def. 4.3: Satisfaction of LTL in Kripke Structures

Let $K = (S_K, I_K, \rightarrow_K, F_K)$ be a Kripke structure over AP, $t = t_0 t_1 t_2 \ldots$ a trace fragment of K, and $\phi \in LTL(AP)$; $t|_i$ denotes the subsequence $t_i t_{i+1} t_{i+2} \ldots$ of t. The satisfaction of ϕ for trace t, *written $t \models \phi$, is inductively defined by*

- $t \models p$, *if* $p \in t_0$,
- $t \models \neg\phi$, *if* $t \not\models \phi$,
- $t \models \phi \wedge \psi$, *if* $t \models \phi$ *and* $t \models \psi$,
- $t \models \boldsymbol{X}\phi$, *if* $t|_1 \models \phi$,
- $t \models \Diamond\phi$, *if there exists* $k \geq 0$ *such that* $t|_k \models \phi$,
- $t \models \Box\phi$, *if for all* $k \geq 0$ *holds* $t|_k \models \phi$,
- $t \models \phi \boldsymbol{U} \psi$, *if there exists* $k \geq 0$ *such that* $t|_k \models \psi$ *and for all* $0 \leq j < k$ *holds* $t|_j \models \phi$,

The Kripke structure K satisfies an LTL formula ϕ, written $K \models \phi$, if all traces of K satisfy ϕ.

LTL in Labeled Transition Systems: A second possibility is to describe systems as labeled transition systems. In contrast to Kripke structures, they do not label states with atomic propositions, but transitions with actions. Let's recall the definition of labeled transition systems from Def. 3.1 on page 46. A labeled transition system T is a tuple $(S_T, I_T, A_T, \rightarrow_T)$ such that S_T is a set of states, $I_T \subseteq S_T$ is a set of initial states, A_T is a set of actions such that the silent action $\tau \notin A_T$, and $\rightarrow_T \subseteq S_T \times (A_T \cup \tau) \times S_T$ is a labeled transition relation. For an LTS $T = (S_T, I_T, A_T, \rightarrow_T)$, we further define:

- a^* denotes a (possibly empty) sequence of a actions.
- If $w = a_1 \ldots a_n$ holds for some $n \in \mathbb{N}$ and $a_1, \ldots, a_n \in (A_T \cup \tau)$, then $s \xrightarrow{w}_T s'$ stands for $s = s'$, if $n = 0$, and $s \xrightarrow{a_1}_T s_1 \ldots s_{n-1} \xrightarrow{a_n}_T s'$ with appropriate $s_1, \ldots, s_{n-1}$ otherwise.
- The LTS T together with a set of atomic propositions AP and a satisfaction relation $s \models p$ (for $s \in S_T$ and $p \in AP$) induces a Kripke structure $K(T) = (S_T, I_T, \rightarrow_T^\bullet, F)$:
  - The labeled transition relation $\rightarrow_T$ is transformed into an unlabeled, total transition relation $\rightarrow_T^\bullet$ which forgets the actions and adds a new transition $s \rightarrow_T^\bullet s$ for each terminal state $s \in S_T$.
  - The labeling function $F : S_T \rightarrow 2^{AP}$ is defined by $F(s) = \{p \in AP \mid s \models p\}$.

When a system described by a labeled transition system satisfies an LTL formula is defined relying on the induced Kripke structure.

Def. 4.4: Satisfaction of LTL in Labeled Transition Systems

Let $T = (S_T, I_T, A_T, \rightarrow_T)$ be a labeled transition system, AP a set of atomic propositions, $s \models p$ a satisfaction relation for $s \in S_T$ and $p \in AP$, and $\phi \in LTL(AP)$ an LTL formula over AP.
T satisfies ϕ, written $T \models \phi$, if $K(T) \models \phi$, i.e., the induced Kripke structure $K(T)$ satisfies ϕ.

4.1.3 Goal Patterns

To support the engineer of a system, goal-oriented requirements engineering recommends a set of temporal logic patterns to describe achieve and cease goals as well as maintain and avoid goals. Dwyer et al. [DAC99] recommends a set of patterns for the specification of properties in finite-state verification which are similarly proposed for the specification of temporal goals by the KAOS methodology [DvLF93, vL09]. We only focus on the most general form of achieve, cease, maintain, and avoid goals (cf. Fig. 4.1). They can be extended to cope with more complex goals like that a system has to maintain a certain property until a new (possibly more desirable) property is achieved (cf. [vL09, Chap. 17]).

$$\begin{aligned} \text{Achieve Goal: } & P \Rightarrow \Diamond Q, \\ \text{Cease Goal: } & P \Rightarrow \Diamond \neg Q, \\ \text{Maintain Goal: } & P \Rightarrow \Box Q, \\ \text{Avoid Goal: } & P \Rightarrow \Box \neg Q, \end{aligned}$$

Figure 4.1: Patterns for temporal goals formulated in LTL (taken from [DvLF93])

An achieve goal expresses that a certain state of the system has to be reached at some point of time in the future. The formalization in LTL thus requires by the term $\Diamond Q$ that the property Q eventually holds at some point in the future. However, it might be that the system only has to achieve Q if it is started in a certain state. This is expressed by the implication $P \Rightarrow$ to indicate the initial property P which requires achieving the property Q at some point in the future. Analogously, a cease goal requires the system to cease the property Q at some point in the future expressed by the term $\Diamond \neg Q$.

In contrast, a maintain goal demands to fulfill a certain property throughout the lifetime of the system. The formalization in LTL thus requires by the term $\Box Q$ that the property Q always holds. Similarly to an achieve goal, it might by that the property Q must only be maintained if the system is started in a certain state expressed by the implication $P \Rightarrow$. Analogously, an avoid goal requires the system to always avoid the property Q expressed by the term $\Box \neg Q$.

Example: To express the goals for the p2p example which were informally defined in the previous section, we have to assume that the system exposes some properties about its state. Firstly, it exposes by $Peer[i]{:}hasFile$ whether the ith peer currently has the file. Secondly, the term $Requester{:}hasFile$ denotes whether the single requester in the file transfer ensemble has the file. Since LTL does not allow quantification, we have to assume given a set of peers $Peer[1]$, $Peer[2]$, and $Peer[3]$ to express our goals.

The achieve goal that the requester has the file at some point in the future if it exists in the p2p network is expressed by the LTL formulae in Fig. 4.2

$$(Peer[1]{:}hasFile \vee Peer[2]{:}hasFile \vee Peer[3]{:}hasFile) \Rightarrow \Diamond Requester{:}hasFile.$$

Figure 4.2: Achieve goal for the p2p example in LTL

The first part of the implication determines whether the file exists in the p2p network in the initial state of the system. Since we assumed that the system only consists of three peers, it is enough to include them in the premise. The second part of the implication

indicates that the ensemble actually achieves its goal at some point in time by specifying that the single requester in the network eventually has the file. Since we formulated the goal as an implication, we require the system only to transfer the file if the file is present in the p2p network; otherwise, the system does not have any achieve goal.

The maintain goal for the transfer file ensemble that the file is not accidentally deleted throughout the lifetime of the system if it was present in the initial state of the system is expressed by the LTL formula in Fig. 4.3. We start from the same premise as

$$(Peer[1]{:}hasFile \vee Peer[2]{:}hasFile \vee Peer[3]{:}hasFile) \\ \Rightarrow \Box(Peer[1]{:}hasFile \vee Peer[2]{:}hasFile \vee Peer[3]{:}hasFile).$$

Figure 4.3: Maintain goal for the p2p example in LTL

before that one peer should have the file. Then, the system has to always maintain the file in the system which is expressed by the second part of the implication. It indicates that at least one peer has the file at all points of time (this must not be the initial owner of the file).

4.1.4 Reasoning about Goals

A formal specification of goals does not only provide a precise formulation of the criterion when a goal is satisfied. The formal specification can also be used for reasoning about goals [vL01].

Firstly, we can use the formalization of goals to validate them. From their formal representation, concrete scenarios can be generated. They describe typical examples or counterexample of the intended system behavior which helps to identify the desired system behavior. We can even go further and detect overlapping or conflicting goals which must be resolved to gain a clear goal specification.

Secondly, we can apply goal verification on a given system specification such that the satisfaction of goals is checked for a the system specification. Techniques of model-checking are used to verify that the specified system actually meets its goal. We therefore can guarantee that the specified system is an operationalization which achieves the desired goals.

Thirdly, formal goal specifications can be used to derive a system's operationalization. Requirements of single agents are derived from goal specifications, for example, by goal refinement. They in turn are used to elicit concrete pre- and post-conditions for operations of agents.

In the following, we will focus on goal verification for ensemble-based systems. We describe goals for ensembles by LTL formulae and verify their satisfaction for a given Helena ensemble specification.

4.2 Helena LTL

To express achieve and maintain goals over Helena ensemble specifications, we use LTL formulae over a set of particular atomic Helena propositions. Thereby, we can refer to properties of role instances and component instances, but also to states in role behaviors. In the following, we formally define LTL formulae over a Helena specification in Sec. 4.2.1. In Sec. 4.2.2, we discuss satisfaction of LTL formulae over a Helena specification based on the semantics presented in Chap. 3.

4.2.1 Helena LTL Formulae

To express Helena LTL formulae, we assume given a Helena ensemble specification $EnsSpec = (\Sigma, behaviors)$ (over a set of component types CT) where Σ is an ensemble structure and *behaviors* is a set of extended role behaviors for each role type occurring in Σ. In the Helena LTL formulae, we can either refer to particular states of role behaviors by state labels or to values of role attributes or component attributes.

Def. 4.5: Helena LTL

Let EnsSpec be a Helena *ensemble specification. The* set $AP(EnsSpec)$ of atomic propositions for *EnsSpec consists of either state label expressions or a boolean expressions over attributes.*

(1) A state label expression *is of the form* $rt[n]@label$ *where* rt *is a role type of* Σ, $n \in \mathbb{N}^+$ *is the identifier of a role instance of* rt, *and label is a state label in a role behavior of the set behaviors.*

(2) An attribute expression *has to be boolean and is built from the usual arithmetic and relational operators, data constants, and expressions of the form* $rt[n]{:}attr$ *or* $ct[n]{:}attr$ *where* $n \in \mathbb{N}^+$ *is the identifier of a role instance or component instance,* rt *is a role type and attr is a role attribute of* rt, *or* $ct \in CT$ *is a component type and attr is component attribute of ct.*

A Helena LTL formula for *EnsSpec is an LTL formula over the set* $AP(EnsSpec)$ *of the atomic* Helena *propositions.*

Example: For our p2p example, we repeat the two goals from Sec. 4.1. To recap, the achieve goal that the requesting peer eventually has the file is expressed by the LTL formula in Fig. 4.4 (which is the same as in Fig. 4.2). In the formula, only state label

$$(Peer[1]{:}hasFile \vee Peer[2]{:}hasFile \vee Peer[3]{:}hasFile) \Rightarrow \Diamond Requester{:}hasFile.$$

Figure 4.4: Achieve goal for the p2p example in Helena LTL

expressions are used. $Peer[1]{:}hasFile$ refers to the value of the attribute *hasFile* of the component instance with the identifier 1 of component type *Peer*; analogously for the state label expression $Requester{:}hasFile$. However, the latter is a shorthand notation for $Requester[1]{:}hasFile$ which can be used since we know by the minimal and maximal multiplicity constraints for the role type *Requester* in the ensemble structure $\Sigma_{transfer}$ (cf. Fig. 2.4 on page 23), that there exists exactly one requester instance in the running ensemble.

Similarly, the maintain goal that the file will never be accidentally deleted is expressed in Helena LTL in Fig. 4.5 (which is the same as in Fig. 4.3).

$$\begin{aligned}&(Peer[1]{:}hasFile \vee Peer[2]{:}hasFile \vee Peer[3]{:}hasFile)\\ &\quad \Rightarrow \Box(Peer[1]{:}hasFile \vee Peer[2]{:}hasFile \vee Peer[3]{:}hasFile).\end{aligned}$$

Figure 4.5: Maintain goal for the p2p example in Helena LTL

4.2.2 Satisfaction of HELENA LTL Formulae

To determine when an ensemble specification satisfies a HELENA LTL formulae, we rely on Def. 4.4 on page 68 which requires a set of atomic propositions and a satisfaction relation to define satisfaction of LTL formulae for a labeled transition system.

For an ensemble specification *EnsSpec*, the semantic rules of HELENA in Fig. 3.3, Fig. 3.4, and Fig. 3.5 generate a labeled transition system $(S_{\text{HEL}}, I_{\text{HEL}}, A_{\text{HEL}}, \rightarrow_{\text{HEL}})$ for a given admissible initial state $\sigma_{init} \in I_{\text{HEL}}$. The atomic propositions for *EnsSpec* which are used to formulate LTL formulae are the state label expressions and attribute expressions defined by the set $AP(EnsSpec)$ in the previous subsection. Therefore, it remains to define a satisfaction relation $\sigma \models p$ for $\sigma \in S_{\text{HEL}}$ and $p \in AP(EnsSpec)$ (the satisfaction relation serves as labeling function when inducing a Kripke structure from the labeled transition system). Satisfaction of LTL formulae in labeled transition systems was already defined in Def. 4.4 and satisfaction in Kripke structures in Def. 4.3.

Def. 4.6: Satisfaction of HELENA LTL Formulae

Let $T_{\text{HEL}} = (S_{\text{HEL}}, I_{\text{HEL}}, A_{\text{HEL}}, \rightarrow_{\text{HEL}})$ be the labeled transition system of a HELENA *ensemble specification EnsSpec, $\sigma = (comps, roles) \in S_{\text{HEL}}$ be a well-defined ensemble state, and let $AP(EnsSpec)$ be the set of atomic propositions for EnsSpec.*
The ensemble state σ satisfies an atomic HELENA *proposition $p \in AP$, denoted by $\sigma \models p$, if*

(1) p is a state label expression $p = rt[n]@label$ and there exists $n \in dom(roles)$ such that $roles(n) = (rt, ci, at^r, v, w, q, label.P)$ and $ci \neq \bot$,

(2) p is an attribute expression $p = e$ and

- *for every subexpression $rt[n]:attr$ of e:*
  *there exists $n \in dom(roles)$ with $roles(n) = (rt, ci, at^r, v, w, q, P)$ and $ci \neq \bot$ such that the value $at^r(attr)$ combined with the values of the other subexpressions of e evaluates to **true** and*
- *for every subexpression $ct[n]:attr$ of e:*
  *there exists $n \in dom(comps)$ with $comps(n) = (ct, at^c, as)$ such that the value $at^c(attr)$ combined with the values of the other subexpressions of e evaluates to **true**.*

Example: For our p2p example, we can show that both goals are actually met by the file transfer example. However, the representation of ensemble states is quite large and the application of the semantics rules is soon hard to do by hand. Therefore, we support checking our HELENA models for goal satisfaction by an automated model-checking approach which is presented in the following sections.

4.3 Publication History

The content of this chapter relies on [HKW15]. In [HKW15], we propose to specify goals of ensembles by linear temporal logic for a simplified version of HELENA.

This chapter extends [HKW15] by an overview about goals, their specification and reasoning about them. It significantly extends the expressive power of the logic to describe goals by attribute expressions to be able to reason about data states.

4.4 Present Achievements and Future Perspectives

Present Achievements: Ensembles normally collaborate to achieve some global goal. In this chapter, we proposed to express these goals by LTL formulae to allow to check for an ensemble specification whether the specified ensemble reaches its global goal. Atomic propositions in LTL formulae refer two types of expressions: state label expressions and attribute expressions. A state label expression denotes whether a particular role instance reached a certain state label in its executed role behavior. An attribute expression reasons about the value of attributes of particular role instances or component instances.

Future Perspectives: The logic for goal specification gives rise for future advancement. In particular, the formulation of goals is sometimes tiresome. Since we cannot use any quantifiers, full enumeration of all possibilities is needed. For example instead of specifying that all peers have a file, we have to enumerate all peers in the system and must specify for each peer that it has the file. An interesting extension would therefore be to use first-order LTL, i.e., LTL with quantifiers. There also exist approaches on model-checking first-order LTL [WTM04, XSCM04], but most of these approaches only offer prototypic implementations which cannot be used as an off-the-shelf model-checker like Spin.

Chapter 5

Verification
Being Sure about Goal Satisfaction

The previous chapters introduced how to specify an ensemble by its structure and behavior as well as how to formulate the global goals which it strives to achieve in HELENA LTL. In this chapter, we focus on the early (pre-implementation) verification of such HELENA models for their intended global goals. To support automated model-checking, we propose to translate HELENA specifications to PROMELA and check satisfaction of LTL properties with the explicit state model-checker Spin [Hol03]. PROMELA is well-suited as a target language since it supports dynamic creation of concurrent processes and synchronous and asynchronous communication as required by HELENA ensembles.

In the following, we informally describe in Sec. 5.1 our approach of model-checking HELENA LTL formulae for HELENA ensembles specifications by translating them to PROMELA and give an intuition why model-checking results from a PROMELA translation can be transferred back to the original HELENA specification. In Sec. 5.2, we present how HELENA is translated to PROMELA. In Sec. 5.3, we explain the practical application of the model-checker Spin to the translated PROMELA specification and how model-checking results can be mapped back to the original HELENA ensemble specification. We conclude this chapter in Sec. 5.4 by related work and give a short outlook about future work in Sec. 5.6.

We illustrate the translation from HELENA to PROMELA by the p2p example. The translated PROMELA specification together with all goals is shown in Appendix C.2 and on the attached CD in the project `eu.ascens.helenaText.p2p` in the file `promela-gen/p2p-check.pml`.

5.1 Approach for Checking HELENA LTL Formulae

To support automated model-checking of HELENA models against HELENA LTL formulae, we do not provide a HELENA-specific model-checker. We rather rely on the well-established explicit state model-checker Spin [Hol03] by translating HELENA specifications to PROMELA, the input language of Spin. PROMELA [Hol03] is a language for modeling systems of concurrent processes. Its most important features are the dynamic creation of processes and support for synchronous and asynchronous communication via message channels. PROMELA verification models serve as input for the model-checker Spin [Hol03]. On the one hand, Spin can be used to run a randomized simulation of the model. On the other hand, it can check LTL properties, formulated over a PROMELA specification, and find and display counterexamples.

To verify LTL properties for HELENA specifications, we exploit PROMELA and Spin. We systematically translate a HELENA specification to PROMELA[1] and check the specified LTL properties in the translated specification with Spin. Thereby, PROMELA is well-suited as a target language: dynamic role creation in HELENA can be expressed by dynamic process creation in PROMELA, and asynchronous message exchange between roles in HELENA by asynchronous communication via message channels in PROMELA.

In the translation, we have to make two assumptions concerning nondeterministic choice:

(1) Process invocation is not allowed as one of the branches of nondeterministic choice. In HELENA, one of the branches of nondeterministic choice is selected based on its executability. If one branch is a process invocation, executability is decided based on the executability of the first action of the invoked process.

In PROMELA, nondeterministic choice will be represented by the PROMELA **if**-construct which nondeterministically executes one of the branches whose first statement is executable, i.e., nondeterministic choice is not a separate action, but is executed together with the first action of the chosen branch. Furthermore, process invocation will be represented by a **goto**-jump to the beginning of the invoked process. Therefore, for nondeterministic choice, executability of a branch consisting of process invocation would be decided based on the executability of the **goto**-jump rather than the executability of the first action of the invoked process if we allowed process invocation as a branch of nondeterministic choice.

Process invocation is easily avoided as a branch of nondeterministic choice by inlining at least the first action of the invoked process into nondeterministic choice and then invoking the desired process.

(2) The first action of a branch of nondeterministic choice has to be executable if it is translated to a sequence of several PROMELA statements. As we will see in the following, all actions concerning components are translated to a sequence of PROMELA statements, i.e., the **create**- and **get**-action which advise a component to adopt or retrieve a role, calling an operation on a component, and setting a component attribute.

To explain why this assumption is necessary, let's consider a nondeterministic choice construct where the first action of one branch is one of the aforementioned and is not executable. In HELENA, this branch would never be selected for execution since the first action of the branch is not executable.

In PROMELA, nondeterministic choice will be represented by the PROMELA **if**-construct which nondeterministically executes one of the branches whose first statement is executable. However, as we assumed it, the HELENA action will be represented by a sequence of PROMELA statements, i.e., a request to a component instance, some internal computation, and an answer from the component instance (details about the translation follow in the next section). Thereby, the request to the component would always be executable while the latter steps would not be executable if the corresponding HELENA action is not executable. Thus, in PROMELA, the branch with the non-executable HELENA action as first action could be selected for execution since executability in PROMELA is only decided

[1] An automatic code generator exploiting the systematic translation from HELENA to PROMELA is presented in Sec. 8.3

based on the request to the component (and not based on the whole sequence of PROMELA statements representing the HELENA action). Even composing the statements to an indivisible sequence with the **atomic**-block of PROMELA would not overcome this problem, since also for an **atomic**-block executability is decided based on the executability of its first action and not based on the executability of the whole sequence of actions in the block.

However, we do not have to completely forbid all HELENA actions, which are translated to a sequence PROMELA statements, as first action of a branch of nondeterministic choice. It is enough to require that they have to be executable. In terms of actions, the following assumptions must be fulfilled:

(a) The **create**-action is only allowed if the multiplicity of instances of the role type to be created is not yet exceeded and the owning component instance does not yet play the role.

(b) The **get**-action is only allowed if the requested owning component is guaranteed to currently adopt the requested role.

These assumptions are not a real restriction of the expressibility of HELENA. On the one hand, nondeterministic choice is mainly used to allow waiting for several incoming messages like in a server-client-architecture. Then, the control flow is externally triggered by the reception of messages. Therefore, it is mostly important to support nondeterministic choice between message receptions and not between any internal actions like role creation or retrieval. On the other hand, if the application requires to decide between role creation and retrieval and we cannot guarantee aforementioned conditions, we can avoid nondeterministic choice and rather use the if-then-else construct. In the if-then-else construct, we inquire with a plays query whether a component currently plays a certain role. Based on this plays query as a guard of an if-then-else construct, we select from the options to create or to retrieve the corresponding role.

For all other actions, we do not require any assumptions since sending and receiving a message translates to one PROMELA statement only[2], calling a component operation, setting a component attribute or role attribute are always executable. Note that labels are not allowed as first actions of nondeterministic choice due to the well-formedness conditions in Def. 2.10 on page 28.

Two more assumptions concern the if-then-else construct:

(3) Process invocation is not allowed as one of the branches of the if-then-else construct. In HELENA, the evolution of the if-then-else construct evaluates the guard and executes the first action of the corresponding branch in one single step. If the corresponding branch is a process invocation, the first action of the invoked process is directly executed in the same step.

In PROMELA, the if-then-else construct will be represented by the PROMELA **if**-construct where the first statement is the translated guard and the following statements are the translated process expression. The guard and the first (possibly-nested) action are executed as one indivisible sequence by enclosing them into

[2]To be able to translate message reception to only one PROMELA statement, the declaration of local variables to receive the content of the message is shifted to the beginning of the translated role behavior in PROMELA as we will discuss in Fig. 5.12.

the **atomic**-block of PROMELA. However, as already mentioned, process invocation will be represented by a **goto**-jump to the beginning of the invoked process. Therefore, atomicity of the if-then-else construct would be lost in PROMELA if we allowed process invocation as a branch of the if-then-else construct.

Process invocation is easily avoided as a branch of the if-then-else-construct by inlining at least the first action of the invoked process into the if-then-else construct and then invoking the desired process.

(4) The first action of the branch of the if-then-else construct which is chosen based on the evaluation of the guard has to be executable.

To explain why this assumption is necessary, let's consider an if-then-else construct where the guard evaluates to **true**, but the first action of the corresponding process expression in this branch is not executable. In HELENA, the if-then-else construct as a whole would not evolve since executability is decided based on the evaluation of the guard **and** the executability of the first action of the corresponding process expression.

As already mentioned, in PROMELA, the HELENA if-then-else construct will be represented by the PROMELA **if**-construct where the first statement is the translated guard and the following statements are the translated action. In principle, the PROMELA **if**-construct nondeterministically chooses between branches whose first statement is executable, i.e., for the translated HELENA if-then-else construct, it decides for the branch where the guard evaluates to **true**. Only after the selection of the branch, it would check the executability of the translated HELENA action. Since we assumed the action not to be executable, the PROMELA translation would not further evolve. However, in PROMELA, a branch would have had already been chosen while in HELENA the evolution blocked before the complete if-then-else construct and can still choose between both branches.

However, we do not have to completely forbid all HELENA actions as first action of a branch of an if-then-else construct. It is enough to require that they have to be executable. In terms of actions, the following assumptions must be fulfilled:

(a) The **create**-action is only allowed if the multiplicity of instances of the role type to be created is not yet exceeded and the owning component instance does not yet play the role (assumption shared with nondeterministic choice).

(b) The **get**-action is only allowed if the requested owning component is guaranteed to currently adopt the requested role (assumption shared with nondeterministic choice).

(c) Sending a message is only allowed if the capacity of the message queue of the receiving role is not yet exceeded.

(d) Receiving a message is not allowed since we cannot guarantee that a corresponding message can always be received.

These assumptions are not a real restriction of the expressibility of HELENA. For **create**- and **get**-actions, the assumption can be guaranteed by a corresponding plays query in the guard and the multiplicity limit can be set to the number of the underlying components to avoid exceeding it. For sending a message, we normally can assume that capacity limits are not reached if the application is

correctly modeled. Even that message reception is not allowed in the if-then-else construct is not a real restriction since waiting for several incoming messages should normally be expressed with nondeterministic choice.

For all other actions, we do not require any assumptions since calling a component operation and setting a component attribute or role attribute are always executable. Note again that labels are not allowed as first actions of nondeterministic choice due to the well-formedness conditions in Def. 2.10 on page 28.

We make two further general assumptions which can be alleviated in future work:

(5) For simplicity reasons, we assume that each role type can only be adopted by a single component type. This is not a restriction of generality since we plan to extend HELENA by component interfaces which would combine all shared features of a set of similar component types into a single interface which can be translated similarly to one single component type.

(6) Additionally, we only verify a single ensemble per translation. Arrays in PROMELA will help us in future work to realize the extension to several parallel ensembles (possibly of different types).

To be able to transfer model-checking results to the original HELENA specification, we need to formally show that a HELENA specification and its PROMELA translation satisfy the same set of LTL formulae. To prove that kind of semantic equivalence, we establish a stutter trace equivalence between the induced semantic Kripke structures of a HELENA specification and of its PROMELA translation. In Chap. 6, the formal proof of semantic equivalence is shown in full detail for two simplified variants of HELENA and PROMELA and is informally extended to full HELENA and PROMELA.

The following two sections will first introduce the translation from HELENA to PROMELA in full detail. Afterwards, we explain the practical application of the model-checker Spin to PROMELA translations and illustrate it at our p2p example.

5.2 Translation from HELENA to PROMELA

In this section, we discuss how a PROMELA model can be constructed from a HELENA ensemble specification. The PROMELA verification model is then used for model-checking LTL properties with Spin [Hol03]. In Sec. 5.2.1, we first give an overview about the features of HELENA and how they are represented in PROMELA. Afterwards in Sec. 5.2.2, we explain the translation to PROMELA in detail by specifying translation functions for each feature of HELENA. As a proof of concept, we provide an automatic code generator implementing the transformation in Sec. 8.3.

Assumption: In the translation, we make several assumptions as described in the previous section:

- Process invocation is not allowed as one of the branches of nondeterministic choice or the if-then-else process construct(cf. item (1) and (3) in Sec. 5.1).
- The **create**-action is only allowed as first action of a branch in nondeterministic choice or an if-then-else construct if the multiplicity of instances of the role type to be created is not yet exceeded and the owning component instance does not yet play the role (cf. item (2a) and (4a) in Sec. 5.1).

- The **get**-action is only allowed as first action of nondeterministic choice or an if-then-else construct if the requested owning component is guaranteed to currently adopt the requested role (cf. item (2b) and (4b) in Sec. 5.1).
- Sending a message is only allowed as first action of a branch of an if-then-else construct if the capacity of the message queue of the receiving role is not yet exceeded (cf. item (4c) in Sec. 5.1).
- Receiving a message is not allowed as first action of a branch of an if-then-else construct (cf. item (4d) in Sec. 5.1).
- Each role type can only be adopted by a single component type and we only verify a single ensemble instance per translation (cf. item (5) and (6) in Sec. 5.1).
- For simplicity, we additionally only allow integers as data parameters of messages.

In the following, the translation idea is showcased at the p2p example. In particular, we show how the p2p example has to be adapted such that it meets all assumptions without changing the goal-directed behavior. With these adaptations, the complete translated PROMELA specification has 531 lines of code and is listed in Appendix C.2.

5.2.1 Overview

PROMELA is a language for modeling systems of concurrent processes. HELENA, though, employs a two layered approach where components adopt roles. Components are passive and only provide storage and computing resources to their adopted roles while the roles themselves are the active entities in ensembles. To transfer the two layered approach of HELENA to PROMELA, we represent both, components and roles, as processes in PROMELA, but with different communication abilities and behavior.

The process for a component does not actively communicate with other processes. It only waits for requests from its adopted roles on a dedicated input channel, executes some internal computations, and responds with an appropriate answer. Hence, a HELENA component is represented by a long-running PROMELA process. The PROMELA component process is repeatedly able to receive requests form its adopted roles and its progress is completely triggered by those requests.

In contrast to a component, a HELENA role is represented by a short-living PROMELA process. The PROMELA role process reflects the corresponding role behavior declaration specified in HELENA by issuing requests to its underlying component process and actively communicating with other role processes. Therefore, the role process needs a kind of connection to its underlying component process and its communication abilities have to include role-to-component facilities and role-to-role facilities.

In the following, we outline which facilities are needed to allow role-to-component communication and role-to-role communication between the processes in PROMELA and how data stored in component and role attributes is represented in PROMELA. Based on these communication and storage facilities, the main ideas of the translation from HELENA to PROMELA are explained: the (long-running) repeated and externally triggered process for a component, the (short-living) active process for a role, and the interplay between both. Each step is showcased at the p2p example.

5.2.1.1 Role-to-Component Communication

In HELENA, roles communicate with components to advise them to adopt or quit playing other roles, to request references to already adopted roles from them, to request and

set the value of component attributes, or to invoke operations on them. Thus, each PROMELA component process relies on a dedicated synchronous channel **self**, only used for communication between itself and its adopted roles. Conversely, each PROMELA process for a role adopted by the component stores a reference to the very same channel by the name **owner**.

The type of requests which can be sent via this channel is restricted by a dedicated user-defined data type. It only allows to send requests for the creation, retrieval and termination of roles which the component can adopt, for access of component attributes, and for invocations of component operations.

Example: We illustrate the translation at our p2p example. For now, we do not show the process of the component *Peer* since we so far just discussed its **self** channel and no behavior. The component process for a *Peer* is shown Sec. 5.2.1.5. However, Fig. 5.1 depicts the used-defined data type for role-to-component communication in our p2p example. Since we have just one component type *Peer*, there is only one user-defined data type `PeerOperation` which restricts the communication of all role types with this component type. The most important feature of the data type `PeerOperation` is the enumeration type in line 2–10. It lists all types of requests which can be sent to a *Peer*: there are constants for the request for role creation, retrieval and termination of the role types *Requester*, *Router*, and *Provider*, for access to the component attributes *hasFile* and *content*, for access to the component association *neighbor*, and for invoking the operation *printFile*. The fields `optype`, `parameters`, and `answer` will be used to built the actual request with concrete values (more details on that can be found in Sec. 5.2.2.2).

```
1  typedef PeerOperation {
2    mtype {
3      CREATE_REQUESTER, GET_REQUESTER, QUIT_REQUESTER, ...
4      CREATE_ROUTER, GET_ROUTER, QUIT_ROUTER, ...
5      CREATE_PROVIDER, GET_PROVIDER, QUIT_PROVIDER, ...
6      GET_HASFILE, SET_HASFILE,
7      GET_CONTENT, SET_CONTENT,
8      GET_NEIGHBOR,
9      OP_PRINTFILE,
10   };
11   mtype optype;
12   chan parameters;
13   chan answer;
14 }
```

Figure 5.1: Data type for role-to-component communication for the p2p example in PROMELA

5.2.1.2 Role-to-Role Communication

In HELENA, roles additionally interact with other roles by exchanging directed messages on input queues. Thus, each PROMELA role process relies on a dedicated (possibly asynchronous) channel **self** to model its input queue. Since channels are global in PROMELA, but input queues are local in HELENA, special care has to be taken that the channel **self** of the current role process is only available to PROMELA processes of roles which are allowed to communicate with the current role in HELENA. Additionally, each PROMELA role process relies on the synchronous channel **owner** to communicate with its owning component as described in the previous section.

The type of messages which can be sent to this channel is restricted by a list of PROMELA types. The list requires the messages to consist of a constant representing the message type declared in the HELENA ensemble structure, a fixed number of channel references representing the role parameters of the message type, and a fixed number of integers representing the data parameters of the message type. Since all messages might differ in the number of parameters, the channel references and integer values might later on be filled up with dummy values.

Example: We illustrate the translation at our p2p example again. Similarly to role-to-component communication, we do not show any role process here since we so far just discussed its **self** channel (and **owner** channel in the previous subsection), but did not yet discuss its behavior. An example for a role process is shown in Sec. 5.2.1.6. However, Fig. 5.2 depicts the enumeration type representing all message types for role-to-role communication in our p2p example. It used for all communications between roles, independently between which particular role types. It lists all types of messages which can be sent between roles. These constants can directly be determined from the set of all message types in our p2p example.

```
1  mtype { reqAddr, sndAddr, reqFile, sndFile }
```

Figure 5.2: Enumeration type for role-to-role communication for the p2p example in PROMELA

The channel **self** of any role in the p2p example then requires that only messages of the signature `{ mtype,chan,int }` are sent to it. First, a constant representing the message type to be sent must be given. Afterwards, exactly one channel representing a role instance parameter and exactly one integer representing a data parameter have to be transmitted. The number of role instance parameters is restricted to exactly one since this is the maximal number of role instance parameters in message types in the ensemble structure of the p2p example (and similarly for data parameters). If a message type does not declare any role instance parameters (or similarly for data parameters), like the message *sndFile*, this will be filled up by dummy parameters later on.

5.2.1.3 Data Storage on Components

In HELENA, a component stores data in component attributes and links to other components in associations. Both are reflected in the PROMELA component process by parameters with corresponding type. Thus, they are only visible for the current process instance, similarly to attributes and associations of components in HELENA.

To access component attributes and associations from an adopted role in PROMELA, an appropriate request can be sent via the aforementioned channel **owner** of the PROMELA role process corresponding to the channel **self** of the PROMELA process for the owning component. Internally, the component process accesses its attributes by retrieving the values of its parameters or by assigning new values to its parameters depending on the issued request.

We show an example for data storage on components in Sec. 5.2.1.5.

5.2.1.4 Data Storage on Roles

In HELENA, a role stores data either permanently in the attributes of its owning component or volatile in its own attributes. Role attributes are reflected in the PROMELA role process by local variables with corresponding type (component attributes are reflected in the corresponding PROMELA component process). Thus, they are only visible for the current process instance, similarly to attributes and associations of components. However, we opted for local variables (instead of parameters as for components) since when we create a new role instance in HELENA, we do not explicitly set the role attributes in the **create**-action. In the PROMELA translation, creating a role instance will correspond to spawning a new role process such that we do not want to have any parameters for that process.

To access role attributes, the set of actions in HELENA is extended by (getters and) setters. They are realized in PROMELA by (access of and) simple assignments to the local variables for role attributes.

We show an example for data storage on roles in Sec. 5.2.1.6.

5.2.1.5 Behavior of Components

On the behavioral side, a component must be able to react to requests and invocations from roles. Thus, the PROMELA component process implements a **do**-loop to wait for requests from its roles on the **self** channel. Depending on the request, it runs some internal computation and sends back a reply. To later on preserve the evolution of the translated if-then-else construct of a role as one indivisible sequence of actions, it is important that the reception of the request from the role and the reaction to it are executed as an indivisible sequence of actions. By introducing an **atomic**-block embracing the reception of the request and the reaction these two steps cannot be interrupted by other processes. Details on the translation and preservation of the semantics of the if-then-else construct will be given in the next section.

The internal computations of a component differ depending on the request from the role. We will walk through all types of requests a role process can send to a component process. Let us first consider role creation which is expressed in HELENA by the action $X \leftarrow \textbf{create}(rt, C)$. Fig. 5.3 depicts the sequence of actions in PROMELA representing role creation. First, the role process which wants the component process C to create another role process X sends an appropriate request to the component process C. The request contains the type rt of the role process to be spawned. The component process C spawns a new process (representing the role) with the **run**-command of PROMELA. To the newly spawned process, it hands over a reference to the channel **self** variable and a reference to a special channel `rtchan` variable. Afterwards, it sends the reference to the special channel `rtchan` variable back to the role process requesting the role creation which stores the reference to the channel `rtchan` in its local variable X.

By this sequence of actions, the component process takes care to initialize the **owner** channel variable and the **self** channel variable of the newly created role process as shown in Fig. 5.4. In the **run**-command, the component process hands over the reference its own **self** channel variable as first parameter. It is stored in the **owner** channel variable of the newly created role process and can then be used for role-to-component communication. Furthermore, the component process hands over the reference to the special channel `rtchan` as second parameter. It is stored in the **self** channel variable of the newly created role process. By sending the reference to the very same channel `rtchan` variable also to the role process issuing the role creation, the two role processes

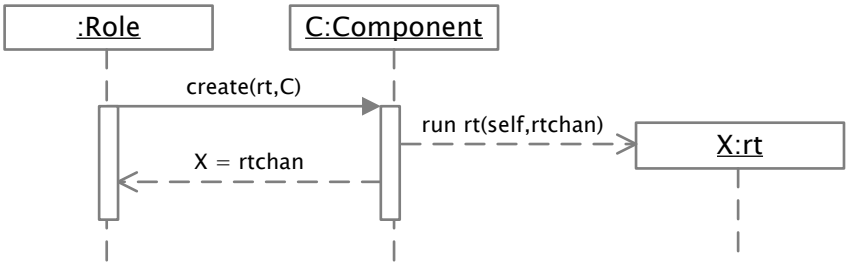


Figure 5.3: Interactions between component process and role processes during creation of a new process `X` for the role type `rt`

can later on communicate via this channel. However, also the component process stores the channel `rtchan` variable such that whenever another role process requests a reference to the newly created role process again (i.e., a request for role retrieval was sent), the component process sends back the stored channel variable as a reference to the role process.

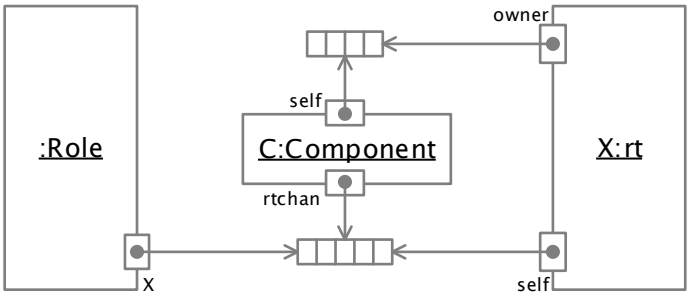


Figure 5.4: Shared channels between a component process and role processes during creation of a new process `X` for the role type `rt`

To retrieve or change the value of component attributes or component associations, the component process evaluates the request from the role process to determine the requested attribute or association and possibly the value to be set. Afterwards, it accesses the parameter corresponding to the attribute or association and either sends the value to the requesting role process or assigns the new value.

Similarly for operation calls, the component process evaluates the request from the role process to determine which operation was invoked. However, the effect of the operation is not specified in Helena. Therefore, in the Promela translation, the operation call does not have any other effect than being processed by the component process. Any effect of the operation must be specified in Promela by hand.

Example: Let us illustrate the complete structure of the Promela process for the component type *Peer* in our p2p example. Fig. 5.5 shows the corresponding process type in Promela.

Its parameters reflect the component attributes *hasFile* and *content* as well as the component association *neighbor* as explained in Sec. 5.2.1.3. Furthermore, the last parameter is the **self** channel which is used for role-to-component communication as explained in Sec. 5.2.1.1.

Before starting the actual behavior of the component process, a dedicated channel is declared for each role type the component can adopt, e.g., line 2 declares the channel `requester` for the role type *Requester*. The capacity of the channel in Promela is initialized with the capacity of the message of the role type in Helena, i.e., 2 for the

```
1  proctype Peer(bool hasFile; int content; chan neighbor; chan self) {
2    chan requester = [2] of { mtype,chan,int };
3    ...
4
5    PeerOperation op;
6    do
7    ::atomic {
8      self?op ->
9        if
10       ::op.optype == GET_HASFILE -> op.answer!hasFile
11       ::op.optype == SET_HASFILE -> op.parameters?hasFile
12       ...
13       ::op.optype == CREATE_REQUESTER ->
14         ...
15         run Requester(self, requester);
16         op.answer!requester
17       ...
18   }
19   od
20 }
```

Figure 5.5: Excerpt of the component process for the component type *Peer* for the p2p example in PROMELA

role type *Requester* in line 2. Furthermore, the channel can only receive elements which have the form `{mtype,chan,int}` (cf. line 2) corresponding to a message name, a single role instance parameter and a single data parameter of type **int**. If the component is later on advised to create a new instance for the role type *Requester*, it hands this channel over to the newly created role as its **self** channel (providing role-to-role communication as described before). If the component is later on requested to retrieve a reference to the very same role instance, it can then send back this channel as reference.

The behavior of the component process is basically a **do**-loop (line 6–19) which continuously waits for a request from a role on its **self** channel (line 8). To decide which request was sent, a large **if**-statement provides branches for every possible request (line 9–17). Depending on the request, the component reacts differently. For example, if the value of the component attribute *hasFile* was requested (`op.optype == GET_HASFILE` in line 10), the component process answers on the channel `op.answer` with the value of the corresponding parameter `hasFile` (`op.answer!hasFile` in line 10). Similarly, if the value of the component attribute *hasFile* was requested to be set (`op.optype == SET_HASFILE` in line 11), the component process waits on the channel `op.parameters` for the new value and assigns it to the corresponding parameter `hasFile` (`op.parameters?hasFile` in line 11). Another example is role creation in line 13–16. The component process spawns a new process for the role where it hands over its own **self** channel as the role's owner and the channel `requester` as the role's **self** channel (line 15). Afterwards, it sends back the reference (here `requester`) to the newly created role via the channel `op.answer` (line 16).

5.2.1.6 Behavior of Roles

On the behavioral side, the PROMELA process for a role must reflect the corresponding role behavior declaration of the HELENA ensemble specification. We translate each process construct and action of the role behavior declaration:

(1) Termination **quit** is translated to two steps in PROMELA. Similarly to the semantic rule of **quit** in Fig. 3.3 on page 57, the role process first requests that the component process quits playing the role. Afterwards, it stops execution by the statement **false**.

(2) Action prefix is translated to sequential composition. Thereby, role creation is expressed by issuing an appropriate request to the owning component process which spawns the new role process and sends back the **self** channel of the newly created role process. Role retrieval also issues an appropriate request to the owning component process which sends back the stored **self** channel of the requested role process (as explained in the previous subsection). Sending and receiving messages is mapped to message exchange on the **self** channel of role processes. Operation calls and access to component attributes issue appropriate requests to the owning component process. Access to role attributes is given by access and assignments to the local variables of the role process.

(3) Nondeterministic choice is translated to the **if**-construct in PROMELA. Similarly to nondeterministic choice in HELENA, the **if**-construct in PROMELA allows to nondeterministically choose between the branches whose first statement is currently executable. However, special care has to be taken since some process constructs and actions in HELENA need additional preparation steps in PROMELA such that they are not expressed by a single PROMELA statement. This leads to a different set of executable branches if in HELENA the action is not executable, but in PROMELA the first preparation steps are executable (and therefore the branch can be selected for execution) while the final steps are not. As explained in Sec. 5.1, process invocation is not allowed as a branch of nondeterministic choice and the **create-** and **get**-action are only allowed as first action of a branch only if the two actions are guaranteed to be executable.

(4) The if-then-else construct is also translated to the **if**-construct in PROMELA where the first statement is the translated guard and the following statements are obtained from the translated process expression. The guard and the first (possibly-nested) action are executed as one indivisible sequence by enclosing them into the **atomic**-block of PROMELA. Since in HELENA the evaluation of the guard and the executability of the first action determines the executability of the whole if-then-else construct, but in PROMELA only the first statement of each branch determines the executability of the whole **if**-construct (even if employing the **atomic**-block), special care has to be taken. If in HELENA the first action is not executable, the whole if-then-else construct will not evolve. In PROMELA, one branch has already been selected before checking the executability of the first action. Thus, as explained in Sec. 5.1, process invocation is not allowed as a branch of the if-then-else construct, the **create-** and **get**-action are only allowed if preceded by a corresponding plays query, sending a message is only allowed if the capacity of the message queue of the receiving role is not exceeded, and receiving a message is not allowed at all.

(5) In HELENA, we can furthermore invoke arbitrary process declarations from a fixed set of process declarations for each role behavior declaration. We do not translate these auxiliary process declarations to self-contained processes in PROMELA. We rather inline them into the calling role behavior adhering to the following procedure:

- When reaching the invocation of a certain process for the first time during the translation from HELENA to PROMELA, we inline the translation of the whole process into the translation of the role behavior declaration. Additionally, we prefix the translation of the process with a particular state label denoting the invoked process.
- When reaching the invocation of the same process once again during the translation of the role behavior declaration (and its invoked processes), we translate the repeated invocation to a **goto** action which jumps to the state label we introduced for this process before.

Example: Let us finally illustrate the complete structure of the process for the role type *Requester* in our p2p example. Fig. 5.6 shows the corresponding process type in PROMELA. Its parameters are only its **owner** channel which is used for role-to-component communication as explained in Sec. 5.2.1.1 and its **self** channel which is used for role-to-role communication as explained in Sec. 5.2.1.2. The attribute *hasFile* of the role type *Requester* is represented by a local variable `roleAttr_hasFile` in line 2. Local variables for all created role instances and formal parameters are listed afterwards, we only show the variables for the created router in line 4 and for the formal parameter *prov* of receiving the message *sndAddr* in line 5.

```
1  proctype Requester(chan owner, self) {
2    bool roleAttr_hasFile;
3
4    chan router;
5    chan prov;
6    ...
7
8    PeerOperation op;
9    op.optype = CREATE_ROUTER;
10   chan answer = [0] of { chan };
11   op.answer = answer;
12   neighbor!op;
13   answer?router;
14
15   router!reqAddr,self,1;
16
17   self?sndAddr,prov,1;
18   ...
19 }
```

Figure 5.6: Excerpt of the role process for the role type *Requester* for the p2p example in PROMELA

The behavior of the role process is basically the translation of the role behavior declaration for a *Requester* (cf. Fig. 2.6 on page 31). The first block (line 8–13) is the translation of the create action for a *Router*. The block basically creates a request for role creation in line 8–11 and sends the requests to the neighboring component of its owner in line 12 (the retrieval of the neighbor from the owning component is not shown here, but is analogous to the role creation request). The neighboring component sends a reference to the **self** channel of the newly created role which is stored in the local variable `router` (line 13). The next part (line 15) is the translation of sending the message *reqAddr*(**self**) to the newly created router. The content consists of the type of the message `reqAddr`, the reference to the requester itself by the channel variable `self` and a final dummy parameter since the message does not have any data parameters.

The last shown part (line 17) is the translation of receiving the message *sndAddr* with the parameter *prov*. The role process waits for a message on its **self** channel. The type of the message has to match the value `sndAddr` and the sent channel reference representing the provider is stored in the local variable `prov`. The final parameter is again a dummy parameter since the message does not have any data parameters.

To meet the discussed assumptions for if-then-else construct, the role behavior of the role type *Router* (cf. Fig. 2.7 on page 31) needs adaptation before it can be translated to PROMELA. All process invocations except the recursive role behavior invocation must be removed since they occur as branches in if-then-else constructs. The resulting role behavior is shown in Fig. 5.7.

```
roleBehavior Router = ?reqAddr(req:Requester)() .
                      if (owner.hasFile) {
                          provider←create(Provider, owner) .
                          req!sndAddr(provider)() .
                          quit
                      }
                      else
                          if ( !plays(Router, owner.neighbor) ) {
                              router←create(Router, owner.neighbor) .
                              router!reqAddr(req)() .
                              Router
                          else { quit }
```

Figure 5.7: Role behavior of a *Router* for the p2p example (adapted)

5.2.1.7 Multiplicities of Role Instances

Lastly, in HELENA, an ensemble structure determines for each role the minimal and maximal number of instances per participating role type. These multiplicities have to be respected throughout the execution of the ensemble. Therefore, we include three global variables in PROMELA per role type: Two global variables are used to represent the minimal and maximal number of instances per role type. The third global variable counts the current number of instances per role type in the evolving ensemble.

The two variables for minimal and maximal number are final and should not be changed throughout the lifetime of an ensemble. The variable for the current number, however, is accessed and changed whenever a new process instance for that role type is created or quit since the process instance finished its translated behavior. The owning component process is responsible for the creation and the termination of role instances and therefore for the update of the current number of process instances per role type. Before a new role process instance is created in PROMELA, the component process checks whether the current number of process instances for that role type is smaller than the maximal allowed number of instances. If the creation action is allowed, the role process instance will be created and the current number of process instances for that role type is increased by one; otherwise the creation action is blocked. Similarly, the component process checks whether the current number of process instances for a certain role type is bigger than the minimal allowed number of instances before it quits playing a role instance. If the quit action is allowed, the component process will quit playing the

role and the current number of process instances of that role type is decreased by one; otherwise the quit action is blocked.

5.2.2 Translation Functions

After the informal overview about the translation of the features of HELENA, we introduce formal translation functions for all of them. We assume given a full HELENA ensemble specification $EnsSpec = (\Sigma, behaviors)$ over CT. The translation proceeds in five steps:

- For each role type in Σ, we introduce three global variables to reflect the minimal, maximal and current number of process instances of that type (cf. Sec. 5.2.1.7).
- For each component type in CT in HELENA, we introduce a user-defined data type in PROMELA to represent all requests which can be invoked on a component by its adopted roles (cf. Sec. 5.2.1.1).
- For each component type in CT in HELENA, we create a process type in PROMELA which is able to adopt roles and repeatedly handle requests from its adopted roles (cf. Sec. 5.2.1.5 together with Sec. 5.2.1.1 and Sec. 5.2.1.3).
- We introduce a user-defined data type in PROMELA to represent all messages which can be exchanged between any roles in the HELENA ensemble specification (cf. Sec. 5.2.1.2).
- For each role type and its corresponding role behavior declaration in HELENA, we create a process type in PROMELA which reflects the execution of the role behavior where any invoked processes are inlined (cf. Sec. 5.2.1.6 together with Sec. 5.2.1.1, Sec. 5.2.1.2, and Sec. 5.2.1.4).

In the following, we will present translation functions for each of the steps. Everything notated in normal or bold font is pure PROMELA code, everything notated in italic font has to be evaluated to get PROMELA code, especially functions prefixed with a $-sign are fixed names in PROMELA which are not further specified here. We furthermore use the notation $\forall e \,.\, expr(e)$. It means that the expression $expr(e)$ is evaluated for all elements e identified by the quantifier.

5.2.2.1 Multiplicities of Role Instances

The multiplicities of role types in HELENA and the current number of corresponding role instances is reflected in PROMELA by three global variables. These variables are created by the function $trans_{\text{mult}}$ shown in Fig. 5.8 for each role type rt in a HELENA ensemble specification $EnsSpec = (\Sigma, behaviors)$ with $\Sigma = (nm, roletypes, roleconstraints)$.

$$
\begin{aligned}
&trans_{\text{mult}}(EnsSpec) = \\
&\quad \forall rt \in roletypes(\Sigma) \,.\, \textbf{int}\ \$min(rt) = min(roleconstraints(rt)); \\
&\quad \forall rt \in roletypes(\Sigma) \,.\, \textbf{int}\ \$max(rt) = max(roleconstraints(rt)); \\
&\quad \forall rt \in roletypes(\Sigma) \,.\, \textbf{int}\ \$current(rt) = 0;
\end{aligned}
$$

Figure 5.8: PROMELA translation of multiplicities

To recap our notation, $\forall rt \in roletypes(\Sigma) \,.\, \textbf{int}\ \$min(rt) = min(roleconstraints(rt));$ means that for each role rt in the ensemble structure Σ, a new global variable (whose name expressed by the function $\$min(rt)$) of type **int** is created. It is meant to store the

(final) minimal number of instances allowed for the role type rt and is thus, it is initialized by this minimal number according to the role constraints of the ensemble structure. Similarly, the notation $\forall rt \in roletypes(\Sigma)$. **int** $\$max(rt) = max(roleconstraints(rt))$; creates a second variable (expressed by the function $\$max(rt)$) to store the (final) maximal number of instances allowed for the role type rt. Thus, it is initialized by the maximal number of allowed instances according to the role constraints of the ensemble structure. A third variable per role type rt (expressed by the function $\$current(rt)$) stores the (non-final) current number of instances of the role type rt and is initialized by 0.

5.2.2.2 Role-to-Component Communication Facilities

To reflect all requests in PROMELA which roles can sent to a component of a certain type[3], we declare a user-defined data type per component type. The function $trans_{\text{ops}}$ creates this user-defined data type per component type $ct = (ctnm, ctattrs, ctassocs, ctops)$ and is shown in Fig. 5.9. Thereby, the set of all role types $rt = (rtnm, rtcomptypes, rtattrs, rtmsgs_{out}, rtmsgs_{in})$ in a HELENA ensemble specification $EnsSpec = (\Sigma, behaviors)$ with $\Sigma = (nm, roletypes, roleconstraints)$ which can be adopted by the component type ct is given by:

$$roletypes(ct, EnsSpec) = \{rt \mid \exists rt \in roletypes(\Sigma) \;.\; ct \in rtcomptypes(rt)\}$$

$trans_{\text{ops}}(ct, EnsSpec) =$

typedef $\$op(ct)$ {
  mtype {
    $\forall attr \in ctattrs(ct)$. $\$getter(attr)$,
    $\forall attr \in ctattrs(ct)$. $\$setter(attr)$,
    $\forall assoc \in ctassocs(ct)$. $\$getter(assoc)$,
    $\forall op \in ctops(ct)$. $opnm(op)$,
    $\forall rt \in roletypes(ct, EnsSpec)$. $\$create(rt)$,
    $\forall rt \in roletypes(ct, EnsSpec)$. $\$get(rt)$,
    $\forall rt \in roletypes(ct, EnsSpec)$. $\$quit(rt)$,
    $\forall rt \in roletypes(ct, EnsSpec)$. $\$playsreq(rt)$,
  };
  mtype optype;
  chan parameters;
  chan answer;
}

Figure 5.9: PROMELA translation of component-to-role communication facilities

The name of the user-defined data type, called "**typedef**", is given by the function $\$op(ct)$. The function is not further specified here since its actual value is not important here. The data type declares three fields which are listed at the end of Fig. 5.9:

The first field "optype" determines the type of the request. Its values are given by the enumeration type, called "**mtype**", listed at the beginning of the data type definition. There are constants for getting or setting the value of a component attribute, for getting the value of a component association, for invoking a component operation, creating, retrieving or quitting to play a certain role with the current component itself as owner, or for determining whether the component already plays a certain role. The constants are represented by functions, e.g., $\$getter(attr)$, which are again not specified here. Note

[3]Note that we assumed that a role type can only be adopted by a single component type.

that again the notation $\forall e$. $expr(e)$ means that the expression $expr(e)$ is evaluated for all elements e identified by the quantifier as explained before.

The second field "parameters" introduces a channel via which parameters for the request can be sent. For example, if the value of a component attribute should be set, this channel is used to transfer the value to the component.

The third field "answer" again introduces a channel. Via this channel any answer to the request can be sent. For example, if the value of a component attribute should be retrieved, this channel is used to transfer the value to the requesting role process.

In summary, a concrete instance of this user-defined data type represents a request from a role process to a component process. The request itself is identified by the field "optype", the channel through which any parameters for the request can be sent is given in the field "parameters", and the channel through which an answer can be sent from the component process to the role process is given in the field "answer".

5.2.2.3 Behavior of Components

Component types themselves are reflected by process type declarations in PROMELA. These process type declarations manage the currently adopted role types and can handle requests from roles. The function $trans_{\text{comp}}$ creates such a process type declaration for a component type $ct = (ctnm, ctattrs, ctassocs, ctops)$ in a HELENA ensemble specification $EnsSpec = (\Sigma, behaviors)$ with $\Sigma = (nm, roletypes, roleconstraints)$ and is shown in Fig. 5.10.

The component type is represented by a new process type declaration (with the same name) with parameters in PROMELA. We explain the parameters beginning from the last one: Firstly, the parameter **self** is a channel which is used for communication from a role process to this component process. Every request for component attributes, operation calls, or role creation, retrieval or termination or plays queries is sent via this channel. Secondly, for each association declared by the component type, a parameter is added with the same name (expressed by the function $\$name(assoc)$). The parameter is typed as channel since references to other components are represented via their **self** channel. Lastly, for each attribute declared by the component type, a parameter with the name of the attribute (expressed by the function $\$name(attr)$) and its type (expressed by the function $\$type(attr)$) is added to the process type declaration. Note that we can only support the PROMELA data types `byte`, `short`, `int`, and `bool`.

During the execution of the process for the component type, we have to store references to all roles which the component currently plays. Therefore, for each role type the component can adopt, we define a boolean local variable $\$plays(rt)$ in the process type declaration which stores whether the component currently plays the corresponding role. A boolean variable is enough since in HELENA each component can adopt a role only once per ensemble and we assumed that we only check one ensemble instance in each PROMELA run. Furthermore, we initialize a new channel for each role type the component can adopt. This channel will be used as the **self** channel of the corresponding role instance as soon as it was created.

Afterwards, we add a state label $\$startlabel(ct)$: **true** to mark the point in the process where initialization was finished.

```
trans_comp(ct, EnsSpec) =
  proctype ct(∀attr  ∈ ctattrs(ct)   . $type(attr) $name(attr),
              ∀assoc ∈ ctassocs(ct) . chan       $name(assoc),
              chan self) {
    ∀rt ∈ roletypes(ct, EnsSpec) . bool $plays(rt) = false;
    ∀rt ∈ roletypes(ct, EnsSpec) . chan $chan(rt) = [cap(roleconstraints(rt))] of {Msg};

    $startlabel(ct) : true;

    $op(ct) op;

    do
    ::atomic {
      self?op ->
      if
      ∀attr ∈ ctattrs(ct) . ::op.optype == $getter(attr) -> op.answer!$name(attr)
      ∀attr ∈ ctattrs(ct) . ::op.optype == $setter(attr) -> op.parameters?$name(attr)
      ∀assoc ∈ ctassocs(ct) . ::op.optype == $getter(assoc) -> op.answer!$name(assoc)
      ∀op ∈ ctops(ct) . ::op.optype == opnm(op) -> \\ add intended behavior here
      ∀rt ∈ roletypes(ct, EnsSpec) . ::op.optype == $create(rt) ->
          if
          ::!$plays(rt) && $current(rt) < $max(rt)- >
              run rt(self, $chan(rt));
              $plays(rt)=true;
              $current(rt)++;
              op.answer!$chan(rt)
          fi
      ∀rt ∈ roletypes(ct, EnsSpec) . ::op.optype == $get(rt)- >
          if
          ::$plays(rt) -> op.answer!$chan(rt)
          fi
      ∀rt ∈ roletypes(ct, EnsSpec) . ::op.optype == $quit(rt) ->
          if
          ::$plays(rt) && $current(rt) < $min(rt) ->
              $plays(rt)=false;
              $current(rt)--
          fi
      ∀rt ∈ roletypes(ct, EnsSpec) . ::op.optype == $playsreq(rt) ->
          op.answer!$plays(rt)
      fi
      }
    od
  }
```

Figure 5.10: PROMELA translation of a component type

Finally, the main behavior of the process of a component is given by a **do**-loop. The loop is responsible for continuously waiting for requests from role processes and for appropriately reacting to the request. Thereby the whole block from waiting for the request until the final reaction has to be executed as one indivisible sequence of actions caused by the keyword **atomic** in PROMELA. Atomicity is needed to reflect the semantics of the if-then-else construct in role behaviors. In HELENA, the if-then-else construct evaluates the guard and executes the first action of the selected branch in one single step. To guarantee this atomicity even if the first action is a request to the owning component (e.g., a **create**-action), the atomic execution of the reception of the request, the internal computation and the answer from the component is established by

the **atomic**-block. In this **atomic**-block, the component waits for the guard "**self**?op" representing a message on the **self** channel of the component. The content of the message is the request from the role process and is stored in the local variable "op" which is typed by the user-defined data type $\$op(ct)$. The request consists of three parts as explained in Sec. 5.2.2.2: the type of the request given by the field "optype" of the local variable "op", a channel "parameters" allowing to send parameters for the request from the requesting role process to the component process, and a channel "answer" allowing to send back a reply to the requesting role process. Depending on the type of the received request, we decide how to react in the following **if**-statement.

- The first line of the loop represents the request of the value of an attribute. For each attribute of the component type, an option of the **if**-statement is created. It checks which attribute was requested by "op.optype == $\$getter(attr)$" where $\$getter(attr)$ determines the type of the request as in Sec. 5.2.2.2. Afterwards, the value of the attribute is sent by the expression "op.answer!$\$name(attr)$ where $\$name(attr)$ is the parameter representing the attribute in the component process.
- Similarly, the request to set the value of an attribute is handled in the second line of the loop. However, this time the component waits for the new value with the expression "op.parameters?$\$name(attr)$" where $\$name(attr)$ is the parameter representing the attribute in the process type declaration and is automatically set as soon as a value is sent via the channel "op.parameters".
- For the retrieval of associations to other components in the third line of the loop, we proceed exactly like for the retrieval of attribute values.
- The call of operations is handled in the fourth line of the loop. Since the behavior of the operation is not specified in HELENA, we only add a placeholder for the behavior of the operation to the translated PROMELA file.
- Most interesting is the handling of role instance creation, retrieval and termination which is represented in the following three blocks of the loop. If role instance creation was requested by "op.optype == $\$create(rt)$" for any role type rt, we check in the **if**-statement of the first block that the component does not yet play the role by the expression !$\$plays(rt)$ and that the number of current instances of that role type did not yet reach the maximum number of allowed instances by $\$current(rt) < \$max(rt)$. If the check is passed, the following statements are executed: A new process representing the role is started by "**run** rt(**self**, $\$chan(rt)$)" where the **self** channel of the component is passed as the **owner** channel of the newly created role and the channel $\$chan(rt)$ is passed as the **self** channel of the newly created process. The value of the boolean variable $\$plays(rt)$ is set to **true**, the global variable $\$current(rt)$ for the current number of instances of that type is increased, and the channel $\$chan(rt)$ representing the **self** channel of the newly created role instance is sent back to the requesting role process by "op.answer!$\$chan(rt)$" such that the requesting role process can now communicate to the newly created role process. Similarly, role retrieval and role termination are handled, but in these cases we do not need atomicity since no new processes are spawn.
- The last line in the loop takes care to answer requests whether the component currently plays a certain role and is self-explanatory with the previous explanation of role creation.

5.2.2.4 Role-to-Role Communication Facilities

To reflect all messages which can be sent between roles of a HELENA ensemble specification, we declare a an enumeration type which lists constants for all messages. The function $trans_{\text{msgs}}$ creates this enumeration type for all message types of an ensemble specification and is shown in Fig. 5.11. Thereby, the set of all message types of a HELENALIGHT ensemble specification $EnsSpec = (\Sigma, behaviors)$ with $\Sigma = (nm, roletypes, roleconstraints)$ is given by:

$$msgs(EnsSpec) = \{msg \mid \exists rt \in roletypes(\Sigma). \; msg \in rtmsgs_{out}(rt) \vee msg \in rtmsgs_{in}(rt)\}.$$

The enumeration type consists of constants for every message type which is declared as outgoing or incoming message of a role type in the underlying ensemble specification.

$$trans_{\text{msgs}}(EnsSpec) = \quad \textbf{mtype} \; \{ \quad \forall msg \in msgs(EnsSpec) \; . \; msgnm(msg) \quad \}$$

Figure 5.11: PROMELA translation of role-to-role communication facilities

5.2.2.5 Behavior of Roles

Role types themselves are reflected by process type declarations in PROMELA. These process type declarations are responsible to execute the behavior prescribed by the corresponding role behavior declaration in HELENA. They thereby issue requests on their owning component using their role-to-component communication facilities and exchange messages with other roles using their role-to-role communication facilities.

Process Type Declaration: The function $trans_{\text{role}}$ in Fig. 5.12 creates such a process type declaration for a role type $rt = (rtnm, rtcomptypes, rtattrs, rtmsgs_{out}, rtmsgs_{in})$ with the corresponding role behavior declaration **roleBehavior** $rt = P$ in a HELENA ensemble specification $EnsSpec = (\Sigma, behaviors)$. As we mentioned at the beginning of this section, we assume that each role type can only be adopted by one component type.

$trans_{\text{role}}(rt, EnsSpec) =$
proctype rt(**chan owner**, **self**) {
$\forall attr \in rtattrs(rt)$. \$$type(attr)$ \$$name(attr)$;
$\forall inst \in roleinsts(rt)$. **chan** \$$name(inst)$;
$\forall param \in dataparams(rt)$. \$$type(param)$ \$$name(param)$;
$\forall re \in opereturnvals(rt)$. \$$type(re)$ \$$name(re)$;

\$$startlabel(rt)$: **true**;

$trans_{\text{proc}}(rt, EnsSpec, P)$

\$$endlabel(rt)$: **false**;
}

Figure 5.12: PROMELA translation of a role type

The role type is represented in PROMELA by a new process type declaration (with the same name) with parameters. The parameter **owner** is a channel which is used for communication from the role process to its owning component process. Every request for component attributes, operation calls, or role creation, retrieval or termination is sent via this channel. The parameter **self** is another channel which is used as input channel of the role process. Every message addressed to the role process is sent via this channel.

The first statements in the process type declaration for the role type declare local variables for all role attributes. For each role attribute *attr* of the role type *rt*, a new uninitialized local variable with the name of the attribute (expressed by the function $\$name(attr)$) and its type (expressed by the function $\$type(attr)$) is added. Note again that we can only support the PROMELA data types `byte`, `short`, `int`, and `bool`. Afterwards, local variables for all role instances which are created in the role behavior or received via message exchange are created. Similarly, local variables for all data parameters received via message exchanged or created as return values of operations calls are added.

After the declaration of all local variables, we add a state label $\$startlabel(rt) : \mathbf{true}$ to mark the point in the process where initialization was finished. Finally, the process expression P of the role behavior declaration for that role type is translated as the main behavior of the process which is expressed by the function call $trans_{\text{proc}}(rt, EnsSpec, P)$. The process is terminated by a dedicated end state label $\$endlabel(rt) : \mathbf{true}$.

Process Expressions: The function $trans_{\text{proc}}$ itself is inductively defined over the structure of process expressions and is shown in Fig. 5.13.

To translate termination with **quit**, we follow the formal semantics in Fig. 3.3 on page 57. The role process first requests that its owning component process quits playing it. To this end, a new local variable "op" is created which is used to compose the request to the owning component. The local variable is typed with the user-defined data type for role-to-component communication (cf. Sec. 5.2.2.2). Thereby, we have to assume that the role type can only be adopted by a single component type (denoted by $rtcomptypes(rt)$) to uniquely determine the user-defined data type of "op". Afterwards, the field "optype" of the local variable "op" is set to $\$quit(rt)$ to express that the termination of the adoption of the role is requested. With this information the request is ready to be sent to the owning component process by the statement **owner**!op. Finally, the role process stops execution jumping to the end label **goto** $\$endlabel(rt)$.

Action prefix $a.P$ is simply translated to sequential composition in PROMELA. We sequentially compose the translation of the action a with the translation of the remaining process expression P. The translation of actions is denoted by the function $trans_{\text{act}}$ (cf. Fig. 5.16) and will be described later on.

Nondeterministic choice is translated to nondeterministic choice with the **if**-construct in PROMELA. Each branch is thereby the direct translation of one option of nondeterministic choice. To preserve the semantics of HELENA that a branch is selected based on the executability of the first action of each branch, it is essential that the assumptions explained in Sec. 5.1 are respected: Firstly, process invocation is not allowed as a branch of nondeterministic choice. Secondly, the **create**- and **get**-action are only allowed as first action of a branch if they are executable.

The if-then-else construct is translated to nondeterministic choice with the **if**-construct in PROMELA as well. To guarantee that the evaluation of the guard and the execution of the first action of the selected branch are performed without interrup-

$trans_{\text{proc}}(rt, EnsSpec, \textbf{quit}) = \$op(rtcomptypes(rt))$ op;
op.optype = $\$quit(rt)$;
owner!op;
goto $\$endlabel(rt)$

$trans_{\text{proc}}(rt, EnsSpec, a.P) = trans_{\text{act}}(rt, EnsSpec, a);$
$trans_{\text{proc}}(rt, EnsSpec, P)$

$trans_{\text{proc}}(rt, EnsSpec, P_1 + P_2) =$ **if**
:: $trans_{\text{proc}}(rt, EnsSpec, P_1)$
:: $trans_{\text{proc}}(rt, EnsSpec, P_2)$
fi

$trans_{\text{proc}}(rt, EnsSpec,$ **if** $(guard)$ $\{P_1\}$ **else** $\{P_2\}) =$ **atomic** {
$trans_{\text{retrieval}}(guard)$
if
:: $trans_{\text{guard}}(guard)$
-> $trans_{\text{proc-first}}(rt, EnsSpec, P_1)$
:: **else**
-> $trans_{\text{proc-first}}(rt, EnsSpec, P_2)$
fi
};
$trans_{\text{proc-remaining}}(rt, EnsSpec, P_1);$
$trans_{\text{proc-remaining}}(rt, EnsSpec, P_2);$

$trans_{\text{proc}}(rt, EnsSpec, N) = \$startlabel(N) : \textbf{true};$
$trans_{\text{proc}}(rt, EnsSpec, Q)$
if N is invoked for the first time and **process** $N = Q \in procdecls(rt)$

$trans_{\text{proc}}(rt, EnsSpec, N) =$ **goto** $\$startlabel(N)$
if process N has already been invoked

Figure 5.13: PROMELA translation of a process expression

tions, we enclose the following translation into an **atomic**-block: If the guard of the if-then-else construct contains plays queries or refers to component attributes, they have to be retrieved before the guard is evaluated. The retrieval is expressed by the function $trans_{\text{retrieval}}$ which is not further specified here since it resembles the translation of operation calls shown in Fig. 5.16. Afterwards, the if-then-else construct is translated to the **if**-construct of PROMELA. The guard of each branch in HELENA is reflected by the first statement of the corresponding branch in PROMELA. The HELENA guard is translated to PROMELA by the function $trans_{\text{guard}}$ which is not further specified here, since it is mainly a direct translation of the boolean expression from HELENA. Afterwards, the first (possibly nested) action of the corresponding process expression is translated before leaving the **atomic**-block. This is expressed by the function $trans_{\text{proc-first}}$ shown in Fig. 5.14.

Intuitively, this function follows the nesting of process constructs until it finally reaches a single action to be executed. This action is translated as we will describe in Fig. 5.16 followed by a jump outside the **atomic**-block where the remaining process expression of this branch is described. Note that also here it is essential that the assumptions explained in Sec. 5.1 and at the beginning of this section are respected to preserve the semantics of HELENA: Firstly, process invocation is not allowed as a branch of the if-then-else construct. Secondly, the **create**- and **get**-action are only allowed as first action of a branch if they are preceded by guards with corresponding

$$
\begin{array}{lcl}
trans_{\text{proc-first}}(rt, EnsSpec, \textbf{quit}) & = & trans_{\text{proc}}(rt, EnsSpec, \textbf{quit}) \\
trans_{\text{proc-first}}(rt, EnsSpec, a.P) & = & trans_{\text{act}}(rt, EnsSpec, a); \\
 & & \textbf{goto}\ \$proclabel(rt) \\
trans_{\text{proc-first}}(rt, EnsSpec, P_1 + P_2) & = & \textbf{if} \\
 & & :: trans_{\text{proc-first}}(rt, EnsSpec, P_1) \\
 & & :: trans_{\text{proc-first}}(rt, EnsSpec, P_2) \\
 & & \textbf{fi} \\
trans_{\text{proc-first}}(rt, EnsSpec, & = & trans_{\text{retrieval}}(guard) \\
\quad \textbf{if}\ (guard)\ \{P_1\} & & \textbf{if} \\
\quad \textbf{else}\ \{P_2\}) & & :: trans_{\text{guard}}(guard) \\
 & & \quad \text{-}\!> trans_{\text{proc-first}}(rt, EnsSpec, P_1) \\
 & & :: \textbf{else} \\
 & & \quad \text{-}\!> trans_{\text{proc-first}}(rt, EnsSpec, P_2) \\
 & & \textbf{fi}
\end{array}
$$

Figure 5.14: PROMELA translation of the first (possibly nested) action of a process expression

plays queries. Thirdly, message reception is not allowed as first action of a branch at all. Afterwards, the else-branch of the if-then-else construct is analogously translated. Finally, the **atomic**-block is closed since the guard and the first action of each branch have been translated now. After the **atomic**-block, we add the translation of the remaining process expressions of both branches such that each branch inside the **atomic**-block can jump to its remaining process expression after having executed its first (possibly nested) action. The translation of the remaining process expression is given by the function $trans_{\text{proc-remaining}}$ and shown in Fig. 5.15.

$$
\begin{array}{lcl}
trans_{\text{proc-remaining}}(rt, EnsSpec, \textbf{quit}) & = & \sqcup \\
trans_{\text{proc-remaining}}(rt, EnsSpec, a.P) & = & \$proclabel(P) : \textbf{true} \\
 & & trans_{\text{proc}}(P) \\
trans_{\text{proc-remaining}}(rt, EnsSpec, P_1 + P_2) & = & trans_{\text{proc-remaining}}(rt, EnsSpec, P_1) \\
 & & trans_{\text{proc-remaining}}(rt, EnsSpec, P_2) \\
trans_{\text{proc-remaining}}(rt, EnsSpec, & = & trans_{\text{proc-remaining}}(rt, EnsSpec, P_1) \\
\quad \textbf{if}\ (guard)\ \{P_1\} & & trans_{\text{proc-remaining}}(rt, EnsSpec, P_2) \\
\quad \textbf{else}\ \{P_2\}) & &
\end{array}
$$

Figure 5.15: PROMELA translation of the remaining process term after having executed the first (possibly nested) action

The idea of the translation of process invocation is to inline the behavior of the process into the behavior of the role. We prefix the inlinement with a particular state label to mark the beginning of the invoked process. Whenever the process gets called in the behavior again, we simply jump back to this state label with a **goto**-statement. Therefore, we distinguish two cases in the translation of process invocation: If the process N is invoked for the first time, we add the dedicated state label (expressed by $\$startlabel(N) : \textbf{true}$) followed by the translation of the process expression Q representing

the behavior of the process N. If the process N has already been invoked, the translation is just a **goto**-statement to the state label $\$startlabel(N)$: **true** representing the beginning of the inlinement of the process N.

Actions: Finally, we describe the translation of actions which is given by the function $trans_{\text{act}}$ in Fig. 5.16. All actions from Def. 2.9 on page 25 are covered. While **create**- and **get**-actions, operation calls, and component attribute setters use the role-to-component communication facilities, sending and receiving messages requires the role-to-role communication facilities. Role attribute setters and state labels can be translated straightforward in the role process itself.

For the translation of role creation with the action $X \leftarrow \textbf{create}(rt, C)$, an appropriate request must be sent to the component C which should adopt the newly created role of type rt. This component will internally spawn a new process for the role and sent back the role's **self** channel as a reference to the newly created role (cf. Sec. 5.2.2.3). To store this local reference, the local variable X of type **chan** was created at the beginning of the process type representing this role type (cf. Fig. 5.12). Afterwards, the request for the component is built by creating a new local variable "op". The local variable is typed with the user-defined data type for role-to-component communication (cf. Sec. 5.2.2.2). Thereby, we again have to assume that a role type can only be adopted by a single component type (denoted by $rtcomptypes(rt)$) to uniquely determine the user-defined data type of "op". Afterwards, the field "optype" of the local variable "op" is set to $\$create(rt)$ to express that the creation of a new role is requested. Furthermore, we have to add a channel "answer" to the request which serves as a callback channel to transmit the reference of the newly created role process from the owning component to the requesting role. The channel "answer" is therefore initialized to transmit messages of type **chan** which is the type of the **self** channel used as the reference. This channel is added in the field "answer" of the request "op". With this information, the request is ready to be sent to the component process which should own the newly created role by the statement C!op. Finally, we wait for the reference of the newly create role process to be sent via the channel "answer" by the statement answer?X and assign the received channel to the local variable X.

Thereby, it is important that the semantic rule for role creation in HELENA (cf. Fig. 3.3 on page 57) prescribes only one step for the role creation and the assignment to the variable X. To reflect that in PROMELA, spawning the new role process from the component process and assigning a value to the local variable X in the requesting role process has to be done in one indivisible step. This has already been taken care of in the behavior of the component process in PROMELA (cf. Fig. 5.10). There, receiving a request from a role up until the appropriate reaction to it was declared as one indivisible sequence. It only remains to include the assignment of the **self** channel to the variable X in this indivisible sequence. Although this assignment is done by the statement answer?X in the role process and not in the component process, atomicity is not lost. If in PROMELA synchronous message exchange is used inside an atomic sequence, control passes from sender to receiver. That means that control passes from the component process with the send statement op.answer!$\$chan(rt)$ to the role process with the receive statement answer?X. Thus, the assignment of the variable X in the role process is directly executed after the indivisible sequence of statements in the component process and therefore we can consider all statements from spawning the new role process until the assignment of the variable X as one atomic step.

$trans_{\text{act}}(rt, EnsSpec, X \leftarrow \textbf{create}(rt, C))$ = \$*op*(*rtcomptypes*(*rt*)) op;
op.optype = \$*create*(*rt*);
chan answer = [0] **of** {**chan**};
op.answer = answer;
C!op;
answer?X

$trans_{\text{act}}(rt, EnsSpec, X \leftarrow \textbf{get}(rt, C))$ = \$*op*(*rtcomptypes*(*rt*)) op;
op.optype = \$*get*(*rt*);
chan answer = [0] **of** {**chan**};
op.answer = answer;
C!op;
answer?X

$trans_{\text{act}}(rt, EnsSpec, Y!msgnm(\overrightarrow{X})(\overrightarrow{e}))$ = $Y!msgnm$,
for $i \in size(\overrightarrow{X})$. $\overrightarrow{X}[i]$,
for $size(\overrightarrow{X}) < i \leq \$maxroleparams(EnsSpec)$. 1,
for $i \in size(\overrightarrow{e})$. $\overrightarrow{e}[i]$,
for $size(\overrightarrow{e}) < i \leq \$maxdataparams(EnsSpec)$. 1

$trans_{\text{act}}(rt, EnsSpec, ?msgnm(\overrightarrow{X}{:}\overrightarrow{rt})(\overrightarrow{x}))$ = **self**?*msgnm*,
for $i \in size(\overrightarrow{X})$. $\overrightarrow{X}[i]$,
for $size(\overrightarrow{X}) < i \leq \$maxroleparams(EnsSpec)$. 1,
for $i \in size(\overrightarrow{x})$. $\overrightarrow{x}[i]$,
for $size(\overrightarrow{x}) < i \leq \$maxdataparams(EnsSpec)$. 1

$trans_{\text{act}}(rt, EnsSpec, \textbf{owner}.opnm(\overrightarrow{e}))$ = \$*op*(*rtcomptypes*(*rt*)) op;
op.optype = *opnm*;
chan parameters = [0] **of** {\$*type*($\overrightarrow{e}$)};
op.parameters = parameters;
owner!op;
parameters!$\overrightarrow{e}$

$trans_{\text{act}}(rt, EnsSpec, \textbf{owner}.attr = e)$ = \$*op*(*rtcomptypes*(*rt*)) op;
op.optype = \$*setter*(*attr*);
chan parameters = [0] **of** {\$*type*(*attr*)};
op.parameters = parameters;
owner!op;
parameters!e

$trans_{\text{act}}(rt, EnsSpec, \textbf{self}.attr = e)$ = \$*name*(*attr*) = e

$trans_{\text{act}}(rt, EnsSpec, label)$ = *label* : **true**

Figure 5.16: PROMELA translation of an action

The translation of the **get**-action for role retrieval is exactly the same as for role creation except that the "optype" of the request is set to \$*get*(*rt*). Internally, the request for role retrieval triggers a different behavior on the owning component than for role creation. Instead of spawning a new role process, the component will just sent back the **self** channel of the requested role (cf. Sec. 5.2.2.3).

For the sake of simplicity, we postpone the explanation of the translation of sending and receiving messages and go on with the explanation of the translation of operation

calls and component attribute setters. For both, the role process must also send requests to its owning component process. Therefore, in both cases, a local variable "op" is created to hold the request as before. For operation call, the field "optype" is set to the name of the operation to be called (expressed by *opnm*; for the component attribute setter, it is set to $\$setter(attr)$. In both cases, we furthermore add a channel "parameters" to the request which will be used to transmit the values of the parameters of operation call or the value of the component attribute to be set to the owning component. With this information, the request is ready to be sent to the owning component process by the statement **owner**!op. Afterwards, the values of the parameters or the attribute are sent to the owning component process by the statement parameters!$\overrightarrow{e}$ or parameters!e.

For sending and receiving messages, we use the role-to-role communication facilities of PROMELA role processes, especially the enumeration type for messages and the input channel **self** of role processes (cf. Sec. 5.2.1.2 and Sec. 5.2.2.4). The translation of sending a message with the action $Y!msgnm(\overrightarrow{X})(\overrightarrow{x})$ sends a message to the receiving role process Y which contains several items. The first item *msgnm* represents the type of the message to be sent and is one of the constants of the aforementioned enumeration type. Afterwards, the values of all role parameters and data parameters are added. We use the notation $\forall i \in size(\overrightarrow{X})\ .\ \overrightarrow{X}[i]$ to denote that we iterate over all role parameters of the list $\overrightarrow{X}$ and add the value of the ith role parameter of the list $\overrightarrow{X}$ to the content list of the message. Afterwards, the list of role parameters is filled up with entries of value 1 to match the maximal length of a role parameter list over all messages in the ensemble specification. Thereby, $\$maxroleparams(EnsSpec)$ denotes the maximal number of role parameters in a message in the ensemble specification *EnsSpec* Similarly, the values of the data parameters given in the list $\overrightarrow{x}$ are added and filled up with entries of value 1. In summary, a message is sent to the role process Y containing the message type as the first entry followed by all role parameters (possibly extended by dummy role parameters to match the maximal length of a role parameter list) and all data parameters (again possibly extended by dummy data parameters).

In contrast to the translation of sending a message, the translation of receiving a message with the action $?msgnm(\overrightarrow{X}{:}\overrightarrow{rt})(\overrightarrow{x}{:}\overrightarrow{dt})$ has to bind all received values to the local variables representing the formal parameters. A message is received on the **self** channel of the role process. The first entry of this message represents the type of the message to be received and is one of the constants of the aforementioned enumeration type. Via pattern matching, this constant will be matched to the received message[4]. Afterwards, all received values are bound to the local variables representing the formal parameters. The local variables have already been declared at the beginning of the process type representing this role type (cf. Fig. 5.12). We use the same notation as before to iterate over the list of parameters. The ith received value of a role parameter is stored in the ith local variable for role parameters (if the list was filled up with entries of value 1, these values are dismissed). Similarly, the ith received value of a data parameter is stored in the ith local variable for data parameters (while fill-up entries are dismissed).

Setting a role attribute with the action **self**.$attr = e$ is translated to a simple assignment in PROMELA. The local variable for the attribute is accessed by $\$name(attr)$ and is assigned with the new value e.

Finally, a state label in HELENA is translated to a state label with the same name in PROMELA. Furthermore, state labels in PROMELA always have to label a certain

[4] If the first message on the channel does not match this constant, message reception blocks until a matching message is at the first position of the channel

statement. To simulate the semantics of labels in HELENA which takes a single step, we use **true** as statement for the state label to also execute a separate step for the state label in PROMELA.

Summary: This concludes the description of the translation functions from HELENA to PROMELA. We showed that we can find a mapping between HELENA and PROMELA and could thus express all HELENA concepts in PROMELA. However, it remains to show that a HELENA ensemble specification and its PROMELA translation are semantically equivalent such that model-checking results from the PROMELA translation can be transferred back to the original HELENA ensemble specification. The formal proof of semantic equivalence is discussed in Chap. 6 for two simplified variants of HELENA and PROMELA in full detail and is informally extended to full HELENA and PROMELA.

5.3 Model-Checking HELENA with Spin

The previous section outlined how to translate a HELENA ensemble specification to PROMELA. This section explains the practical application of Spin to the PROMELA translation. We summarize in Sec. 5.3.1 how a concrete initial ensemble state is established in PROMELA and how HELENA LTL formulae need to be adapted to conform to the PROMELA translation. Afterwards in Sec. 5.3.2, we explain the principles of model-checking with Spin and two different state compression techniques in Spin. Both subsections are illustrated by our p2p example. The full PROMELA specification for model-checking has 512 lines of code (the specification has automatically been generated with the code generator presented in Sec. 8.3) and is listed in Appendix C.2. It can also be found on the attached CD in the project `eu.ascens.helenaText.p2p` in the file `promela-gen/p2p-check.pml`

5.3.1 Preparation of the PROMELA Translation

Establishing an Initial Ensemble State: To prepare the PROMELA translation for model-checking with Spin, we first have to take into account that the semantics of HELENA and therefore satisfaction of LTL formulae is defined relatively to a given initial state σ_{init}. Thus, when model-checking the corresponding PROMELA translation, we have to establish the corresponding initial state $trans_{\text{init}}(\sigma_{init})$ in PROMELA and verify properties relatively to this initial state. We setup the initial state in a dedicated `init`-process. This process is mainly used to initialize the **self** channels of components and to start component processes as well as to start the initial role instances in σ_{init} with the appropriate owning component. Thereby, it is important that component processes are started before role processes since role processes need a reference to their owning component. We will later on explain the `init`-process for the p2p example (cf. Fig. 5.17).

Translating HELENA LTL Formulae in PROMELA Furthermore, we have to translate the HELENA LTL formulae to PROMELA. In HELENA LTL formulae, we use state label expressions and attribute expressions as atomic propositions (cf. Def. 4.5 on page 71). In principle, they can directly be reused in PROMELA with small adaptations to express operators in ASCII. Additionally, some PROMELA-specific adaptations and scoping adaptations need to be done:

Let's consider a state label expression $rt[n]@label$ where rt is a role type, n an identifier of a role instance, and *label* a state label in a role behavior declaration. Since we

express HELENA role behavior declarations by PROMELA process declarations with the same name and reuse the name of a HELENA state label for the corresponding PROMELA state label, we can directly employ the same syntactic expression in PROMELA LTL formulae. The individual ingredients are thereby newly interpreted: rt is a process type, n an identifier of a role process instance, and *label* a state label in a process declaration. However, special care has to be taken for the identifier of the role process instance in PROMELA. In HELENA, we start counting role instances beginning from 1 (cf. Def. 3.4 on page 50). In PROMELA, however, the identifiers of processes for components and roles are shared, but we create component processes before role processes in the `init`-process explained in the previous paragraph. Therefore, we have to add the number of components to the HELENA identifier of a role instance to get the identifier of the corresponding role process in PROMELA.

Let's move on to attribute expressions. In HELENA, an expression $rt[n]@attr$ refers to the value of the attribute *attr* of the role instance with identifier n of type rt. To transfer this expression to PROMELA, we make two adaptations. Firstly, the identifier is increased as explained before. Secondly, to distinguish role attributes from component attributes, we prefix role attributes with `roleAttr_` in the PROMELA translation.

However, attribute expressions can also contain expressions like $ct[n]@attr$ where n is an identifier of a component instance, ct is a component type and *attr* is a component attribute of ct. In contrast to expressions over role attributes described before, the expression $ct[n]@attr$ can directly be reused in PROMELA. The identifier does not need to be adapted since component identifiers start at 1 in both, HELENA and PROMELA[5]. Furthermore, component attributes are reflected by parameters of component processes and therefore do not get a prefix as local variables.

With these adaptations, LTL formulae are added to the PROMELA file by using the inline specification facilities of PROMELA. The syntax for inline specification is:

```
ltl <name> { <formula> }
```

After the prefix `ltl`, a name identifying the ltl formula is given followed by the desired ltl formulae in curly braces.

Defining LTL Formulae in PROMELA relatively to an Initial Ensemble State: Lastly, as mentioned before, the semantics of HELENA and therefore satisfaction of LTL formulae is defined relatively to a given initial state σ_{init}. Therefore, all translated PROMELA LTL formulae have to be extended such that they are defined relatively to the initial state $trans_{\text{init}}(\sigma_{init})$. We extend the translated PROMELA LTL formula ϕ to $\Box(init \Rightarrow \phi)$. *init* is thereby a property which only holds when the initialization in the `init`-process in PROMELA according to the given initial state in HELENA was finished.

P2P Example: At the beginning of this chapter, we introduced an achieve goal and a maintain goal for our p2p example in HELENA LTL (cf. Fig. 4.4 and Fig. 4.5). To model-check these goals in the full HELENA specification of the p2p example (cf. Chap. 2), we can use the translated PROMELA specification and verify the goals for that specification. To be able to translate the p2p example, we have to remove all process invocations as branches in if-then-else constructs. Therefore, the role behavior of a router is adapted by inlining all process invocations as shown in Appendix C.1. This HELENA ensemble

[5] In PROMELA, process identifiers start at 0, but the `init`-process always gets the identifier 0 such that all user-defined processes have an identifier greater than 0.

specification is translated as proposed in Sec. 5.2 and excerpts of the translation are shown in Sec. 8.3. However, to be able to allow Spin verification, the adaptations mentioned in this subsection need to be carried out. Thus, we first have to define an initial state according to which the goals should be verified. This initial state has then to be established in the `init`-process of the PROMELA translation. Finally, both goals have to be transformed to PROMELA respecting the adaptations previously described.

As initial state in HELENA, we assume that the underlying component-based platform consists of three peers where each of the peers might or might not have the file which is stored in the network. Note that it is possible that none of the peers has the file and thus, the file does not exist in the network. Additionally, only one role instance of type *Requester* exists which is owned by the first peer. The translation of this initial state to PROMELA is established by the `init`-process in Fig. 5.17.

```
1  init {
2    chan p1 = [0] of { PeerOperation };
3    chan p2 = [0] of { PeerOperation };
4    chan p3 = [0] of { PeerOperation };
5
6    if
7    ::run Peer(false,0,p2,p1);
8    ::run Peer(true,12345,p2,p1);
9    fi;
10
11   if
12   ::run Peer(false,0,p3,p2);
13   ::run Peer(true,12345,p3,p2);
14   fi;
15
16   if
17   ::run Peer(false,0,p1,p3);
18   ::run Peer(true,12345,p1,p3);
19   fi;
20
21   chan req;
22   PeerOperation op;
23   op.optype = CREATE_REQUESTER;
24   chan answer = [0] of { chan };
25   op.answer = answer;
26   p1!op;
27   answer?req;
28 }
```

Figure 5.17: The `init`-process for the p2p example in PROMELA

This process first creates three channels `p1`, `p2`, and `p3` in line 2–4. These channels are later on used as the **self** channels of the corresponding peers and therefore allow role-to-component communication. Afterwards, in line 6–19, processes for the three peers are created. A peer process in PROMELA has four parameters corresponding to the peer's component attributes and associations (cf. Fig. 2.1 on page 19) and its **self** channel. The first and second parameter reflect the peer's component attributes `hasFile` and `content`, the third parameter reflects the peer's component association `neighbor`, and the last parameter is the **self** channel of the component process. For each peer, it is nondeterministically decided whether the peer has the file or not, e.g., in line 7 a peer is created which does not have the file expressed by the first parameter being **false** and the second parameter being 0 and in line 8 a peer is created which has the file expressed by the first parameter being **true** and the second parameter being 1. As a third parameter the neighbor is set, so for example for peer `p1`, peer `p2` (reflected

by its **self** channel) is the neighbor. Finally, the corresponding channel from line 2–4 is set as **self** channel. Last in the initialization process, the initial role instance of a requester is created. We use role-to-component communication facilities as they are used during role behavior execution to create this role process. The local variable `req` in line 21 will be used to store the reference to the **self** channel of the newly created role process. In line 22–25, the request to spawn a new role process for the requester is created. The request is sent to the peer `p1` in line 26. The component process for peer `p1` will internally spawn the new role process and therefore execution of the whole transfer ensemble is started. Finally, the `init`-process receives the reference to the newly created role process in line 27.

Furthermore, the two goals need to be translated to Promela. Respecting the aforementioned adaptations to LTL formulae, the achieve goal for our p2p example in Fig. 4.4 is translated and inlined in the Promela file as shown in Fig. 5.18.

```
ltl Achieve {
    [] ( Requester@startRequester ->
         ((Peer[1]:hasFile  ||  Peer[2]:hasFile  ||  Peer[3]:hasFile)
          -> <> Requester:roleAttr_hasFile)
       )
```

Figure 5.18: Achieve goal for the p2p example inlined in Promela

We use `Requester@startRequester` to describe when the initialization process in Promela was finished. Since in the initial state, exactly one role instance of type `Requester` exists, we know that as soon as its start label `startRequester` is reached, initialization was finished. Furthermore, disjunction is expressed by the operator $\vee$ in Helena LTL which is now translated to `||`. Lastly, the role attribute `hasFile` has to be prefixed with `roleAttr_` to distinguish role attributes from component attributes. The adaptation of the identifier of a role identifier cannot be seen here, since we used the abbreviation `Requester` for `Requester[1]`. However, if one would have used it, the translation in the Promela goal would be `Requester[4]` since the underlying component-based platform consists of three peers.

Similarly, the maintain goal for our p2p example in Fig. 4.5 is translated and inlined into the Promela file as shown in Fig. 5.19.

```
ltl Maintain {
    [] ( Requester@startRequester ->
         ((Peer[1]:hasFile  ||  Peer[2]:hasFile  ||  Peer[3]:hasFile)
          -> [] (Peer[1]:hasFile  ||  Peer[2]:hasFile  ||  Peer[3]:hasFile))
       )
```

Figure 5.19: Maintain goal for the p2p example inlined in Promela

With these adaptations, the Promela translation is ready to be checked against the prepared LTL formulae. We will give results and statistics on model-checking the p2p example in the next subsection.

5.3.2 Running Spin

To perform model-checking, we rely on Spin Version 6.4.4 64Bit. Spin first generates a verifier in C. Generation for a PROMELA file `sample.pml` is started with the command:

```
"spin.exe -a sample.pml"
```

Depending on whether a liveness property like achieve goals or a safety property like maintain goals, the generated C file is compiled differently to an executable verifier, called `pan` as abbreviation for process analyzer. However, for both types of goals, we enable advanced code optimization techniques in the C compiler with the runtime option `-O2`. This option yields a more optimized code and therefore speeds up runtime of the later verification process. For achieve goals, compilation is started with the command:

```
"gcc -O2 -o pan pan.c"
```

Afterwards, the verifier is started for an achieve goal with the name `AchieveGoal` in the original PROMELA file with the command:

```
"pan -a -N Achieve"
```

For maintain goals, compilation is started with the command:

```
"gcc -O2 -DSAFETY -o pan pan.c"
```

Afterwards, the verifier is started for a maintain goal with the name `MaintainGoal` in the original PROMELA file with the command:

```
"pan -N Maintain"
```

In both cases, achieve goals and maintain goals, the verifier executes a full state space search. Thus, nondeterminism resulting from **if-** and **do-**statements and scheduling of processes is resolved in all possible ways and the state space is searched exhaustively.

Typical Model Size: Holzmann et al. [HB07] use different case studies to evaluate the model-checker Spin. For small models like the dining philosophers[6], the size of the vector to store a single state is a few hundred bytes, i.e., around 200 bytes and the state space to be searched contains a few million states, i.e., 1–35 million states. For larger models as discussed in [HB07], the state space grows to several hundred million states, i.e., 10–600 million states, and/or the size of the vector to store a single state grows up to 4000 bytes.

Memory Consumption: Spin can by default cope with state vectors with a size up to 1024 bytes (B). If this upper bound is exceeded by the employed model, Spin may be advised to allocate more memory for the state vector by adding the option `-DVECTORSZ=N` to the compilation of the executable verifier (step 2) where `N` is the size of the state vector in bytes. On the other hand, Spin uses up to 200 mega bytes (MB) to store the searched state space. If the model exceeds this memory limit, the upper bound can be increased by adding the option `-DMEMLIM=N` to the compilation of the executable verifier (step 2) where `N` is the memory to be allocated in mega bytes. This number should most certainly not exceed the amount of physical memory in the machine used for verification.

[6] Most of the benchmark example models can be downloaded with the Spin distribution from `http://www.spinroot.com/spin/multicore/`

Lossless Optimization: To scale verification to larger systems as we will handle them later on in our case study, Spin offers different optimization techniques [Hol03, Chap. 9] which improve the memory consumption of the verification run, but may increase the runtime.

The first optimization technique *Partial Order Reduction (POR)* reduces the number of reachable states to be searched during verification. POR is enabled in Spin by default and can be disabled by adding the parameter `-DNOREDUCE` to the second step of compiling the C verifier.

The second type of optimization techniques reduces the amount of memory for the storage of states without loss of information, but possibly with increase of runtime. On the hand, we use *Collapse* compression by adding the option `-DCOLLAPSE` to the compilation of the executable verifier (step 2). Collapse compression exploits the fact that many states only differ at a few points. Hence, the overlapping part does not need to be stored multiple times, but only the differing part needs to extracted and stored. On the other hand, we use a *Minimized Automaton (MA)* representation for the state space by adding the option `-DMA=N` to the compilation of the executable verifier (step 2). where `N` is an estimate of the maximal depth of the graph used as the MA representation. This compression technique builds and maintains a minimal deterministic state automaton representing states without duplicates. To achieve maximal compression, both techniques can be used simultaneously.

Lossy Optimization: If these lossless optimization techniques are not able to reduce the size of the searched state space sufficiently, Spin also offers some lossy optimization techniques [Hol03, Chap. 9] to stay in the available memory limits for verification.

The most prominent technique to tackle very large models is *bit state space search.* It is added to the compilation of the executable verifier (step 2) by the option `-DBITSTATE`. Its main idea is to give the explored state space some structure by storing the states in a hash-table. During searching the state space, every explored state is looked up in the hash-table. If it is not contained in the hash-table, it has not been visited before, is added the hash-table and search continues for this part of the state space. If it is already contained in the hash-table, it has been visited before and therefore search can stop exploring this part of the state space since it already has been searched before. In principle, exploiting a hash-table as a structuring means for the state space is not a lossy optimization techniques. Each state is stored in the slot of the hash-table which is characterized by its hash-value. If several states have the same hash-value, they are stored in the same slot as a list. However, if the memory consumption of the hash-table exceeds the memory of the machine running the verification, no more states can be added to the hash-table and the verification run starts to become lossy. States are missed and the verification run does not cover the whole state space anymore. An indicator for this coverage is the hash-factor printed in the output of a bit state space search with Spin. It is approximately the number of available slots in the hash table divided by the slots which are actually engaged by a state during search. If the hash-factor is greater than 100, we can expect a coverage of 100% since collisions are not likely in this case. If the hash-factor approaches 1, the coverage decreases down to 0%.

Another technique is *hash-compact search (HC4).* It is added to the compilation of executable verifier (step 2) by the option `-DHC4`. In contrast to bit state space search, this method is more likely to achieve a good coverage. Similarly to bit state space search, the state space is stored in a hash-table, but it uses a hash-function which could address a lot more slots than they are actually available. With this large hash-function,

collision are unlikely to happen such that instead of the state, we can now just store the hash-value of the state. This yields a compression of the state space in addition to omitting already visited states. This means that for large models which exceed the memory during a full state space search even with the lossless compression techniques mentioned before, one should first try to run a hash-compact search which is most likely to reach a coverage of 100% if enough memory is available. Only if hash-compact search also exceeds the memory, one should use bit state space search to get results with a lower coverage.

Multi-core Model-Checking: Lastly, Spin supports a *multi-core depth-first search* mode since Version 5.0 and a *multi-core breadth-first search* mode since Version 6.2 (more details can be found in the online documentation at `http://www.spinroot.com/spin/multicore/`).

The depth-first search mode [HB07] is enabled by adding the option `-DNCORE=N` to the compilation of verifier where `N` is the number of core to be used. The depth-first search mode works for liveness properties like achieve goals and safety properties like maintain goals. However, for checking liveness properties only two cores can be used because the checking algorithm is mainly dualcore.

The breadth-first search mode [Hol12] is enabled by adding the option `-DBFS_PAR` to the compilation of verifier. During verification, it uses by default all available cores except one. This multi-core verification mode is especially targeted at safety properties, it was also extended to allow verification of bounded liveness properties.

Spin Output: The output of Spin contains statistics about the verification runs in terms of memory consumption and execution time. It also lists the employed optimization techniques and possibly unreached states during the full state space search. Most importantly, Spin output whether verification ended with an error or not. An error can have two reasons: Either Spin ran out of memory which is shown at the beginning of the output or the checked LTL property is not satisfied in the PROMELA specification. For the latter, Spin also provides a counterexample by issuing the command:

```
"spin.exe -t -c sample.pml"
```

The counterexample is a trace through the PROMELA specification. To map the counterexample back to HELENA, it is therefore necessary to fully understand the translation from HELENA to PROMELA. It is still an open problem how to generate a trace in the original HELENA specification instead of the trace in the PROMELA translation.

Example: For our p2p example, we check both goals presented in Fig. 5.18 and Fig. 5.19. Verification is performed with Spin version 6.4.4 64-bit and GCC version 6.9.2 64-bit[7] on a 64-bit Debian 8.1 desktop computer with 32GB RAM and eight Intel(R) Xenon(R) cores each running on 3.40Ghz.

The achieve goal is satisfied for the translated PROMELA specification (and therefore because of stutter trace equivalence also for the original HELENA ensemble specification). The statistics of different verification runs concerning optimization are shown in Table 5.1: The table compares runs in full state space search without any optimization technique, without any state compression, but with POR optimization, and with POR optimization and both state compression techniques Collapse Compression and

[7] It is important to use the 64-bit versions of Spin and GCC to be able to address more than 2GB of memory.

Table 5.1: Statistics of model-checking the p2p example against its achieve goal in a full state space search without any optimization (no opt.), with POR optimization (POR), and POR optimization together with both two compression techniques Collapse compression and Minimized Automaton (MA) representation (lossless opt.) and in a hash-compact search (HC4)

	no opt.	**POR**	**lossless opt.**	**HC4**
search depth	283	283	283	283
state vector size (B)	736	736	736	736
stored states	24134	5069	7309	5069
transitions	130917	14991	14991	14991
theoretic memory (MB)	17.584	3.693	5.381	3.693
actual memory (MB)	10.044	2.220	0.402	0.361
elapsed time (sec)	0.11	0.02	0.05	0.01

MA representation against a run in hash-compact search (HC4). The search depth of 283, i.e., the maximal depth of a path through the search space, and the state vector size of 736 bytes is the same for all verification runs. However, the runs differ highly in the actual memory consumption for storing all states. Without any optimization, the actual[8] memory consumption is around 10 MB; with POR optimization but without any state compression, the actual memory consumption is around 2 MB; with POR optimization and both state compression techniques, the actual memory consumption is only around 0.4 MB; and with hash-compact search 0.4 MB. Concerning execution time, runtime is improving from 0.11 seconds for a run without any optimization technique to 0.02 seconds with POR optimization only. This improvement is caused by the fact that POR optimization reduces the number of reachable states to be searched. However, execution time increases to 0.05 seconds again when state compression techniques are additionally used. State compression techniques reduce the memory consumption for storing states, but this comes with the disadvantage of possibly increasing runtime because of low-performance data structures. One interesting side-effect of using POR optimization together with state compression is that POR optimization might not be able to reduce the reachable states as much as without state compression (the number of stored states with state compression is 7309 while it is 5069 without state compression). For hash-compact search, runtime improves to 0.01 seconds again since POR optimization can come to its full potential such that the number of searched states is the same as with POR optimization only. As we can see, hash-compact search has a coverage of 100% in this case and is therefore reliable since the same number of states is search as with POR optimization only.

The output of the verification runs furthermore produces an overview about the states which were not reached during the verification. Since we employ a full state space search, these states represent dead code in the PROMELA specification which can never be reached. This does not impede verification, but can be used to improve the

[8]The theoretic memory consumption is the memory consumption which would be needed if no optimization techniques are employed at all. The actual memory consumption, however, is the memory consumption which is really needed and results from applying optimization techniques. Note that even without any special optimization techniques or with only POR optimization, Spin employs a simple byte masking technique to reduce memory consumption.

specification. In our case, the end states of all role processes are not reached. This is expected since the last statement of each role process is **false** which is never executable. Furthermore, some statements in the component process for the peer are not reached. They all refer to requests which are never sent in our p2p example, e.g., a plays query for a requester, and thus can be tolerated as dead code.

The maintain goal is also satisfied for the translated PROMELA specification (and therefore also for the original HELENA ensemble specification). The statistics of the different verification runs concerning optimization are shown in Table 5.2. The search

Table 5.2: Statistics of model-checking the p2p example against its maintain goal in a full state space search without any optimization (no opt.), with POR optimization (POR), and POR optimization together with both two compression techniques Collapse compression and Minimized Automaton (MA) representation (lossless opt.) and in a hash-compact search (HC4)

	no opt.	**POR**	**lossless opt.**	**HC4**
search depth	283	283	283	283
state vector size (B)	736	736	736	736
stored states	25574	5293	5293	5293
transitions	89770	9290	9290	9290
theoretic memory (MB)	18.438	3.857	3.852	3.857
actual memory(MB)	10.541	2.318	0.306	0.459
elapsed time (sec)	0.08	0.01	0.03	0.01

depth and the state vector size are the same as for the achieve goal. Again, the runs for verifying the maintain goal differ highly in the actual memory consumption for storing all states. Without any optimization, the actual memory consumption is around 11 MB; with POR optimization but without any state compression, the actual memory consumption is around 2 MB; with POR optimization and both state compression techniques, the actual memory consumption is only around 0.3 MB; and with hash-compact search 0.5 MB Concerning execution time, runtime is improving from 0.08 seconds for a run without any optimization technique to 0.01 seconds with POR optimization only, but again increased to 0.03 seconds when state compression techniques are additionally used. The reasons are similar to the verification of the achieve goal. Finally, runtime is improved to 0.01 seconds again if employing hash-compact search. The verification runs furthermore produce the same set of unreached states which are again acceptable due to the aforementioned reasons.

5.4 Related Work

Our approach of verification is in-line with goal-oriented requirements approaches like KAOS [vL09]. They also specify goals by LTL properties. However, they translate their system specifications into the process algebra FSP [MK06], which is not sufficient to represent the dynamics of ensembles since dynamic process creation and directed communication are not supported.

To the best of our knowledge, PROMELA and Spin can express more HELENA features than other model-checkers, especially dynamic process creation and directed message passing. For example, TAPAs [CDLT08] allows to verify concurrent systems specified

in CCS. The specification language does not support any notion of data and dynamic process creation can only be simulated by parallel composition with process invocation. CADP [GLMS13] is a software toolbox for the verification of distributed processes. The tool supports FSP and LOTOS NT (amongst others) as input language and allows to verify LTL properties as required by HELENA. However, both input languages are not enough to support all features of HELENA. FSP does not allow directed message passing between two processes and cannot dynamically create new processes. In contrast, LOTOS NT is a very expressive language which merges process calculi into a programming language. As in TAPAs, dynamic process creation can only be simulated by parallel composition with process invocation. Furthermore, LOTOS NT only allows synchronous communication by rendezvous on gates. mCRL2 [GM14] is a formal specification language for concurrent systems which is supported by a toolset for modeling, validation and verification. The input language resembles FSP, but allows multiactions. As in FSP, communication between processes is achieved via synchronization between actions such that directed message exchange is not possible. Furthermore, dynamic process creation is again only possible by parallel composition with process invocation. It still remains to investigate whether there exists any further model-checkers which could express the HELENA features better than Spin, especially the semantics of the if-then-else construct and atomicity of a sequence of actions. Especially, since we will present the framework jHELENA implementing HELENA ensemble specifications in Java in Chap. 7 and a systematic translation of HELENA ensemble specifications to jHELENA in Sec. 8.4, Java PathFinder [Lau16, VHB+03] could be an interesting option for model-checking HELENA. Java PathFinder allows to analyze executable Java programs for properties like deadlocks, unhandled exceptions, and data races. To check an LTL formula as we use it to express goals, the LTL formula has to be transformed to its corresponding Büchi automaton and implemented by a user-defined listener for the Java PathFinder. This transformation is not directly supported by the core Java PathFinder, but only by an external extension [Cuo12] which is not maintained since 2012. Nevertheless, using Java PathFinder could help us to directly check the final implementation of HELENA ensemble specifications in Java without making detour to another model specific for a particular model-checker.

Techniques for the development of ensembles have been thoroughly studied in the recent ASCENS project [WHKM15]: Closely related to our work is the SCEL language. Therefore, we look at its verification in more detail. In [DLL+14], ensemble-based systems are described by simplified SCEL programs and translated to PROMELA. The main idea of the translation is to declare new PROMELA process types for every process construct used in the SCEL description of the ensemble-based system. For example, action prefix is translated to two PROMELA process types, the first one representing the execution of the action and the second one the execution of the remaining process. Thus, the behavior of the first process type is just the execution of the action and afterwards the spawning of a new process instance which is responsible for executing the remaining process term. In our opinion, this translation suffers from the excessive creation of new process types and process instances. During verification, Spin will only allow to create 255 processes at maximum, afterwards it simply ignores additional spawns. Apart from that, the authors omitted the **new** operator in the simplified version of SCEL and thus did not provide any verification mechanism for dynamically created components. In contrast, HELENA allows dynamic role creation which is fully represented in our PROMELA translation. Furthermore, the translation from simplified SCEL to PROMELA is neither proved semantically correct nor automated while our

translation from HELENA to PROMELA is proven correct in Chap. 6 and automated with the HELENA workbench in Chap. 8. In addition to model-checking SCEL specifications with Spin, SCEL specifications can also be implemented in Java relying on the jRESP framework [Lor16, De 16] which also provides a prototypic statistical model-checker. It can verify whether the implementation of a SCEL specification in jRESP satisfies a reachability property with a certain degree of confidence. So far, HELENA is not able to cope with uncertainty in the environment and thus exact model-checking is possible, but as soon as changing environmental conditions are included in HELENA, verification of HELENA ensemble specifications should move to statistical model-checking as proposed in jRESP for SCEL.

DFINDER [CBK15] implements efficient strategies exploiting compositional verification of invariants to prove safety properties for BIP ensemble models, but does not deal with dynamic creation of components.

DEECo ensemble models [BGH+13] are implemented with the Java framework jDEECo and verified with Java Pathfinder [CBK15]. Thus, opposed to HELENA, they do not need any translation. However, since DEECo relies on knowledge exchange rather than message passing, they do not verify communication behaviors.

In the field of distributed systems, multiparty session types [CDPY15] describe communication protocols of interacting processes on a global level. The behavior of each process is obtained by projecting the global multiparty session type on a single participant. These local projections are used to prove communication safety, protocol fidelity, and progress of the global multiparty session types. In contrast to our approach, these properties can be proven in the context of interleaved multiparty sessions while we only consider one ensemble instance during a verification run so far. However, in the HELENA approach, we support the verification of arbitrary LTL formulae over the underlying ensemble specification while verification of multiparty session types is restricted to communication safety, protocol fidelity, and progress only.

Finally, our approach has been strongly inspired by the way how the distributed language KLAIM has been transferred to Maude in [EMMW15]. There, the correctness of the translation was established by a stutter bisimulation which preserves CTL^* properties (without *next*). The translation of HELENA into PROMELA is, however, not stutter bisimilar but stutter trace equivalent and thus only preserves LTL formulae (without *next*) as we will discuss in Chap. 6.

5.5 Publication History

The content of this chapter relies on [HKW15] and mostly on [Kla15b]. In [HKW15], we introduce a formal translation from a simplified version of HELENA to PROMELA, the input language of the model-checker Spin. In [Kla15b], we informally describe the extension of this translation to full HELENA and give some insights in the formal translation.

This chapter extends these previous publications by fully defining the translation from full HELENA to PROMELA which has only been described in excerpts so far. Furthermore, we include a comprehensive introduction to model-checking HELENA ensemble specifications with Spin, present some optimization techniques included in Spin, and provide detailed statistics about model-checking the p2p example.

5.6 Present Achievements and Future Perspectives

Present Achievements: To check the HELENA LTL goals in a HELENA ensemble specification, we proposed to translate HELENA to PROMELA and to verify goal satisfaction in the PROMELA translation with the model-checker Spin. The translation mainly transfers the two-level approach of HELENA components and roles to PROMELA processes: A HELENA component is represented by a long-running PROMELA process which is repeatedly able to receive requests, like operation calls or role creation and retrieval requests, from its adopted roles. A HELENA role is represented by a short-living PROMELA process which reflects the corresponding role behavior declaration specified in HELENA. It issues requests to its underlying component process and exchanges messages with other role processes. Using this PROMELA translation, the original HELENA goals can be verified with small adaptations in Spin. The obtained results can directly be transferred back to HELENA since we can show that a HELENA ensemble specification and its PROMELA translation satisfy the same set of $LTL_{\setminus \mathbf{X}}$ formulae (the formal proof is given in the next chapter).

We illustrated our approach for checking satisfaction of goals for HELENA ensemble specifications with our p2p example. Although being a very small example, it uses nearly all features of HELENA and provides a good proof of concept. Even at this small example, we could see that PROMELA is well-suited to express the distinct HELENA features and Spin provides powerful optimization and compression methods to speed up verification and to reduce memory consumption. To show that our approach also scales to larger systems, we provide a larger case study in Chap. 10.

Future Perspectives: Nevertheless, there exist interesting areas of future work.

Expressiveness of Spin: We had to restrict the usage of nondeterministic choice and the if-then-else construct to preserve semantic equivalence between a HELENA ensemble specification and its PROMELA translation. Those restrictions are mainly caused by two design choices of PROMELA: Firstly, PROMELA prescribes that a branch of an **if**-construct can only be selected for execution if its first action is executable. It is not possible to extend the selection mechanism such that a block of actions determines the executability of a branch (even not if enclosed in an **atomic**-block). Since some HELENA actions have to be translated to several PROMELA statements, we cannot cope with them as first actions of nondeterministic choice or the if-then-else construct if they are not executable. Secondly, PROMELA's selection **if**-construct does not offer the possibility to evaluate the guard of a branch and to execute the corresponding first action of the branch as one step. Therefore, the semantics of the HELENA if-then-else construct cannot be directly expressed in PROMELA if the first action is not executable. However, as discussed in Sec. 5.4, PROMELA can express more HELENA features than other off-the-shelf model-checkers. Thus, these expressibility problems could at the moment only be overcome by a custom-made HELENA-specific model-checker.

Extension to Component Interfaces: For simplicity reasons, we made the assumption that a role type can only be adopted by one component type. To alleviate this assumption, component interfaces can be introduced into the HELENA approach as already proposed in Sec. 2.7 such that a role type only requires a certain component interface from its owning component. In PROMELA, all component types

would then be represented by corresponding component processes, but requests to components would no longer be component-specific, but interface-specific.

Model-Checking of Parallel Ensembles: So far, we only check one ensemble instance per verification run. This assumption can be alleviated by allowing each component process to adopt the same role in several ensembles. Instead of only one role instance per type, the component process then has to manage an array of role instances, each adopted in a different ensemble. While the extension to several parallel ensembles is syntactically easily realized in PROMELA, it has a huge impact on the size of the search space. To be able to still verify several parallel ensembles, the translation has to be optimized to reduce the amount of space to store the state of the ensemble-based system.

Representation of Counterexamples in HELENA*:* Currently there is no support for mapping model-checking results from PROMELA and Spin back to HELENA. Spin produces a counterexample if an LTL formulae is not satisfied, but the counterexample is given as a trace through the PROMELA translation. Therefore, deep understanding of the translation from HELENA to PROMELA is necessary to be able to map the trace back to HELENA. An automatic generation of the corresponding trace through the HELENA specification would alleviate this problem.

Chapter 6

Correctness Proof
Allowing HELENA to Rely on Spin

In the previous chapters, we proposed to specify goals for a HELENA ensemble specification in LTL and to check satisfaction of goals by translating HELENA to PROMELA and using the model-checker Spin for verification. However, to be able to use the results from model-checking a PROMELA translation for the original HELENA ensemble specification, we have to show that they both satisfy the same set of *LTL* formulae. To prove that kind of semantic equivalence, we establish stutter trace equivalence between the induced semantic Kripke structures of a HELENA specification and of its PROMELA translation. In the proof of stutter trace equivalence, we rely on a new general criterion that Kripke structures are stutter trace equivalent if particular simulations (called $\approx$-stutter simulations) can be established in both directions. This criterion is explained and proven correct in Sec. 6.1. Relying on results from the literature [BK08], we know that satisfaction of LTL formulae (without the *next* operator) is preserved in stutter trace equivalent Kripke structures. As a consequence, we can verify LTL properties for a HELENA specification by model-checking its PROMELA translation. Thereby, the only restrictions we have to make is that no role behavior declaration in HELENA may start with a state label and process invocation is not a branch in nondeterministic choice.

In this chapter, we first explain the notion of stutter trace equivalence and $LTL_{\setminus \mathbf{X}}$[1] preservation in general and introduce the new criterion for stutter trace equivalence in Sec. 6.1. This criterion will be used in the formal proof of the correctness of our approach which follows afterwards. We formally prove stutter trace equivalence for two simplified variants of HELENA and PROMELA and informally argue that the proof can be extended to full HELENA and PROMELA. In Sec. 6.2, we reduce the syntax and semantics of HELENA to the main features of roles defining the simplified variant HELENALIGHT of our modeling language. In Sec. 6.3, we identify a subset of PROMELA, called PROMELALIGHT, which is sufficient to express all HELENALIGHT concepts, and derive SOS rules for PROMELALIGHT from the transition system semantics provided for full PROMELA in [Wei97]. Afterwards in Sec. 6.4, we discuss the formal translation function from HELENALIGHT to PROMELALIGHT which is a simplified variant from the full translation in Sec. 5.2. Based on the formal definition of HELENALIGHT and PROMELALIGHT, we establish stutter trace equivalence between the induced semantic Kripke structures of both types of specifications in Sec. 6.5. Finally, we argue in Sec. 6.6 that the correctness proof can be extended to full HELENA and therefore justify our approach to translate HELENA to PROMELA and to use Spin as a model-checker.

[1] $LTL_{\setminus \mathbf{X}}$ is the fragment of LTL that does not contain the *next* operator $\mathbf{X}$.

6.1 Foundations on $LTL_{\setminus \mathbf{X}}$ Preservation

To define when two Kripke structures satisfy the same set of $LTL_{\setminus \mathbf{X}}$ formulae, we introduce the notion of stutter trace equivalence. This equivalence entails $LTL_{\setminus \mathbf{X}}$ preservation according to the literature [BK08] and, thus, allows us to deduce satisfaction of the same set of $LTL_{\setminus \mathbf{X}}$ formulae.

We first consider paths of Kripke structures (cf. Def. 4.2): two paths of Kripke structures over the same set of atomic propositions AP are *stutter trace equivalent* if their traces only differ in the number of their stutter steps. That means that there exist sets of atomic propositions $P_i \subseteq AP$ (with $i \in \mathbb{N}$) such that the traces of both paths have the form $P_0^+ P_1^+ P_2^+ \ldots$ where P_i^+ denotes a non-empty sequence of the same set P_i.

The notion of stutter trace equivalence is extended to Kripke structures by considering all paths of the Kripke structures.

Def. 6.1: Stutter Trace Equivalence

Two Kripke structures K_1 and K_2 are stutter trace equivalent *if for each path of K_1 there exists a stutter trace equivalent path of K_2 and vice versa.*

Stutter trace equivalence between two Kripke structures allows to find a path in one Kripke structure for a path in the other one, we say the path is simulated. The two paths exhibit the same trace of atomic propositions except stutter steps which do not change the set of satisfied atomic propositions. That means, we can find a mapping between the two paths which sequentially maps states satisfying the same set of atomic propositions. Stutter trace equivalence apparently preserves satisfaction of atomic propositions on equivalent paths according to this mapping. However, stutter trace equivalence does not preserve the behavior expressed by the compared Kripke structures. For stutter trace equivalence, it is only necessary that for every path in the first Kripke structure there exists a path in the stutter trace equivalent Kripke structure which preserves satisfaction of atomic propositions on mapped states, but it is not necessary that at every state on the path in the second Kripke structure preserves the branching structure of the corresponding state of the first Kripke structure.

To exemplify this, we consider the excerpt of the two Kripke structures K_1 and K_2 in Fig. 6.1 (the set of atomic propositions which hold in each state are annotated as subscripts A, B and C). In K_1, we have the path $s = s_0 s_1 \ldots$ with the trace $AB\ldots$. For this path, we find exactly one stutter trace equivalent path in K_2, namely $t = t_0 t_1 t_2 \ldots$ with the trace $AAB\ldots$. We can define a relation $\approx = \{(s_0, t_0), (s_0, t_1), (s_1, t_2), \ldots\}$ which relates all states on the stutter trace equivalent paths satisfying the same set of atomic propositions. While this relation preserves satisfaction of atomic propositions, it does not (necessarily) preserve the branching structure of the Kripke structures in general. In our example, we can choose between two paths in state s_0, one leading to s_2 and one leading to s_1, but in state t_1 we do no longer have the choice. However, we can find a relation $\sim$ which only relates those states which exhibit the same branching behavior, here the relation $\sim = \{(s_0, t_0), (s_1, t_2), \ldots\}$.

In principle, it would be nice to define a relation which simulates all paths of one Kripke structure by the other and preserves satisfaction of atomic propositions. To this end, we introduce the notion of $\approx$-*stutter simulations* in Def. 6.2. In the definition, we combine the idea of the two relations $\sim$ and $\approx$ into one simulation relation. With the relation $\sim$, we simulate all paths (and therefore the branching structure) of one Kripke

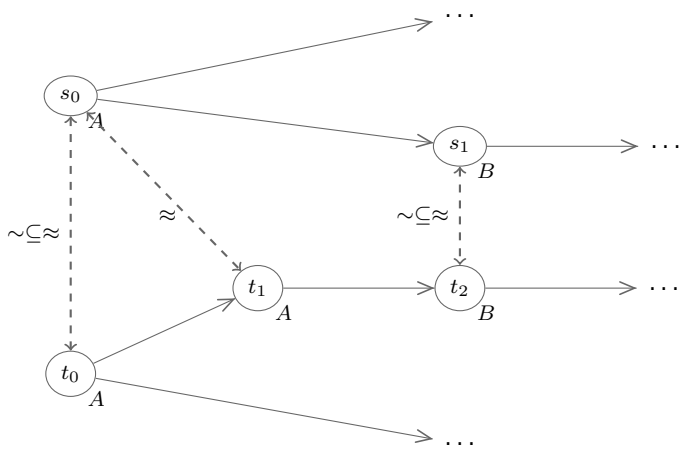


Figure 6.1: Stutter trace equivalent Kripke structures K_1 (above) and K_2 (below)

structure in the other. With the relation $\approx$, we allow stutter steps on the paths which do not have to preserve branching.

Def. 6.2: $\approx$-Stutter Simulation

Let $K_1 = (S_1, I_1, \rightarrow_1, F_1)$ and $K_2 = (S_2, I_2, \rightarrow_2, F_2)$ be two Kripke structures over AP. Let $\approx \subseteq S_1 \times S_2$ be a relation.
A relation $\sim \subseteq S_1 \times S_2$ is a $\approx$-stutter simulation of K_1 by K_2 *if*

(1) $\sim \subseteq \approx$ and

(2) for all $s \in S_1, t \in S_2$ with $s \sim t$: if $s \rightarrow_1 s'$,
then $s' \sim t$ or it exists $t \rightarrow_2 t_1 \rightarrow_2 \ldots \rightarrow_2 t_n \rightarrow_2 t'_1 \rightarrow_2 \ldots \rightarrow_2 t'_m \rightarrow_2 t'$ $(n, m \geq 0)$ such that $s \approx t_i$ for all $i \in \{1, \ldots, n\}$, $s' \approx t'_j$ for all $j \in \{1, \ldots, m\}$ and $s' \sim t'$.

K_1 is $\approx$-stutter simulated by *K_2 if there exists a $\approx$-stutter simulation $\sim$ of K_1 by K_2 such that $s_0 \sim t_0$ for all $s_0 \in I_1, t_0 \in I_2$.*

Interestingly, (mutual) $\approx$-stutter simulations are sufficient to provide a criterion whether two Kripke structures are stutter trace equivalent if we require two additional properties from the underlying stutter step relation $\approx$. Obviously, the relation $\approx$ has to guarantee preservation of atomic propositions since stutter steps shall not change the satisfaction of atomic propositions. Additionally, the relation $\approx$ has to be *divergence-sensitive*, i.e., if one of $\approx$-equivalent states takes infinitely many stutter steps (which do not change satisfaction of atomic propositions), the other one also has to take infinitely many stutter steps. That means that divergence-sensitivity guarantees that for every (infinite) path of one Kripke structure there actually exists an (infinite) path in the other one, i.e., the paths of the two Kripke structures continuously take stutter steps.

Def. 6.3: Property-Preserving Relation

Let $K_1 = (S_1, I_1, \rightarrow_1, F_1)$ and $K_2 = (S_2, I_2, \rightarrow_2, F_2)$ be two Kripke structures over AP. A relation $\approx \subseteq S_1 \times S_2$ is property-preserving *if for all $s \in S_1, t \in S_2$, $s \approx t$ implies $F_1(s) = F_2(t)$.*

Def. 6.4: Divergence-Sensitive Relation

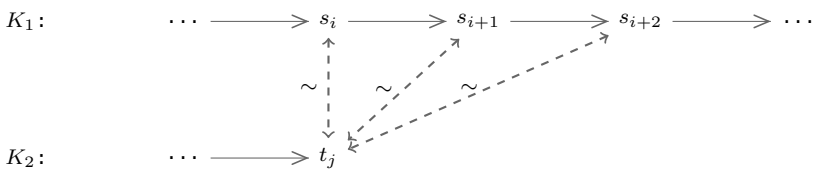
Let $K_1 = (S_1, I_1, \rightarrow_1, F_1)$ and $K_2 = (S_2, I_2, \rightarrow_2, F_2)$ be two Kripke structures over AP. A relation $\approx \subseteq S_1 \times S_2$ is divergence-sensitive *if for all $s_1 \in S_1, t_1 \in S_2$ with $s_1 \approx t_1$ holds: if there exists an (infinite) path fragment $s_1 s_2 s_3 \ldots$ in K_1 with $s_i \approx t_1$ for all $i \geq 1$, then there exists an (infinite) path fragment $t_1 t_2 t_3 \ldots$ in K_2 with $s_1 \approx t_j$ for all $j \geq 1$ and symmetrically for (infinite) path fragments in K_2.*

Thm. 6.5: Stutter Trace Equivalence

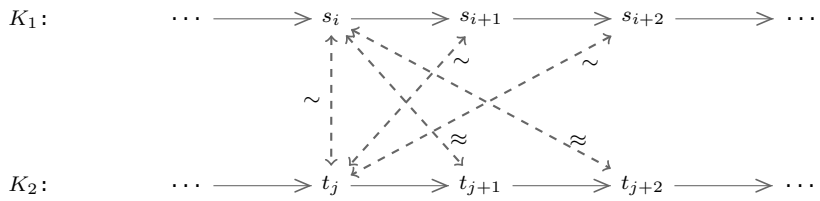
Let K_1 and K_2 be two Kripke structures over AP with states S_1, S_2 resp. Let $\approx \subseteq S_1 \times S_2$ be a property-preserving and divergence-sensitive relation and $\approx^{-1}$ its inverse relation. If K_1 is $\approx$-stutter simulated by K_2 and K_2 is $\approx^{-1}$-stutter simulated by K_1, then K_1 and K_2 are stutter trace equivalent.

Proof of Thm. 6.5

In the proof, we have to show that for every (infinite) path in one Kripke structure there exists an (infinite) path in the other Kripke structure such that the two paths are stutter trace equivalent.
Since K_1 is $\approx$-stutter simulated by K_2, we find for each path $s = s_0 s_1 s_2 \ldots$ of K_1 a finite sequence $t = t_0 t_1 t_2 \ldots t_j$ or an infinite sequence $t = t_0 t_1 t_2 \ldots$ of states $t_i \in S_2$ (of K_2) such that the properties of a $\approx$-stutter simulation from Def. 6.2 hold for all states on the path s and the found sequence t. We first show that the found sequence t cannot be a finite since the relation $\approx$ is divergence-sensitive. We assume that the path s in K_1 is simulated by a finite sequence $t = t_0 t_1 t_2 \ldots t_j$ in K_2. Since the properties of a $\approx$-stutter simulation must be satisfied for the path s and the finite sequence t, the only possibility is that a state $s_i \in S_1$ on the path s of K_1 is reached such that from s_i, K_1 takes infinitely many steps $s_i \rightarrow_1 s_{i+1}$ while the corresponding sequence t in K_2 does not take any further steps and remains in the state t_j such that $s_i \sim t_j$, $s_{i+1} \sim t_j$, $s_{i+2} \sim t_j$, and so on. This reflects the first condition of item (2) in Def. 6.2 of a $\approx$-stutter simulation. The following graphics depicts the situation:



Since we assumed that the relation $\approx$ is divergence-sensitive and it holds that $\sim \subseteq \approx$, the relation $\sim$ is divergence-sensitive, too. Therefore, the depicted situation cannot happen, i.e., the found sequence cannot get stuck in the state t_j while the path in K_1 continuously evolves. The found sequence must rather evolve according the property of a divergence-sensitive relation in Def. 6.4. The following graphics depicts the required extension of the finite path t:



This means that every path s of K_1 is simulated by an infinite sequence, i.e. path t of K_2.
Secondly, we have to discuss why the two paths are stutter trace equivalent. Since we produced the two paths by a $\approx$-stutter simulation, we know that for every state s_i on the path of K_1 we find a corresponding state t_j on the path of K_2 such that either $s_i \sim t_j$ or $s_i \approx t_j$ holds. Furthermore, we know that the relation $\approx$ is property-preserving and that $\sim \subseteq \approx$ holds. Therefore, all states on the paths which are related by $\sim$ or $\approx$ satisfy the same set of atomic propositions which conforms to the definition of stutter trace equivalence of paths.
Since the relation $\approx$ is property-preserving and divergence-sensitive, the inverse relation $\approx^{-1}$ is property-preserving and divergence-sensitive, too. Therefore, the argumentation from before symmetrically holds in the other direction if we assume that K_2 is $\approx^{-1}$-stutter simulated by K_1. ■

Proof of Thm. 6.5

The question arises which LTL formulae are satisfied by two stutter trace equivalent Kripke structures. It is clear that the *next* operator $\mathbf{X}$ of temporal logic is not preserved since stutter steps are allowed. However, if we restrict our attention to the temporal logic $LTL_{\setminus \mathbf{X}}$, we can use a result of [BK08] which shows that all formulae of $LTL_{\setminus \mathbf{X}}$ are preserved. In practice, eliminating the *next* operator is not a great loss since interesting properties are not so much concerned with what happens in the next step as to what eventually happens [Lam83].

Thm. 6.6: $LTL_{\setminus \mathbf{X}}$ Preservation

Let K_1 and K_2 be two stutter trace equivalent Kripke structures over AP. For any $LTL_{\setminus \boldsymbol{X}}$ formula ϕ over AP, we have $K_1 \models \phi \Leftrightarrow K_2 \models \phi$.

Proof of Thm. 6.6

The proof can be found in [BK08, pp. 534–535] (Thm. 7.92 and Cor. 7.93). ■

6.2 HelenaLight

In HelenaLight, we simplify the full Helena approach by omitting the underlying component types and consider only role types, whose instances can be dynamically created, and their interactions. Additionally, we omit any notion of data such that we do not consider attributes and data parameters anymore. Table 6.1 gives an overview about the features of Helena and how they are abstracted in HelenaLight.

Firstly, HelenaLight omits the underlying component-based platform (marked with (1) in Table 6.1). Because of the lack of components, roles do not have an owner.

Table 6.1: Features of HELENA vs. HELENALIGHT

	HELENA	HELENALIGHT
components	attributes	— (1)
	associations	— (1)
	operations	— (1)
roles	owner	— (1)
	attributes	— (2)
	messages	without data (2) and with exactly one role parameter (3)
ensemble structures	capacity of input queues	(cf. HELENA)
	min/max multiplicity	— (4)
actions	role instance creation	without owner (1)
	role instance retrieval	— (1)
	sending a message	without data (2) and with exactly one role parameter (3)
	receiving a message	without data (2) and with exactly one role parameter (3)
	component operation call	— (1)
	component attribute setter	— (1)
	role attribute setter	— (2)
process constructs	termination	no role instance release (1)
	action prefix	(cf. HELENA)
	nondeterministic choice	(cf. HELENA)
	if-then-else	— (2)
	process invocation	only recursive role behavior invocation (5)

Thus, role creation is independent from an owner and role retrieval cannot be realized since there does not exist any owner which manages the references to its adopted roles. Similarly, at the end of a role behavior, the role must not be released from its owner when reaching the null process **quit**. Furthermore, operations cannot be called on components and component attributes cannot be set.

Secondly, any notion of data is omitted (marked with (2) in Table 6.1). Thus, roles do not have attributes, there are no role attribute setters, and messages do not have any data parameters. The if-then-else construct is completely omitted.

We additionally simplify HELENA in three aspects: We only allow a single role instance parameter in messages instead of a list of role instance parameters (marked by (3) in Table 6.1). We do not restrict the number of allowed instances for roles anymore (marked by (4) in Table 6.1). We do not allow arbitrary process invocation anymore, but only recursive invocation of the role behavior itself (marked by (5) in Table 6.1). In the following, the simplified variant of HELENA is defined in detail.

6.2.1 HELENALIGHT Syntax

Ensemble Structures: Role types in HELENALIGHT are characterized by their name and a set of outgoing and incoming message types. In contrast to full HELENA, we omit role attributes and consider message types with exactly one role parameter.

Def. 6.7: HelenaLight Message Type

A message type *msg* in HelenaLight *is of the form* $msgnm(X{:}rt)$ *such that* $msgnm$ *is the name of the message type and* X *is a formal parameter of the* HelenaLight *role type* rt.

Def. 6.8: HelenaLight Role Type

A role type in HelenaLight *is a tuple* $rt = (rtnm, rtmsgs_{out}, rtmsgs_{in})$ *such that*

- $rtnm$ *is the name of the role type, and*
- $rtmsgs_{out}$ *and* $rtmsgs_{in}$ *are sets of* HelenaLight *message types for outgoing and incoming messages supported by the role type.*[a]

[a] In the following, we often write rt synonymously for the role type name $rtnm$.

Ensemble structures specify which role types are needed for a collaboration. In contrast to full Helena, we omit multiplicities constraining the number of admissible role instances for each role type. We assume asynchronous communication and specify for each role type the (positive) capacity of the input queue of each role instance of that type, i.e., the capacity must be greater than zero.

Def. 6.9: HelenaLight Ensemble Structure

An ensemble structure Σ in HelenaLight *is a tuple* $\Sigma = (nm, roletypes, roleconstraints)$ *such that*

- nm *is the name of the ensemble structure,*
- $roletypes$ *is a set of* HelenaLight *role types, and*
- *for each* $rt \in roletypes$, $roleconstraints(rt)$ *is a finite capacity* $c > 0$ *of the input queue of* rt.

As in full Helena, we only consider closed ensemble structures Σ. This means that any outgoing message of some role type of Σ must occur as an incoming message of at least one role type of Σ and vice versa, and any parameter type occurring in a message type is a role type of Σ.

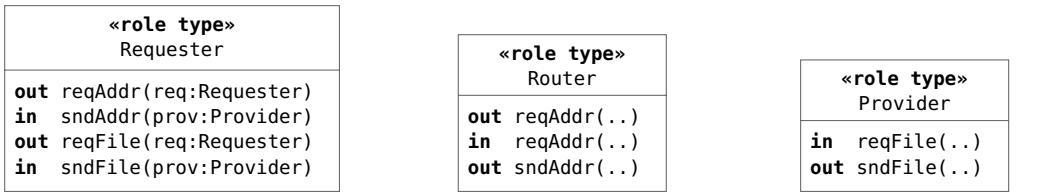
Example: We will consider a simplified variant of the p2p example of Chap. 2. The three role types for requester, router, and provider are formally defined in Fig. 6.2 and depicted in Fig. 6.3.

In contrast to the specification in full Helena, we omit the underlying component type *Peer* and all role attributes as well as data parameters in the specification of the three role types. However, since every message in HelenaLight must have exactly one role instance parameter, we have to equip the message *sndFile* of the *Requester* and the *Provider* with an arbitrary parameter which will not matter in the following.

The ensemble structure for the file transfer ensemble in HelenaLight names the participating role types and their capacity, but no multiplicities (cf. Fig. 6.4 and Fig. 6.5).

$$
\begin{aligned}
\mathit{Requester} &= (\text{"}\mathit{Requester}\text{"}, \mathit{msgs}_{out}(rq), \mathit{msgs}_{in}(rq)) \\
\quad \text{with } \mathit{msgs}_{out}(rq) &= \{\mathit{reqAddr}(\mathit{req}{:}\mathit{Requester}), \mathit{reqFile}(\mathit{req}{:}\mathit{Requester})\} \\
\quad \text{and } \mathit{msgs}_{in}(rq) &= \{\mathit{sndAddr}(\mathit{prov}{:}\mathit{Provider}), \mathit{sndFile}(\mathit{prov}{:}\mathit{Provider})\} \\
\\
\mathit{Router} &= (\text{"}\mathit{Router}\text{"}, \mathit{msgs}_{out}(ro), \mathit{msgs}_{in}(ro)) \\
\quad \text{with } \mathit{msgs}_{out}(ro) &= \{\mathit{reqAddr}(\mathit{req}{:}\mathit{Requester}), \mathit{sndAddr}(\mathit{prov}{:}\mathit{Provider})\} \\
\quad \text{and } \mathit{msgs}_{in}(ro) &= \{\mathit{reqAddr}(\mathit{req}{:}\mathit{Requester})\} \\
\\
\mathit{Provider} &= (\text{"}\mathit{Provider}\text{"}, \mathit{msgs}_{out}(pv), \mathit{msgs}_{in}(pv)) \\
\quad \text{with } \mathit{msgs}_{out}(pv) &= \{\mathit{reqFile}(\mathit{req}{:}\mathit{Requester})\} \\
\quad \text{and } \mathit{msgs}_{in}(pv) &= \{\mathit{sndFile}(\mathit{prov}{:}\mathit{Provider})\}
\end{aligned}
$$

Figure 6.2: All role types for the p2p example in HelenaLight



(a) Role type *Requester* **(b)** Role type *Router* **(c)** Role type *Provider*

Figure 6.3: All role types for the p2p example in HelenaLight in graphical notation

$$
\begin{aligned}
&\Sigma_{transfer} = (\text{"}\Sigma_{transfer}\text{"}, \{\mathit{Requester}, \mathit{Router}, \mathit{Provider}\}, \mathit{roleconstraints}) \\
&\quad \text{with } \mathit{roleconstraints}(\mathit{Requester}) = 2, \\
&\qquad\quad \mathit{roleconstraints}(\mathit{Router}) \;\; = 2, \\
&\qquad\quad \mathit{roleconstraints}(\mathit{Provider}) \; = 1
\end{aligned}
$$

Figure 6.4: Ensemble structure $\Sigma_{transfer}$ for the p2p example in HelenaLight

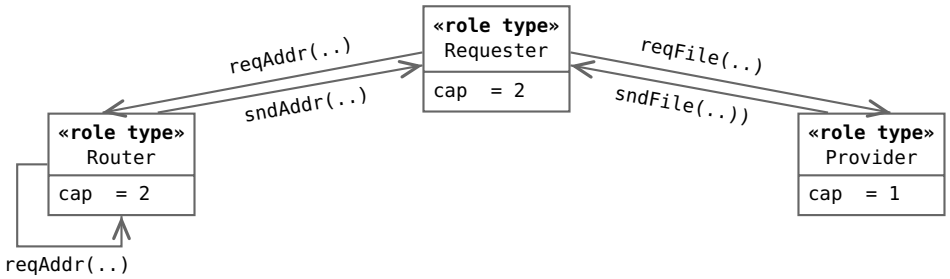


Figure 6.5: Ensemble structure $\Sigma_{transfer}$ for the p2p example in HelenaLight in graphical notation

Ensemble Specifications: To perform a goal-oriented task, the ensemble needs to exhibit a certain behavior. Therefore, we specify a behavior for each role type occurring in the running ensemble which must be respected during execution. Given an ensemble structure Σ, process expressions (over Σ) are used to specify role behaviors. They are built from termination, action prefix, nondeterministic choice, and process invocation. We consider four types of actions: sending (!) and receiving (?) a message, role instance

creation, and labeling a state (used for verification only). Opposed to full HELENA, we omit component instances on which role instances are created and any data in message exchange. Furthermore, we omit the **get** action, operation calls, and any attribute setters since we do not have attributes in HELENALIGHT. Lastly, we also weaken guarded choice to nondeterministic choice since we omitted data completely, i.e., in message exchange and as attributes.

Def. 6.10: HELENALIGHT Process Expression

A process expression in HELENALIGHT *is built from the following grammar, where N is the name of a process, msgnm is the name of a* HELENALIGHT *message type, X and Y are names of variables, rt is a* HELENALIGHT *role type (more precisely the name of a role type), and label is the name of a state label:*

$$
\begin{array}{rll}
P ::= & \textbf{\textit{quit}} & \textit{(termination)} \\
 & \mid a.P & \textit{(action prefix)} \\
 & \mid P_1 + P_2 & \textit{(nondeterministic choice)} \\
 & \mid N & \textit{(process invocation)} \\
a ::= & X \leftarrow \textbf{\textit{create}}(rt) & \textit{(role instance creation)} \\
 & \mid Y!msgnm(X) & \textit{(sending a message)} \\
 & \mid ?msgnm(X{:}rt) & \textit{(receiving a message)} \\
 & \mid label & \textit{(state label)}
\end{array}
$$

A receive action $?msgnm(X{:}rt)$ (and resp. a create action $X \leftarrow \textbf{create}(rt)$) declares and opens the scope for a local variable X of type rt. We assume that the names of the declared variables are unique within a process expression and different from **self** which is a special, predefined variable which refers to the current role instance and can always be used.

In the context of an ensemble structure $\Sigma = (nm, roletypes, roleconstraints)$, a process expression has to satisfy some conditions to be well-formed. These conditions are directly derived from the well-formedness conditions of process expressions in full HELENA (cf. Def. 2.10 on page 28) and do not include any new restrictions.

Def. 6.11: Well-Formedness of a HELENALIGHT Process Expression

Let $\Sigma = (nm, roletypes, roleconstraints)$ be a HELENALIGHT *ensemble structure. A* HELENALIGHT *process expression P is* well-formed *for a* HELENALIGHT *role type $rt \in roletypes$ w.r.t. Σ, if*

(1) in any action prefix of P, all actions are well-formed for rt w.r.t. Σ,

(2) in any nondeterministic choice of P, the first actions of the two branches are either incoming messages or any other action than an incoming message,

(3) in any nondeterministic choice of P, state labels are not the first action of any branch,

(4) a process expression does not immediately invoke itself, also not by a chain of process invocations being the first and last invocation the same.

An action is well-formed *for a* HelenaLight *role type $rt \in roletypes$ w.r.t. Σ if*

*(1) for $X \leftarrow$ **create**(rt'),*

- *(a) $rt' \in roletypes$,*
- *(b) X has not been declared before,*

(2) for $Y!msgnm(X)$,

- *(a) the role type rt supports the message type $msgnm(X':rt')$ as outgoing message,*
- *(b) the type of the role instance Y supports the message type $msgnm(X':rt')$ as incoming message,*
- *(c) the actual parameter X is of type rt',*
- *(d) the expressions X and Y only name the predefined constant **self**, or variables or parameters which have been declared before,*

(3) for $?msgnm(X:rt')$,

- *(a) the role type rt supports the message type $msgnm(X:rt')$ as incoming message,*
- *(b) X has not been declared before,*

(4) all state labels are unique within the process expression P.

Building on process expressions, we can now define role behavior declarations. Opposed to full Helena, a role behavior declaration cannot invoke other processes, but can invoke itself recursively. Furthermore, we neither allow recursive process invocations as the first process construct nor as one immediate option of (possibly nested) nondeterministic choice if nondeterministic choice is the first process construct in the role behavior declaration (as defined in the well-formedness condition). That means, we neither allow **roleBehavior** $rt = rt$ nor **roleBehavior** $rt = (rt + P)$ (and nested variants). We especially name this condition again in the following definition of a role behavior declaration.

Def. 6.12: HelenaLight Role Behavior Declaration

Let Σ be a HelenaLight *ensemble structure and rt be a* HelenaLight *role type in Σ. A* role behavior declaration for rt in HelenaLight *has the form*

$$\textbf{\textit{roleBehavior}}\ rt = P$$

where P is a HelenaLight *process expression which is well-formed for rt w.r.t. Σ such that recursive process invocations may occur in P at most for rt and not immediately.*

An ensemble specification consists, as in full Helena, of two parts: an ensemble structure and a set of role behavior declarations for all role types occurring in the ensemble structure.

Def. 6.13: HelenaLight Ensemble Specification

An HelenaLight ensemble specification *is a pair* $EnsSpec = (\Sigma, behaviors)$ *such that* Σ *is a* HelenaLight *ensemble structure and behaviors is a set of* HelenaLight *role behavior declarations which contains exactly one declaration* ***roleBehavior*** $rt = P$ *for each role type* $rt \in \Sigma$.

Example: In the context of our simplified p2p example, we have to consider the restrictions HelenaLight imposes on the specification of behaviors. The behavior of a requester (cf. Fig. 6.6) does not name any component instance where the initial router is created (line 1) and all data parameters in message exchanges are removed (e.g., line 6). To illustrate the use of state labels, we introduce two state labels (line 4 and 7) to mark the state when the requester received the address of a provider and the file itself.

```
roleBehavior Requester = router ← create(Router) .   (1)
                         router!reqAddr(self) .      (2)
                         ?sndAddr(prov:Provider) .   (3)
                         stateSndAddr .              (4)
                         prov!reqFile(self) .        (5)
                         ?sndFile(prov2:Provider) .  (6)
                         stateSndFile .              (7)
                         quit                        (8)
```

Figure 6.6: Role behavior of a *Requester* for the p2p example in HelenaLight

Similarly, the behavior of a provider is adapted to HelenaLight (cf. Fig. 6.7).

```
roleBehavior provider = ?reqFile(req:Requester) .
                        stateReqFile .
                        req!sndFile(self) .
                        quit
```

Figure 6.7: Role behavior of a *Provider* for the p2p example in HelenaLight

However, the role behavior of a router has to be fundamentally restructured (cf. Fig. 6.8). Process expressions in HelenaLight can only use nondeterministic choice instead of guarded choice. Thus, in contrast to the router behavior in full Helena in Fig. 2.7 on page 31, the router *nondeterministically* either provides the file or forwards the request (cf. nondeterministic choice in line 2–8). Additionally, we do not provide an action to retrieve an already existing role instance (action **get** in full Helena). Therefore, a router can only forward the request to a newly created router (cf. line 6–8) and not to an already existing one as in full Helena.

6.2.2 HelenaLight Semantics

On the semantic level, we reduce the formal semantics of full Helena in Chap. 3 to the proposed simplified variant HelenaLight. Therefore, as for full Helena, the semantic domain of HelenaLight ensemble specifications are labeled transition systems

```
roleBehavior Router = ?reqAddr(req:Requester) .                (1)
                      ( provider ← create(Provider) .          (2)
                        req!sndAddr(provider) .                (3)
                        quit )                                 (4)
                      +                                        (5)
                      ( router ← create(Router) .              (6)
                        router!reqAddr(req) .                  (7)
                        Router )                               (8)
```

Figure 6.8: Role behavior of a *Router* for the p2p example in HelenaLight

describing the evolution of ensembles. The states capture the current state of an ensemble with its constituent role instances. Structured operational semantics (SOS) rules define the allowed transitions between those ensemble states.

Ensemble States: Let us first consider the states of an ensemble. As for full Helena, an ensemble state in HelenaLight captures the set of currently existing role instances with their local states, but in contrast to full Helena, we abstract from component instances and any notion of data. Thus, the local state of a role instance in HelenaLight only stores its role type, the values of all role instance variables used in its role behavior including the special, predefined variable **self**, the content of its message queue, and a process expression describing the current progress of its role behavior.

Def. 6.14: HelenaLight Role Instance State

The local state of a role instance *in* HelenaLight *is a tuple* (rt, v, q, P) *which stores the following information:*

- *the unmodifiable role type* $rt = (rtnm, rtmsgs_{out}, rtmsgs_{in})$ *of the instance,*
- *a (possibly partial) extensible local environment function* $v : rivars \cup \{\textbf{\textit{self}}\} \to \mathbb{N}^+$ *mapping local role instance variables to values, i.e., role instances identified by natural numbers,*
- *the current content* q *of the input queue of the instance (the empty queue is denoted by* ε*, the length of* q *is denoted by* $|q|$*), and*
- *a process expression* P *representing the current control state of the instance or* $\perp$ *representing termination.*

The set $\mathcal{L}_{roles}$ *denotes all local states of role instances.*

An ensemble state in HelenaLight only has to represent the current state of its constituent role instances. Hence, an ensemble state is a single (extensible) finite function which maps role instance identifiers to local states of role instances.

Def. 6.15: HelenaLight Ensemble State

The global state of an ensemble *in* HelenaLight *is an (extensible) finite function* $\sigma : \mathbb{N}^+ \to \mathcal{L}_{roles}$ *mapping each role instance identified by a unique role identifier to a local state.*

A HELENALIGHT ensemble state has to satisfy the same restrictions to be well-defined as in HELENA. However, the restrictions are reduced to local states of role instances only since we omit the level of component instances in HELENALIGHT completely. That means that (1) the type of the role instance is part of the underlying ensemble structure, (2) the **self** reference refers to the unique identifier of the role instance, (3) all variables in the definition domain of the local environment function have a value pointing to an existing role instance, (4) the input queue of the role instance stores at most as many messages as its capacity, (5) all messages in the input queue only name role instances as parameters which exist in the current ensemble state, (6) the process expression P describing the current control state of the role instance is well-formed except that variables which occur in the local environment function do not have to be declared in P before.

Def. 6.16: Well-Definedness of a HELENALIGHT Ensemble State

A global HELENALIGHT ensemble state σ is well-defined *w.r.t. a* HELENALIGHT *ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *with* $\Sigma = (nm, roletypes, roleconstraints)$ *if for all* $i \in \sigma$ *and* $\sigma(i) = (rt, v, q, P)$*:*

(1) $rt \in roletypes$*,*

(2) $v(\textbf{\textit{self}}) = i$*,*

(3) for any $X \in dom(v) : v(X) \in dom(\sigma)$*,*

(4) $|q| \leq roleconstraints(rt)$*,*

(5) for $q = msgnm_1(k_1) \cdot \ldots \cdot msgnm_m(k_m) : k_1, \ldots, k_m \in dom(\sigma)$*,*

(6) P *is well-formed for* rt *w.r.t.* Σ *with the exception of all (local) variables* X *occurring in* $dom(v)$*.*

Similarly, an admissible initial ensemble state in HELENALIGHT has to satisfy the same conditions as in HELENA, but without any notion of component instances. There must exist at least one role instance in an admissible initial ensemble state. All role instances existing in the initial state must be initial themselves, in the sense that they must be at the beginning of their corresponding role behavior without having executed any actions so far.

Def. 6.17: Admissible Initial HELENALIGHT Ensemble State

A well-defined HELENALIGHT *ensemble state* σ *is an* admissible initial state *for the* HELENALIGHT *ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *with* $\Sigma = (nm, roletypes, roleconstraints)$ *if*

(1) there exists $i \in dom(\sigma)$*,*

(2) for all $i \in dom(\sigma)$*:* $\sigma(i) = (rt, \emptyset[\textbf{\textit{self}} \mapsto i], \varepsilon, P)$ *such that*

(3) behaviors contains the declaration ***roleBehavior*** $rt = P$*, i.e.,* P *is the process expression in the declaration of the role behavior for* rt*.*

Again, well-definedness is not a real restriction if we consider the execution of an ensemble starting in an admissible initial ensemble state: Any admissible initial state

of *EnsSpec* is well-defined per definition and the SOS rules of HelenaLight which are presented in the next section preserve well-definedness. This follows from the syntactic restrictions for process expressions and therefore role behavior declarations in HelenaLight ensemble specifications. The most important restrictions result from send actions. A send action in a process expression in HelenaLight is only well-formed if (amongst others) the variables X and Y have been declared before (or refer to the predefined variable **self**). Declaration however is done via receive or create actions such that each send action must be preceded by appropriate receive and create actions if a process expression well-formed. That matches the requirements for well-definedness of the ensemble states described in Def. 6.16.

Structured Operational Semantics Rules: Starting from an admissible initial state, we can now evolve ensemble states. The allowed transitions are captured by structured operational semantics (SOS) rules. We again pursue an incremental approach by splitting the rules into two different layers. The first layer describes how a single role behavior evolves according to the reduced set of constructs for process expressions of the last section. The second layer builds on the first one by defining the evolution of a whole ensemble from the concurrent evolution of its constituent role instances. In contrast to full Helena, we abstract away from component instances adopting roles and omit any data like attributes of role instances or data parameters of messages.

On the first level, we only formalize the progress of a single role behavior given by a process expression in HelenaLight. Fig. 6.9 defines the SOS rules inductively over the reduced structure of HelenaLight process expressions in Def. 6.10 where the symbol $\hookrightarrow$ describes transitions on this level (in contrast to full Helena, we can omit the role instance i and the ensemble state σ at the transition).

(action prefix)	$a.P \overset{a}{\hookrightarrow} P$	
(nondet. choice-1)	$\dfrac{P_1 \overset{a}{\hookrightarrow}_{i,\sigma} P_1'}{P_1 + P_2 \overset{a}{\hookrightarrow} P_1'}$	
(nondet. choice-2)	$\dfrac{P_2 \overset{a}{\hookrightarrow} P_2'}{P_1 + P_2 \overset{a}{\hookrightarrow} P_2'}$	
(process invocation)	$\dfrac{P \overset{a}{\hookrightarrow} P'}{rt \overset{a}{\hookrightarrow} P'}$	if **roleBehavior** $rt = P$

Figure 6.9: SOS rules for the evolution of roles in HelenaLight

Termination **quit** cannot evolve at all. In contrast to full Helena, a role is not adopted by a component such that the process construct **quit** simply terminates execution without any further internal termination action like quitting to play a role. There are no restrictions on the execution of actions and therefore on action prefix. Nondeterministic choice can either evolve the left branch or the right branch of the construct depending on the executability of either branch. If another process rt is invoked in the current process expression, it evolves like the process expression P declared in the role behavior rt. Note that the rule for process invocation relies on a given role behavior declaration (and can therefore only be evaluated in a context where this role behavior

declaration is given). Furthermore, note that process invocations are only allowed for the role behavior itself in HELENALIGHT. In summary, the rules for the evolution of roles in HELENALIGHT differ only slightly from the ones in HELENA: the if-then-else construct is omitted in HELENALIGHT; process invocation is restricted to recursive role behavior invocation only such that we do not allow to call any other associated process declaration.

On the next level, we consider ensemble states and the concurrent execution of role instances. For a given ensemble specification $\mathit{EnsSpec} = (\Sigma, \mathit{behaviors})$, the allowed transitions between ensemble states, denoted by $\rightarrow_{\text{HEL}}$, are described by the SOS rules in Fig. 6.10. They evolve an ensemble specification *EnsSpec* under the assumption of asynchronous communication. For each rule, the transition between two ensemble states is inferred from a transition of process expressions on the role instance level, denoted by $\hookrightarrow$ in Fig. 6.13. The rules concern state changes of existing role instances in accordance to communication actions, the creation of new role instances (which start execution in the initial state of the behavior of their corresponding role type) and state label actions. The labels on the transitions of $\rightarrow_{\text{HEL}}$ indicate which role instance i currently executes which action from its role behavior specification.

$$\text{(create)} \quad \frac{P_i \xhookrightarrow{X \leftarrow \mathbf{create}(rt_j)} P_i'}{\sigma \xrightarrow{i:X \leftarrow \mathbf{create}(rt_j)}_{\text{HEL}} \sigma'}$$

$$\text{if} \begin{cases} (1)\ i \in \mathit{dom}(\sigma), \sigma(i) = (rt_i, v_i, q_i, P_i), \\ (2)\ \mathbf{roleBehavior}\ rt_j = P_j, \\ (3)\ \sigma' = \sigma[i \mapsto (rt_i, v_i[X \mapsto \mathit{next}(\sigma)], q_i, P_i')] \\ \qquad\qquad [\mathit{next}(\sigma) \mapsto (rt_j, \emptyset[\mathbf{self} \mapsto \mathit{next}(\sigma)], \varepsilon, P_j)]. \end{cases}$$

$$\text{(send)} \quad \frac{P_i \xhookrightarrow{Y!msgnm(X)} P_i'}{\sigma \xrightarrow{i:Y!msgnm(X)}_{\text{HEL}} \sigma'}$$

$$\text{if} \begin{cases} (1)\ i \in \mathit{dom}(\sigma), \sigma(i) = (rt_i, v_i, q_i, P_i), \\ (2)\ v_i(Y) = j \in \mathit{dom}(\sigma), \sigma(j) = (rt_j, v_j, q_j, P_j), \\ \quad\ |q_j| < \mathit{roleconstraints}(rt_j), \\ (3)\ v_i(X) = k \in \mathit{dom}(\sigma) \\ (4)\ \sigma' = \sigma[i \mapsto (rt_i, v_i, q_i, P_i')] \\ \qquad\qquad [j \mapsto (rt_j, v_j, q_j \cdot \mathit{msgnm}(k), P_j)]. \end{cases}$$

$$\text{(receive)} \quad \frac{P_i \xhookrightarrow{?msgnm(X:rt_j)} P_i'}{\sigma \xrightarrow{i:?msgnm(X:rt_j)}_{\text{HEL}} \sigma'}$$

$$\text{if} \begin{cases} (1)\ i \in \mathit{dom}(\sigma), \sigma(i) = (rt_i, v_i, \mathit{msgnm}(j) \cdot q_i, P_i), \\ (2)\ j \in \mathit{dom}(\sigma), \\ (3)\ \sigma' = \sigma[i \mapsto (rt_i, v_i[X \mapsto j], q_i, P_i')] \end{cases}$$

$$\text{(label)} \quad \frac{P_i \xhookrightarrow{label} P_i'}{\sigma \xrightarrow{i:label}_{\text{HEL}} \sigma[i \mapsto (rt_i, v_i, q_i, P_i')]}$$

$$\text{if } i \in \mathit{dom}(\sigma), \sigma(i) = (rt_i, v_i, q_i, P_i).$$

Figure 6.10: SOS rules for the evolution of ensembles in HELENALIGHT

- In contrast to full HELENA, we do not have rules for role instance retrieval, operation call, and attribute setters since these actions are not part of HELENALIGHT.
- We also omit the rule for **quit** since in full HELENA, at the end of each role behavior we need an additional step to remove the adopted-by relation between the role instance and the owning component instance.
- For role instance creation, we do no longer have to take care for the component instance which adopts the new role instance and for any multiplicity restrictions for the type of the role to be created.
- The rules for sending and receiving a message are the same as in full HELENA except that we do not have to include any data parameters.
- State labels are handled exactly as in full HELENA.

Semantics: The semantic rules of HELENALIGHT in Fig. 6.10 generate a labeled transition system with ensemble states evolving by role instance creation, communication actions of roles, and state labels. For an ensemble specification *EnsSpec* and any admissible ensemble state σ_{init}, we retrieve the labeled transition system $T_{\text{HEL}} = (S_{\text{HEL}}, I_{\text{HEL}}, A_{\text{HEL}}, \rightarrow_{\text{HEL}})$ with $I_{\text{HEL}} = \{\sigma_{init}\}$. The states S_{HEL} are all HELENALIGHT ensemble states of the HELENALIGHT ensemble specification *EnsSpec*, the set I_{HEL} of initial states only contains σ_{init}, the actions A_{HEL} are all HELENALIGHT actions on ensemble level, and the transitions in $\rightarrow_{\text{HEL}}$ are described by the SOS rules in Fig. 6.10.

6.2.3 HELENALIGHT LTL

To express goals over HELENALIGHT ensemble specifications, we use a subset of the LTL formulae defined for full HELENA in Def. 4.5 on page 71. In contrast to full HELENA, we omit atomic propositions involving attributes and restrict them to state label expressions of the form $rt[n]@label$ only where rt is a role type of Σ, $n \in \mathbb{N}^+$ is the identifier of a role instance, and *label* is a state label in a role behavior. LTL formulae are built over these propositions as explained in Def. 4.1 on page 67.

Satisfaction: As for full HELENA, we rely on Def. 4.4 on page 68 which requires a set of atomic propositions and a satisfaction relation to define satisfaction of LTL formulae for a labeled transition system.

For an ensemble specification *EnsSpec*, the semantic rules of HELENALIGHT in Fig. 6.10 generate a labeled transition system $(S_{\text{HEL}}, I_{\text{HEL}}, A_{\text{HEL}}, \rightarrow_{\text{HEL}})$ for a given admissible initial state $\sigma_{init} \in I_{\text{HEL}}$. The atomic propositions for *EnsSpec* which are used to formulate LTL formulae are the state label expressions defined by the set $AP(EnsSpec)$. Therefore, it remains to define a satisfaction relation $\sigma \models p$ for $\sigma \in S_{\text{HEL}}$ and $p \in AP(EnsSpec)$.

As for full HELENA, an ensemble state σ satisfies $rt[n]@label$, denoted by $\sigma \models rt[n]@label$, if there exists an active role instance of type rt with identifier n in σ and the next action performed by this role instance is the state label *label*. Formally, that means that in σ there exists $n \in dom(\sigma)$ with $\sigma(n) = (rt, v, q, label.P)$.

Example: For the p2p example, we consider both goals from Sec. 4.2.1. Since we omit any notion of components and data in HELENALIGHT, we can no longer refer to $Peer[..]{:}hasFile$. Thus, we reformulate the achieve goal from Fig. 4.4 on page 71 in Fig. 6.11.

$$\Box(Provider@stateReqFile \Rightarrow \Diamond Requester@stateSndFile)$$

Figure 6.11: Achieve goal for the p2p example in HELENALIGHT LTL

In HELENALIGHT, we omit component types and cannot refer to attributes. Therefore, we express that the file exists in the network by the provider reaching its state labeled by *stateReqFile* (note that we have to add $\Box$ since this state label expression does not hold in the initial state). In this state, the provider has received the request for the file. Thus, since a provider was created, we know that the file exists in the p2p network. Similarly, we express that the file was transferred to the requester by using the state label *stateSndFile* of the requester.

The maintain goal in Fig. 4.5 on page 71 cannot be transferred to HELENALIGHT since we have no notion of data in HELENALIGHT. Thus, we cannot express that the file is stored in the network or deleted from it.

6.3 PROMELALIGHT

PROMELA [Hol03] is a language for modeling systems of concurrent processes. Its most important features are the dynamic creation of processes and support for synchronous and asynchronous communication via message channels. In our model-checking approach for HELENA, we translate a HELENA ensemble specification to PROMELA and check the translated specification with the model-checker Spin [Hol03] for goal satisfaction. Thus, we introduce a simplified variant of PROMELA which is sufficient to express all features of HELENALIGHT. We present syntax and semantics of PROMELALIGHT and discuss goal specifications and their satisfaction in PROMELALIGHT.

6.3.1 PROMELALIGHT Syntax

The following syntax is a simplified version of the PROMELA syntax defined in [Wei97]. The constructs specify a significant sub-language of the PROMELA definition which is sufficient as a target for the translation of HELENALIGHT.

PROMELALIGHT Specifications: Intuitively, a PROMELALIGHT specification consists of a set of process types whose behavior is specified by process expressions. We first define process expressions in PROMELALIGHT based on [Wei97]. We use the same names for nonterminals as in [Wei97], but sometimes we unfold the original definitions to get a smaller grammar. In contrast to [Wei97], we added the expression **false** as an explicit construct corresponding to **quit** in HELENALIGHT. Furthermore, the conditional statement and the **goto** *s* tatement are not treated as process steps, but as a process. Consequently, gotos can only occur at the end of a process expression. We also removed guards from conditional statements, thus obtaining nondeterministic choice.

Def. 6.18: PROMELALIGHT Process Expression

A PROMELALIGHT process expression *seq is built from the following grammar, where label is the name of a state label (used for gotos and verification), var, var_1, and var_2 are the names of variables, const is a constant, pt is the name of a process type, and typelist is a list of types separated by a comma:*

seq	::= ***false***	*(empty process)*
	$\mid$ $step; seq$	*(sequential composition)*
	$\mid$ ***if*** $:: seq_1 :: seq_2$ ***fi***	*(nondeterministic choice)*
	$\mid$ ***goto*** $label$	*(goto)*
$step$	::= ***run*** $pt(var)$	*(run)*
	$\mid$ $var_1!const, var_2$	*(send)*
	$\mid$ $var_1?const, var_2$	*(receive)*
	$\mid$ $label$: ***true***	*(state label)*
	$\mid$ ***chan*** var	*(channel declaration)*
	$\mid$ ***chan*** $var = [const]$ ***of*** $\{typelist\}$	*(channel declaration)*

Note that send and receive steps concern data tuples $const, var_2$ consisting of a constant and a variable. Variables can only refer to channels. A channel declaration **chan** $var = \ldots$ opens the scope for a local channel variable var. We assume that the names of the declared variables are unique within a process expression and different from **self**, which is a predefined variable of type **chan** that can always be used.

A process expression built from the aforementioned grammar has to satisfy some conditions to be well-formed. Thereby, we rely on the notion of initialization of a variable: a variable is *initialized* if either the variable occurs in a receive step as var_2 or in a channel declaration with initialization as var or is the special variable **self**.

Def. 6.19: Well-Formedness of a PromelaLight Process Expression

A PromelaLight *process expression* seq *is* well-formed *if*

(1) all variables occurring in a send or run step have been initialized before,

(2) the variable var_1 *in a receive step has been initialized before and the variable* var_2 *has been declared before, and*

(3) the variable var in a channel declaration has not been declared before,

(4) state labels are unique within the process expression seq, and

(5) state labels are not the first statement of seq_1 *or* seq_2 *in the nondeterministic choice construct* ***if*** $:: seq_1 :: seq_2$ ***fi***.

Process expressions are used to define process types. In PromelaLight, a process type has always one parameter **self** of type **chan** which represents a distinguished input channel for each process instance.

Def. 6.20: PromelaLight Process Type Declaration

A PromelaLight process type declaration *has the form*

$$\textbf{\textit{proctype}}\ pt(\textbf{\textit{chan self}})\{seq_1; start_{pt} : \textbf{\textit{true}}; seq_2\}$$

where pt is the name of the process type and seq_1 *and* seq_2 *are well-formed process*

expressions not containing a state label $start_{pt}$: ***true*** *and any* ***goto*** *expression occurring in seq has the form* ***goto*** $start_{pt}$.

The above definition associates a process expression to a process type pt. It allows a restricted version of recursion by introducing the state label start_{pt} : **true** and allowing to jump back to that via **goto** start_{pt}. This syntactic restriction simplifies the semantics since the continuation of a **goto** is then uniquely determined. Hence, we do not need to carry the full body of a process type declaration in the semantic states and to search for labels in the body to find the continuation as in [Wei97].

Notation: We use the notation $pvars(pt)$ to denote all variables from channel declaration in the process type declaration for the process type pt.

This allows us to finally define a PromelaLight specification which just consists of a set of PromelaLight process type declarations. For simplification, PromelaLight specifications often use enumerations, declared by the keyword **mtype**, which define symbolic names for constants. These symbolic names can then be used instead of constants in send or receive actions as it is shown in the example in Fig. 6.12. Since these symbolic names just improve readability, we do not formally introduce them in the PromelaLight specification, but regard them as simplification constructs.

Def. 6.21: PromelaLight Specification

A PromelaLight specification *PrmSpec consists of a set of* PromelaLight *process type declarations.*

Example: The formal translation from HelenaLight to PromelaLight will be discussed in Sec. 6.4. However to illustrate PromelaLight, we already present here, in Fig. 6.12, the PromelaLight translation of the simplified variant of the p2p example. Let us briefly look at the process type declaration for a router in comparison to the role behavior declaration in Fig. 6.8. Nondeterministic choice is expressed by reusing the **if**-construct of PromelaLight. Role instance creation in HelenaLight is translated to starting a new process in PromelaLight (line 20 and 24). Asynchronous message exchange is obtained by passing an asynchronous channel to the newly created process for communication (line 11 and 12).

6.3.2 PromelaLight Semantics

The semantic domain of PromelaLight specifications are again labeled transition systems. They describe the evolution of a global PromelaLight state. Such a global state captures the states of the currently existing processes and their associated channels. Structured operational semantics (SOS) rules again define the allowed transitions between those global states.

Global States: Let us first consider the global state of a PromelaLight specification. Similarly to ensemble states in HelenaLight, a global state in PromelaLight captures the currently existing process instances. However, in contrast to input queues in HelenaLight, process instances communicate via channels which are not owned by a local process, but belong to the global state. Hence, a global state of a PromelaLight specification captures (1) the set of the currently existing channel instances

```
1  mtype { reqAddr,
2          sndAddr,
3          reqFile,
4          sndFile }
5
6  proctype Requester(chan self) {
7    chan router = [2] of { mtype, chan };
8    chan prov;
9    chan prov2;
10
11   startRequester: true;
12
13   run Router(router);
14
15   router!reqAddr,self;
16
17   self?sndAddr,prov;
18
19   stateSndAddr: true;
20
21   prov!reqFile,self;
22
23   self?sndFile,prov2;
24
25   stateSndFile: true;
26
27   false
28 }
```

(a) Message definitions and process type declaration for `Requester`

```
1  proctype Provider(chan self) {
2    chan req;
3
4    startProvider: true;
5    self?reqFile,req;
6    stateReqFile: true;
7    req!sndFile,self;
8    false
9  }
10 proctype Router(chan self) {
11   chan req;
12   chan prov = [1] of { mtype, chan };
13   chan router = [2] of { mtype, chan };
14
15   startRouter: true;
16
17   self?reqAddr,req;
18   if
19   ::
20     run Provider(prov);
21     req!sndAddr,prov;
22     false
23   ::
24     run Router(router);
25     router!reqAddr,req;
26     goto startRouter
27   fi
28 }
```

(b) Process type declarations for `Provider` and `Router`

Figure 6.12: The p2p example in PromelaLight

(together with their states) and (2) the set of the currently existing process instances (together with their local states).

Let us now look more closely to the formal definition of a global state in PromelaLight. Intuitively, a global state describes the local states of all currently existing channels and the local states of all currently existing process instances. Thus, firstly, the local state of a channel stores on the one hand the unmodifiable type of entries which are allowed in the channel and the maximal capacity of the channel and on the other the current entries in the channel.

Def. 6.22: PromelaLight Channel State

The local state of a channel in PromelaLight *is a tuple* (T, κ, ω) *which stores the following information:*

- *the unmodifiable type* T *of entries of the channel,*
- *the unmodifiable capacity* $\kappa > 0$ *of the channel*[a]*, and*
- *the content* ω *which is a word of* T*-values (the empty word is denoted by* ε*).*

The set $\mathcal{C}$ *denotes all local states of channels.*

[a] In PromelaLight we consider only asynchronous communication and do therefore not allow a capacity $\kappa = 0$.

Secondly, the local state of a process instance stores on the one hand the unmodifiable type of the process referring to a process type declaration. On the other hand, it stores the current values of all local variables used in the process type declaration and a process expression describing the current progress in its role behavior.

Def. 6.23: PROMELALIGHT Process Instance State

The local state of a PROMELALIGHT process instance *is a tuple* (pt, β, π) *which stores the following information:*

- *the unmodifiable process type* pt *of the instance,*
- *a local environment function* $\beta : pvars(pt) \to \mathbb{N}^+ \cup \{\textbf{\textit{null}}\}$ *mapping local variables to values (i.e., channel identifiers or* ***null****), and*
- *process expression* π *representing the current control state of the instance.*

The set $\mathcal{P}$ *denotes all local states of process instances.*

Finally, a global state of a PROMELALIGHT specification captures the local states of all currently existing channel instances and process instances. The channel instances and process instances are represented by finite functions where each channel instance and each process instance is uniquely identified by a positive natural number[2].

Def. 6.24: Global PROMELALIGHT State

The global state γ of a PROMELALIGHT specification *is a pair* $(\mathsf{ch}, \mathsf{proc})$ *such that*

- $\mathsf{ch} : \mathbb{N}^+ \to \mathcal{C}$ *is an extensible finite function mapping each channel instance identified by a unique channel identifier to a local channel state and*
- $\mathsf{proc} : \mathbb{N}^+ \to \mathcal{P}$ *is an extensible finite function mapping each process instance identified by a unique process identifier to a local process instance state.*

A global PROMELALIGHT state has to satisfy some restrictions to be well-defined: (1a) all entries in a channel only name channel instances as parameters which exist in the current global PROMELALIGHT state, (2a) the PROMELALIGHT specification contains the corresponding process type declaration for each process instance, (2b) the **self** reference of a process instance is in the definition domain of izs local environment function, (2c) the **self** reference points to an existing channel instance, (2d) all variables in the definition domain of the local environment function of a process instance (except the **self** reference) point to an existing channel instance or are **null**.

Def. 6.25: Well-Definedness of a Global PROMELALIGHT State

A global PROMELALIGHT state $\gamma = (\mathsf{ch}, \mathsf{proc})$ is well-defined *w.r.t. a* PROMELALIGHT *specification PrmSpec if*

(1) for all $c \in dom(\mathsf{ch})$ *and* $\mathsf{ch}(c) = (T, \kappa, \omega)$*:*

(a) for $\omega = (msgnm_1, c_1) \cdot \ldots \cdot (msgnm_m, c_m)$*:*

[2] For technical reasons, explained in the discussion of initial states below, we deviate from [Wei97] and do not use 0 as an identifier for channels and processes.

$c_1, \ldots, c_m \in dom(\mathsf{ch})$ *and* $\kappa \geq m$,

(2) for all $i \in dom(\mathsf{proc})$ *and* $\mathsf{proc}(i) = (pt, \beta, \pi)$*:*

(a) PrmSpec contains a process type declaration for pt,

(b) $\boldsymbol{self} \in dom(\beta)$ *and* $\beta(\boldsymbol{self}) = i$,

(c) $\beta(\boldsymbol{self}) \in dom(\mathsf{ch})$*, and*

(d) for any $X \in dom(\beta)$ *with* $X \neq \boldsymbol{self}$*:* $\beta(X) \in dom(\mathsf{ch}) \cup \{\boldsymbol{null}\}$,

(e) π *is well-formed except that all (local) variables* $var \in dom(\beta)$ *must not be declared or initialized before usage in* π.

Furthermore, an admissible initial global PromelaLight state has to satisfy some restrictions: (1) all existing channel instances do not contain any entries, (2a) the local environment function of a process instance is empty except the mapping for **self** reference which points to a unique channel instance, and (2b) a process instance refers to a process type declaration contained in the PromelaLight specification and its current control state corresponds to the complete process expression of the process type declaration.

Def. 6.26: Admissible Initial Global PromelaLight State

A well-defined global PromelaLight *state* $\gamma = (\mathsf{ch}, \mathsf{proc})$ *is an admissible initial state for a* PromelaLight *specification PrmSpec if*

(1) for all $c \in dom(\mathsf{ch})$*:* $\mathsf{ch}(c) = (T, \kappa, \varepsilon)$ *for some* T *and* κ,

(2) for all $i \in dom(\mathsf{proc})$*:* $\mathsf{proc}(i) = (pt, \emptyset[\boldsymbol{self} \mapsto c_i], start_{pt} : \boldsymbol{true}; seq)$ *such that*

(a) $c_i \in dom(\mathsf{ch})$ *with* $c_i \neq c_j$ *for* $i \neq j$ *and*

(b) PrmSpec contains the process type declaration $\boldsymbol{proctype}\ pt(\boldsymbol{chan\ self})\{seq_1; start_{pt} : \boldsymbol{true}; seq_2\}$.

Concrete initial states in PromelaLight are constructed by running an appropriate initialization as shown in line 20-23 of Fig. 6.12b where one channel and one requester instance, using that channel as input, are created. The initialization is executed by a root process `init` which implicitly obtains the identifier 0. However, we do not consider this process in a PromelaLight specification and are not interested in the verification of properties for the root process (which anyway does not have any counterpart in a Helena specification). Thus, we use in our semantic framework and in atomic propositions of LTL formulae only positive natural numbers for process identifiers.

As for Helena and HelenaLight, well-definedness is not a real restriction since any admissible initial state is well-defined per definition and the SOS rules of PromelaLight (which are presented in the following) preserve well-definedness. This follows from the syntactic restrictions for well-formed process expressions, and therefore process type declarations in PromelaLight specifications. Again the most important restrictions result from send actions. A send action in a process expression in PromelaLight is only well-formed if (amongst others) the variables var_1 and var_2 have been declared before (or refer to the predefined variable **self**). Declaration however is done

via channel declarations only. That matches the requirements for well-definedness of the global PROMELALIGHT states described in Def. 6.25.

Structured Operational Semantics Rules: Starting from an admissible initial state, we can now evolve global PROMELALIGHT states. The allowed transitions are captured by structured operational semantics (SOS) rules. We also follow a two-level SOS approach which has been advocated for the formal PROMELA semantics in [Wei97]. On the first level, the SOS rules only deal with the progress of process expressions specified by the nonterminal symbol *seq* in Def. 6.18. Process instances and their concurrent execution are considered on the second level.

On the first level, we only formalize the progress determined by a single process expression. Fig. 6.13 defines the SOS rules inductively over the structure of PROMELALIGHT process expressions in Def. 6.18 where the symbol describes transitions on this level. In contrast to [Wei97], we postpone not only the treatment of process instances, but also the treatment of local environments and the consideration of channel instances to the second level.

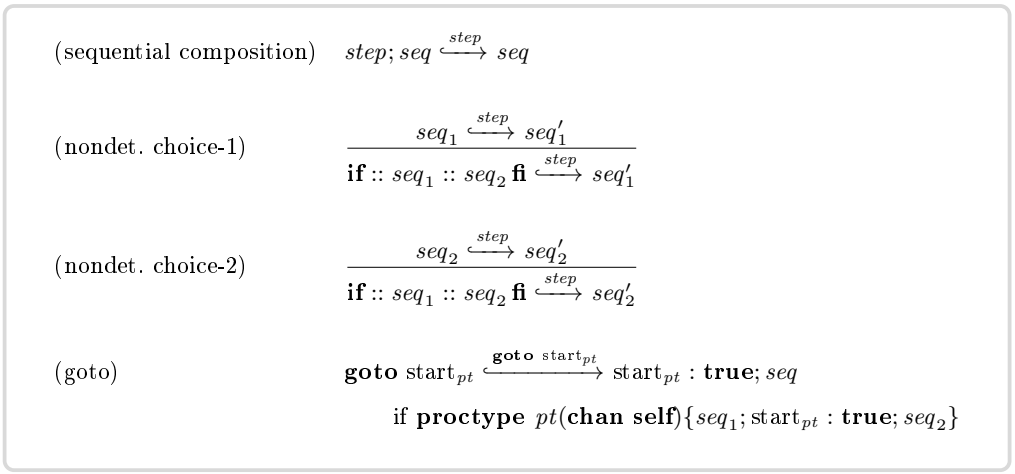


Figure 6.13: SOS rules for the evolution of process expressions in PROMELALIGHT

The empty process **false** cannot evolve at all. Sequential composition may always evolve by doing the first step of the composition. Nondeterministic choice can either evolve the left branch or the right branch of the construct depending on the executability of either branch. The rule for the **goto** expression relies on a given process declaration. Since, in PROMELALIGHT process expressions, it is only allowed to jump to the start label start_{pt}, the rule is only defined for a start label and the **goto** expression evolves to a process expression reflecting the initial state of a process.

On the next level, we consider global states and the concurrent execution of process instances. For a given PROMELALIGHT specification *PrmSpec*, the allowed transitions between global PROMELALIGHT states, denoted by $\rightarrow_{\text{PRM}}$, are described by the SOS rules in Fig. 6.14 and Fig. 6.15.

Transitions between global states are initiated by the actions for sending and receiving a message, running a new process, state labels and channel declarations. They evolve a set of process instances which execute in accordance with their process types under the assumption of asynchronous communication. For each rule, the transition between two global states is inferred from a transition of a single process expression, denoted by $\hookrightarrow$ in Fig. 6.13. We use the same notations as in [Wei97] with less compo-

nents due to the simplified language (with the exception that we also store the type of a process instance in its local state). The labels on the transitions of $\rightarrow_{\text{PRM}}$ indicate which process instance i currently executes which step from its process type specification. In the rules, we use the shorthand notations for the extension and update of finite functions from Sec. 3.1.

(goto)
$$\frac{\pi_i \xhookrightarrow{\textbf{goto } label} \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{i:\textbf{goto } label}_{\text{PRM}} (\mathsf{ch}, \mathsf{proc}[i \mapsto (pt_i, \beta_i, \pi_i')])}$$
if $i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i)$.

(run)
$$\frac{\pi_i \xhookrightarrow{\textbf{run } pt_j(var)} \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{i:\textbf{run } pt_j(var)}_{\text{PRM}} (\mathsf{ch}, \mathsf{proc}')}$$
$$\text{if } \begin{cases} (1)\ i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i), \\ (2)\ \beta_i(var) = c \in dom(\mathsf{ch}), \\ (3)\ \textbf{proctype } pt_j(\textbf{chan self})\{seq_1; \text{start}_{pt_j} : \textbf{true}; seq\}, \\ (4)\ \mathsf{proc}' = \mathsf{proc}[i \mapsto (pt_i, \beta_i, \pi_i')] \\ \qquad\qquad [next(\mathsf{proc}) \mapsto (pt_j, \emptyset[\textbf{self} \mapsto c], seq_1; \text{start}_{pt_j} : \textbf{true}; seq_2)] \end{cases}$$

(send)
$$\frac{\pi_i \xhookrightarrow{var_1!const, var_2} \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{i:var_1!const, var_2}_{\text{PRM}} (\mathsf{ch}', \mathsf{proc}')}$$
$$\text{if } \begin{cases} (1)\ i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i), \\ (2)\ \beta_i(var_1) = c \in dom(\mathsf{ch}), \mathsf{ch}(c) = (T, \kappa, \omega), |\omega| < \kappa, \\ (3)\ \beta_i(var_2) = v \in dom(\mathsf{ch}), \\ (4)\ \mathsf{ch}' = \mathsf{ch}[c \mapsto (T, \kappa, \omega \cdot (const, v))], \\ (5)\ \mathsf{proc}' = \mathsf{proc}[i \mapsto (pt_i, \beta_i, \pi_i')] \end{cases}$$

(receive)
$$\frac{\pi_i \xhookrightarrow{var_1?const, var_2} \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{i:var_1?const, var_2}_{\text{PRM}} (\mathsf{ch}', \mathsf{proc}')}$$
$$\text{if } \begin{cases} (1)\ i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i), \\ (2)\ \beta_i(var_1) = c \in dom(\mathsf{ch}), \mathsf{ch}(c) = (T, \kappa, (const, v) \cdot \omega), v \in dom(\mathsf{ch}), \\ (3)\ var_2 \in dom(\beta_i), \\ (4)\ \mathsf{ch}' = \mathsf{ch}[c \mapsto (T, \kappa, \omega)], \\ (5)\ \mathsf{proc}' = \mathsf{proc}[i \mapsto (pt_i, \beta_i[var_2 \mapsto v], \pi_i')] \end{cases}$$

Figure 6.14: SOS rules for the evolution of concurrent process instances in PROMELALIGHT (part 1)

- The rules for **goto** and state labels are the simplest ones since they only have to consider a single process instance. If the process instance i can execute either a **goto** action or a state label action, the global state of the PROMELALIGHT specification evolves to a state where i just executed that action.
- If a new process instance of type pt_j is spawned, we create a new process instance in proc which is in its initial control state.
- If a process instance i can send a message via $var_1!const, var_2$, we retrieve the channel c which is referenced by var_1 from the local environment β_i of i, check

whether the capacity of the channel c is not yet exceeded by $|\omega| < \kappa$, and add the message $(const, v = \beta_i(var_2))$ to the channel's input.

- Likewise, if a process instance i can receive a message via $var_1?const, var_2$, we retrieve the channel c which is referenced by var_1 from the local environment β_i of i, retrieve the first message $(const, v)$ from the channel c, and bind the value v to the variable var_2 in the local environment β_i of i.
- If a channel variable var is declared without initialization in the process instance i, we just extend the local environment β_i of i by this new (fresh) variable var, but only assign the value **null**.
- If a channel variable var is declared with initialization in the process instance i, we create a new channel in ch and extend the local environment β_i of i by the new (fresh) variable var which is assigned to the newly created channel.

(label)
$$\frac{\pi_i \xhookrightarrow{label:\mathbf{true}} \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{i:label:\mathbf{true}}_{\text{PRM}} (\mathsf{ch}, \mathsf{proc}[i \mapsto (pt_i, \beta_i, \pi_i')])}$$
$$\text{if } i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i).$$

(chan-1)
$$\frac{\pi_i \xhookrightarrow{\mathbf{chan}\ var} \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{i:\mathbf{chan}\ var}_{\text{PRM}} (\mathsf{ch}, \mathsf{proc}')}$$
$$\text{if } \begin{cases} (1)\ i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i), \\ (2)\ \mathsf{proc}' = \mathsf{proc}[i \mapsto (pt_i, \beta_i[var \mapsto \mathbf{null}], \pi_i')] \end{cases}$$

(chan-2)
$$\frac{\pi_i \xhookrightarrow{\mathbf{chan}\ var=[const]\ \mathbf{of}\ \{typelist\}} \pi_i'}{(\mathsf{ch}, \mathsf{proc}) \xrightarrow{i:\mathbf{chan}\ var=\ldots}_{\text{PRM}} (\mathsf{ch}', \mathsf{proc}')}$$
$$\text{if } \begin{cases} (1)\ i \in dom(\mathsf{proc}), \mathsf{proc}(i) = (pt_i, \beta_i, \pi_i), \\ (2)\ \mathsf{ch}' = \mathsf{ch}[next(\mathsf{ch}) \mapsto (typelist, const, \varepsilon)] \\ (3)\ \mathsf{proc}' = \mathsf{proc}[i \mapsto (pt_i, \beta_i[var \mapsto next(\mathsf{ch})], \pi_i')] \end{cases}$$

Figure 6.15: SOS rules for the evolution of concurrent process instances in PROMELALIGHT (part 2)

Semantics: The semantic rules of PROMELALIGHT in Fig. 6.14 and Fig. 6.15 generate a labeled transition system with global PROMELALIGHT states evolving by sending and receiving a message, running a new process, state labels and channel declarations. For a PROMELALIGHT specification *PrmSpec* and any admissible global state γ_{init}, we retrieve the labeled transition system $T_{\text{PRM}} = (S_{\text{PRM}}, I_{\text{PRM}}, A_{\text{PRM}}, \rightarrow_{\text{PRM}})$ with $I_{\text{PRM}} = \{\gamma_{init}\}$. The states S_{PRM} are all global PROMELALIGHT states of the PROMELALIGHT specification *PrmSpec*, the set I_{PRM} of initial states only contains γ_{init}, the actions A_{PRM} are all PROMELALIGHT actions on global level, and the transitions in $\rightarrow_{\text{PRM}}$ are described by the SOS rules in Fig. 6.14 and Fig. 6.15.

6.3.3 PROMELALIGHT LTL

To express goals over PROMELALIGHT specifications, we use LTL formulae. As in HELENALIGHT, we restrict the atomic propositions of LTL formulae to state label expres-

sions of the form $pt[n]@label$. However, we explicitly exclude the state label expression $pt[n]@\text{start}_n$ from the set of atomic propositions. The state label start_n : **true** will later on be used in PromelaLight process expression as markers for **goto** jumps to allow recursion and should therefore not be considered as an atomic proposition. LTL formulae are built over these propositions as explained in Def. 4.1 on page 67.

Def. 6.27: PromelaLight LTL

Let PrmSpec be a PromelaLight *specification. The* set $AP(PrmSpec)$ of atomic propositions for *PrmSpec consists only of state label expressions.*
A state label expression *is of the form $pt[n]@label$ where pt is a process type declaration of PrmSpec, $n \in \mathbb{N}^+$ is the identifier of a process instance and $label \neq start_n$ is a state label in a process declaration of PrmSpec.*
A PromelaLight LTL formula for *PrmSpec is an LTL formula over the set $AP(PrmSpec)$ of the atomic* PromelaLight *propositions.*

Satisfaction: As for full Helena and HelenaLight, we rely on Def. 4.4 on page 68 which requires a set of atomic propositions and a satisfaction relation to define satisfaction of LTL formulae for a labeled transition system.

For a PromelaLight specification *PrmSpec*, the semantic rules of PromelaLight in Fig. 6.14 and Fig. 6.15 generate a labeled transition system $(S_{\text{PRM}}, I_{\text{PRM}}, A_{\text{PRM}}, \rightarrow_{\text{PRM}})$ for a given admissible initial state $\gamma_{init} \in I_{\text{PRM}}$. The atomic propositions for *PrmSpec* which are used to formulate LTL formulae are the state label expressions defined by the set $AP(PrmSpec)$. Therefore, it remains to define a satisfaction relation $\gamma \models p$ for $\gamma \in S_{\text{PRM}}$ and $p \in AP(PrmSpec)$.

Similarly to Helena and HelenaLight, a global PromelaLight state γ satisfies $pt[n]@label$, denoted by $\gamma \models pt[n]@label$, if there exists an active process instance of type pt with identifier n in γ and the next action performed by this process instance is the state label $label$: **true**. Formally, that means that in $\gamma = (\mathsf{ch}, \mathsf{proc})$ there exists $n \in dom(\mathsf{proc})$ with $\mathsf{proc}(n) = (pt, \beta, label : \textbf{true}; \pi)$.

6.4 Translation from HelenaLight to PromelaLight

Since we omit the component-based platform and any notion of data in HelenaLight, also the translation from HelenaLight to PromelaLight can be simplified. The translation only proceeds in two steps: Firstly, we provide role-to-role communication facilities by creating a user-defined enumeration type in PromelaLight for all message types from HelenaLight. The enumeration type only provides symbolic names for the message types to the translation of role behaviors. These symbolic names are simple constants which do not influence verification. Thus, we do not have to consider them in the correctness proof in the following section. Secondly, for each role type and its corresponding role behavior declaration in HelenaLight, we create a process type in PromelaLight which reflects the execution of the role behavior. Thereby, the translation of the role behavior declaration is changed due to the simplification of guarded choice to nondeterministic choice and arbitrary process invocation to recursive role behavior invocation. Furthermore, role creation does not longer have to be requested from a component, but can directly spawn a new process. Role-to-role communication is additionally simplified since we omit any notion of data.

In the following, we will present translation functions for each of the steps. Everything notated in normal or bold font is pure PROMELA code, everything notated in italic font has to be evaluated to get PROMELA code. We assume given a HELENALIGHT specification $EnsSpec = (\Sigma, behaviors)$ with $\Sigma = (nm, roletypes, roleconstraints)$ being a HELENALIGHT ensemble structure.

6.4.1 Role-to-Role Communication Facilities

To reflect all message types of the ensemble specification *EnsSpec*, we declare an enumeration type in PROMELALIGHT, called **mtype**, whose constants denote all message names occurring in the role types of Σ. This enumeration type is the same as in the full translation (cf. Sec. 5.2.2.4). For the formal proof, the introduction of the enumeration type itself is not strictly needed and does not influence the equivalence proof since all symbolic names could also be replaced by the corresponding constants. It is introduced due to readability of the final PROMELALIGHT specification only.

The translation function $trans_{\text{msgs}}$ for message types is specified in Fig. 6.16 and just creates this enumeration type. Thereby, the set of all message types of a HELENALIGHT ensemble specification $EnsSpec = (\Sigma, behaviors)$ with $\Sigma = (nm, roletypes, roleconstraints)$ is given by:

$$msgs(EnsSpec) = \{msg \mid \exists rt \in roletypes(\Sigma).\ msg \in rtmsgs_{out}(rt) \vee msg \in rtmsgs_{in}(rt)\}.$$

$$trans_{\text{msgs}}(EnsSpec) = \textbf{mtype}\ \{\ \forall msg \in msgs(EnsSpec)\ .\ msgnm(msg)\ \}$$

Figure 6.16: Translation of role-to-role communication facilities in HELENALIGHT

The notation $\forall msg \in msgs(EnsSpec)\ .\ msgnm(msg)$ means that for every message type *msg* in the ensemble specification *EnsSpec* a symbolic name is created which is given by the name *msgnm* of the message type *msg*. We implicitly assume that the created symbolic names are separated by comma. Note that the curly braces do not express set braces, they are pure PROMELALIGHT braces which embrace the symbolic names constituting the enumeration type.

6.4.2 Behaviors of Roles

As for full HELENA, role types and their role behavior declaration are reflected by process type declarations in PROMELA. These process type declarations are responsible to execute the behavior prescribed by the corresponding role behavior declaration in HELENA. The translation function is specified in Fig. 6.17) and relies on additional channel declarations specified in Fig. 6.18. The translation function of a role behavior declaration differs from the translation for full HELENA in Sec. 5.2 in the following aspects: the if-then-else construct is omitted and arbitrary process invocation is simplified to recursive role behavior invocation only. Furthermore, role creation does not longer have to be requested from a component, but can directly spawn a new process. Role-to-role communication when sending and receiving messages is simplified since we omit any notion of data.

A role behavior declaration for a role type *rt* is translated to a process type in PROMELALIGHT with the same name. The process type has one channel parameter **self**

$$
\begin{array}{ll}
trans_{\text{role}}(\textbf{roleBehavior}\ rt = P) & = \textbf{proctype}\ rt(\textbf{chan self})\ \{ \\
 & \quad chandecls_{\text{proc}}(P) \\
 & \quad \text{start}_{rt} : \textbf{true}; \\
 & \quad trans_{\text{proc}}(P) \\
 & \} \\
 & \\
trans_{\text{proc}}(\textbf{quit}) & = \textbf{false} \\
trans_{\text{proc}}(a.P) & = trans_{\text{act}}(a); trans_{\text{proc}}(P) \\
trans_{\text{proc}}(P_1 + P_2) & = \textbf{if} :: trans_{\text{proc}}(P_1) :: trans_{\text{proc}}(P_2)\ \textbf{fi} \\
trans_{\text{proc}}(N) & = \textbf{goto}\ \text{start}_N \\
 & \\
trans_{\text{act}}(X \leftarrow \textbf{create}(rt)) & = \textbf{run}\ rt(X) \\
trans_{\text{act}}(Y!msgnm(X)) & = Y!msgnm, X \\
trans_{\text{act}}(?msgnm(X{:}rt)) & = \textbf{self}?msgnm, X \\
trans_{\text{act}}(label) & = label : \textbf{true}
\end{array}
$$

Figure 6.17: Translation of a role behavior declaration in HelenaLight

$$
\begin{array}{ll}
chandecls_{\text{proc}}(\textbf{quit}) & = \sqcup \\
chandecls_{\text{proc}}(a.P) & = chandecls_{\text{act}}(a); chandecls_{\text{proc}}(P) \\
chandecls_{\text{proc}}(P_1 + P_2) & = chandecls_{\text{proc}}(P_1) \sqcup chandecls_{\text{proc}}(P_2) \\
chandecls_{\text{proc}}(N) & = chandecls_{\text{proc}}(N) \\
 & \\
chandecls_{\text{act}}(X \leftarrow \textbf{create}(rt)) & = \textbf{chan}\ X = [roleconstraints(rt)]\ \textbf{of}\ \{\textbf{mtype}, \textbf{chan}\} \\
chandecls_{\text{act}}(Y!msgnm(X)) & = \sqcup \\
chandecls_{\text{act}}(?msgnm(X{:}rt)) & = \textbf{chan}\ X \\
chandecls_{\text{act}}(label) & = \sqcup
\end{array}
$$

Figure 6.18: Channel declarations for a HelenaLight process expression

which reflects the input queue of the corresponding role type. The process expression defining the process type in PromelaLight starts with the declarations of all local variables which will be used throughout the process (cf. the function $chandecls_{\text{proc}}$ in Fig. 6.18). Namely, for each **create**-action $X \leftarrow \textbf{create}(rt)$ in the process expression P defining the role behavior, a local variable X of type **chan** is declared and initialized which will later on represent the **self** channel of the role to be created. For each message reception $?msgnm(X{:}rt)$, a local variable X of type **chan** is declared which will later on be used to store the reference X received with the message. Back in the translation of the role behavior declaration, a label $\text{start}_{rt} : \textbf{true}$ follows which is used to reflect recursive process invocation by **goto** jumps in PromelaLight. The label $\text{start}_{rt} : \textbf{true}$ must be unique, i.e., in the HelenaLight role behavior declaration for the role type rt, there does not exist a state label start_{rt}. Afterwards, the process expression P of the role behavior is translated into PromelaLight with $trans_{\text{proc}}(P)$.

The translation of a HelenaLight process expression is inductively defined over its structure. Termination with **quit** in HelenaLight is translated to the empty process **false** in PromelaLight. Action prefix $a.P$ in HelenaLight is translated the sequential composition of the translation $trans_{\text{act}}(a)$ of the action a and the translation

$trans_{\text{proc}}(P)$ of the process P. Nondeterministic choice $P_1 + P_2$ in HELENALIGHT is translated to nondeterministic choice with the **if**-construct in PROMELALIGHT. The process P_1 and P_2 are recursively translated to PROMELALIGHT. Process invocation N in HELENALIGHT is translated to a **goto** expression in PROMELALIGHT which jumps to the state label N in PROMELALIGHT. Since we only allowed recursive process invocation for at most the role type rt in a HELENALIGHT role behavior declaration for rt, this **goto** expression can only jump to the beginning start_{rt} : **true** of the process expression.

The translation of a HELENALIGHT action is inductively defined over its structure again. The creation of a new role instance X in HELENALIGHT for the role type rt is translated to running a new process instance of type rt in PROMELALIGHT. To reflect the implicit input queue of the newly created role instance, we declared and initialized a dedicated channel X with the capacity $roleconstraints(rt)$, which is retrieved from the associated ensemble structure, at the beginning of the PROMELALIGHT process type (cf. $chandecls_{\text{act}}$ in Fig. 6.18). This channel X is now given as the input parameter **self** to the new process instance in PROMELALIGHT. This channel is able to receive message consisting of a constant of an enumeration type (denoted by **mtype**) and a channel instance (denoted by **chan**). These two values reflect the message name and the role instance parameter of messages in HELENALIGHT. A send action in HELENALIGHT is translated to a send action in PROMELALIGHT where the message name *msgnm* and the role instance parameter X are now only separated by a comma. Note that while Y is a reference to a role instance in HELENALIGHT, it is a reference to a channel instance in PROMELALIGHT. For a receive action in HELENALIGHT, we declared a channel variable X at the beginning of the PROMELALIGHT process type (cf. $chandecls_{\text{act}}$ in Fig. 6.18) representing the role instance parameter X. To actually receive the message in PROMELALIGHT, we wait for a corresponding message with parameter X on the channel **self** which reflects the input queue of the current role instance. A state label in HELENALIGHT is directly translated to a state label in PROMELALIGHT.

6.4.3 Translation of an Ensemble Specification

In summary, a HELENALIGHT ensemble specification *EnsSpec* is translated to a PROMELALIGHT specification by translating all role behavior declarations from HELENALIGHT to PROMELALIGHT. Moreover, we add the enumeration type $trans_{\text{msgs}}(EnsSpec)$ created from all message types of the HELENALIGHT specification for readability issues as described before.

Def. 6.28: PROMELALIGHT Translation

Let $EnsSpec = (\Sigma, behaviors)$ be a HELENALIGHT *ensemble specification such that any role behavior $b \in behaviors$ does not contain a state label $start_b$. Its translation $trans(EnsSpec)$ to* PROMELALIGHT *is given by the set of all $trans_{role}(b)$ for $b \in behaviors$.*

Example: As an example, we consider the HELENALIGHT specification of the p2p example. The role types are given in Fig. 6.2, the ensemble structure in Fig. 6.4, and the behaviors of all roles in Fig. 6.6, Fig. 6.7, and Fig. 6.8. With the rules from Fig. 6.16, Fig. 6.17, and Fig. 6.18, the specification is translated to the PROMELALIGHT specification shown in Fig. 6.12.

6.4.4 Translation of Initial States

To be able to show semantic equivalence between HelenaLight and PromelaLight in Sec. 6.5, we additionally have to translate admissible initial states.

Def. 6.29: Translation of Initial States

Let σ be an admissible initial HelenaLight *ensemble state for a* HelenaLight *ensemble specification with local states $\sigma(i) = (rt, \emptyset[\textbf{\textit{self}} \mapsto i], \varepsilon, P)$ for all $i \in dom(\sigma)$.*
Its translation is the admissible initial PromelaLight *state $trans_{init}(\sigma) = (\mathsf{ch}, \mathsf{proc})$, such that*

(1) $dom(\mathsf{ch}) = dom(\sigma)$,

(2) $dom(\mathsf{proc}) = dom(\sigma)$, *and*

(3) for all $i \in dom(\mathsf{proc})$:
$\mathsf{proc}(i) = (rt, \emptyset[\textbf{\textit{self}} \mapsto c_i], chandecls_{proc}(P)\ start_{rt} : \textbf{\textit{true}}; trans_{proc}(P))$
with $c_i = ((\textbf{\textit{mtype}}, \textbf{\textit{chan}}), roleconstraints(rt), \varepsilon) \in dom(\mathsf{ch})$
and $c_i \neq c_j$ *for* $i \neq j$.

6.5 Correctness Proof

In this section, we sketch the proof of the correctness of the translation from HelenaLight to PromelaLight, i.e., that a HelenaLight specification and its PromelaLight translation satisfy the same set of $LTL_{\setminus \mathbf{X}}$ formulae. To this end, Thm. 6.5 provides a criterion for stutter trace equivalence of Kripke structures on which we can apply Thm. 6.6 entailing preservation of $LTL_{\setminus \mathbf{X}}$ formulae. In Sec. 6.5.1, we introduce a set of silent actions which are internal in PromelaLight and do not have any direct counterparts in HelenaLight. In Sec. 6.5.2, we define two relations $\sim$ and $\approx$ between the Kripke structures induced from a HelenaLight specification and its PromelaLight translation. In Sec. 6.5.3, we give an overview about the proof obligations to be able to apply Thm. 6.5. We have to show that

- the relation $\approx$ preserves satisfaction of atomic propositions,
- the relation $\approx$ is divergence-sensitive,
- any admissible initial state of a HelenaLight ensemble specification and its PromelaLight translation are related by the relation $\sim$,
- the relation $\sim$ is a $\approx$-stutter simulation of the Kripke structure of the HelenaLight specification by the Kripke structure of the PromelaLight translation, and
- the inverse relation $\approx^{-1}$ is a $\approx^{-1}$-stutter simulation in the other direction.

Having proven stutter trace equivalence, we can then apply Thm. 6.6 entailing preservation of $LTL_{\setminus \mathbf{X}}$. The full proof with all details can be found in Appendix A.

6.5.1 Silent Actions

To prove stutter trace equivalence of Kripke structures, we rely on the transitions of the LTSs which induce the Kripke structures. Thereby, PromelaLight introduces addi-

tional transitions compared to HelenaLight: Firstly, some preparations for declaring and initializing local variables are needed at the beginning of the translated role behavior in PromelaLight. Secondly, process invocation is reflected by several transitions in PromelaLight, i.e., a jump to the beginning of the role behavior with the action **goto** start_{pt}, the execution of the start state label with the action start_{pt} : **true**, and the execution of the first action of the invoked process. Both actions, the jump and the execution of the start state label, do not change satisfaction of atomic propositions. Thus, we consider these actions in PromelaLight as silent and denote them by τ.

Def. 6.30: Silent Action

(1) On the level of process types in PromelaLight*, we consider all actions of the form* ***chan*** *var,* ***chan*** *var* = [*const*] ***of*** {*typelist*}*,* $start_{pt}$: ***true*** *and* ***goto*** $start_{pt}$ *for all process types pt as* silent *and denote them by* τ*.*

(2) On the level of concurrent processes in PromelaLight*, we consider all (global) actions of the form i:****chan*** *var, i:****chan*** *var* = [*const*] ***of*** {*typelist*}*,* $i{:}start_{pt}$: ***true*** *and i:****goto*** $start_{pt}$ *for all process types pt as* silent *and denote them by* τ*.*

(3) All other PromelaLight *actions are considered to be* non-silent*.*

In the correctness proof, we need an additional notation concerning transitions.

Notation: The function $trans_{\text{act-global}}$ determines for each HelenaLight action on the ensemble-level the corresponding PromelaLight action on the level of concurrent processes and is defined as follows:

$$
\begin{array}{ll}
trans_{\text{act-global}}(i : X \leftarrow \textbf{create}(rt_j)) & = i : \textbf{run}\ rt_j(X) \\
trans_{\text{act-global}}(i : Y!msgnm(X)) & = i : Y!msgnm, X \\
trans_{\text{act-global}}(i :?msgnm(X{:}rt_j)) & = i : \textbf{self}?msgnm, X \\
trans_{\text{act-global}}(i : label) & = i : label : \textbf{true}
\end{array}
$$

6.5.2 Simulation Relations

We now define two relations which both express a correspondence between HelenaLight ensemble states and global PromelaLight states, but require a different level of correspondence. We will later on show that they define $\approx$-stutter simulations between the Kripke structures obtained from the labeled transitions systems of the semantics of HelenaLight and PromelaLight.

Def. 6.31: Relation $\sim$ and $\approx$

Let

- $K(T_{\text{HEL}}) = (S_{\text{HEL}}, A_{\text{HEL}}, \rightarrow^{\bullet}_{\text{HEL}}, F_{\text{HEL}})$ *be the induced Kripke structure of a* HelenaLight *ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *with* $\Sigma = (nm, roletypes, roleconstraints)$ *and*
- $K(T_{\text{PRM}}) = (S_{\text{PRM}}, A_{\text{PRM}}, \rightarrow^{\bullet}_{\text{PRM}}, F_{\text{PRM}})$ *be the induced Kripke structure of a* PromelaLight *specification.*

The relation $\sim \subseteq S_{\text{HEL}} \times S_{\text{PRM}}$ *is defined as follows:* $\sigma \sim \gamma = (\mathsf{ch}, \mathsf{proc})$ *if*

(1) $dom(\sigma) = dom(\mathsf{proc})$ *and*

(2) for all $i \in dom(\sigma)$ *with* $\sigma(i) = (rt_i, v_i, q_i, P_i)$ *and* $\mathsf{proc}(i) = (pt_i, \beta_i, \pi_i)$*:*

(a) $rt_i = pt_i$,

(b) $dom(v_i) \subseteq dom(\beta_i)$ *such that for all* $X \in dom(v_i)$*:*
$v_i(X) = j \Leftrightarrow \beta_i(X) = \beta_j(\mathbf{self})$ *(where* $\mathsf{proc}(j) = (pt_j, \beta_j, \pi_j)$*),*

(c) $q_i = msgnm_1(k_1) \cdot \ldots \cdot msgnm_m(k_m) \Leftrightarrow$
$\mathsf{ch}(\beta_i(\mathbf{self})) = (T, \kappa, (msgnm_1, \beta_{k_1}(\mathbf{self})) \cdot \ldots \cdot (msgnm_m, \beta_{k_m}(\mathbf{self})))$,
$T = (\mathbf{mtype}, \mathbf{chan})$, $\kappa = roleconstraints(rt_i)$ *and*
$\mathsf{proc}(k_j) = (pt_{k_j}, \beta_{k_j}, \pi_{k_j})$ *for all* $1 \leq j \leq m$,

(d) $\pi_i = trans_{proc}(P_i)$ *or*
$\pi_i = chandecls_{proc}(P_i)$ $start_{rt_i} : \mathbf{true}; trans_{proc}(P_i)$
with ***roleBehavior*** $rt_i = P_i \in behaviors$.

The relation $\approx \subseteq S_{\text{HEL}} \times S_{\text{PRM}}$ *is defined just as the relation* $\sim$ *with the exception of item (2d) which is replaced by*

(2d) $trans_{proc}(P_i) \xrightarrow{\tau^*} \pi_i$ *or*
$chandecls_{proc}(P_i)$ $start_{rt_i} : \mathbf{true}; trans_{proc}(P_i) \xrightarrow{\tau^*} \pi_i$
with ***roleBehavior*** $rt_i = P_i$.

Obviously, it holds that $\sim \subseteq \approx$.

Firstly, in the defined relations, there must be as many role instances in HelenaLight as process instances in PromelaLight. Secondly, the local state of each role instance i must be related to the local state of the process instance with the same identifier i: (a) The role type rt_i must match the process type pt_i. (b) The local variables in v_i must have counterparts in β_i, but note that the value types of HelenaLight and PromelaLight are subtly different. A local variable in HelenaLight points to a role instance whereas a local variable in PromelaLight points to a channel. More precisely, a local variable in HelenaLight points to the name of a role instance; in the PromelaLight translation the same variable points to the input channel of the corresponding process instance. Furthermore, note that vice versa, there might be local variables in β_i which do not have any counterparts in v_i. (c) The content of the input queue of the role instance must match the content of the corresponding channel of the process instance. As for local variables, the input queue of the role instance consists of role instance identifiers whereas the related PromelaLight input channel contains the identifiers of the input channels of the process instances (corresponding to these role instances). (d) For the process expression π_i occurring in the local state of the process instance, we either require that it is the same as the translation of the process expression P_i occurring in the local state of the role instance or that it adds declarations (and initializations) of all local variables and the start label $\text{start}_{rt_i} : \mathbf{true}$ to the translation of P_i if P_i is the process expression expressing the whole role behavior for the role type rt_i. The latter takes into account that the translation of a role behavior into PromelaLight adds the declaration of local variables and a start label at the beginning of the translated role behavior. For the relation $\approx$, we weaken both conditions such that π_i must only be reachable by evolution with arbitrary many τ actions.

6.5.3 Overview of Proof Structure

Table 6.2 summarizes the structure and the ideas how to prove that the induced Kripke structures of a HelenaLight specification and its PromelaLight translation satisfy the same set of $LTL_{\setminus \mathbf{X}}$ formulae. The full proof with all details can be found in Appendix A. The table lists all auxiliary lemmata and propositions with their proof obligations which we need to finally prove $LTL_{\setminus \mathbf{X}}$ preservation. We further indicate which lemmata and assumptions were necessary to prove the different proof obligations and how the syntax definitions and well-formedness conditions of HelenaLight ensemble specifications contribute.

The ultimate goal of the proof is to apply Thm. 6.6 which states that two stutter trace equivalent Kripke structures satisfy the same set of $LTL_{\setminus \mathbf{X}}$ formulae. Therefore, we show stutter trace equivalence of a HelenaLight specification and its PromelaLight translation according to Thm. 6.5 by establishing two $\approx$-stutter simulations with appropriate properties like property-preservation, divergence-sensitivity, and relating initial states.

Prop. A.8 shows that the relation $\sim$ is a $\approx$-stutter simulation of a HelenaLight specification by its PromelaLight translation. In Prop. A.10, we show that the relation $\approx^{-1}$ is a $\approx^{-1}$-stutter simulation in the other direction. For both propositions, an auxiliary lemma is needed: Lemma A.7 relates one step on the role type level of HelenaLight to the corresponding sequence of steps on the process type level of PromelaLight for the $\approx$-stutter simulation of HelenaLight by PromelaLight and vice versa Lemma A.9 for the $\approx^{-1}$-stutter simulation of PromelaLight by HelenaLight. The two lemmata are needed to be able to reason about transitions on ensemble level in HelenaLight based on transitions on role type level and about transitions on global level in PromelaLight based on transitions on process type level.

Prop. A.2 shows that the relation $\approx$ is property-preserving based on the auxiliary lemma Lemma A.1 which reasons about the relation of structure of process expressions in HelenaLight and PromelaLight.

Prop. A.5 shows that the relation $\approx$ is divergence-sensitive. To prove that lemma we need two auxiliary lemmata Lemma A.3 and Lemma A.4 which explain which transitions in HelenaLight and PromelaLight are stutter steps according to the relation $\approx$.

Prop. A.6 shows that any admissible initial state of a HelenaLight ensemble specification is related to its PromelaLight translation by the relation $\sim$.

Lastly, all five propositions Prop. A.2, Prop. A.5, Prop. A.6, Prop. A.8, and Prop. A.10 entail according to Thm. 6.5 that the induced Kripke structure of a HelenaLight specification and the induced Kripke structure of its PromelaLight translation are stutter trace equivalent as stated in Thm. A.11. Therefore, we can deduce in Cor. A.12 that they satisfy the same set of $LTL_{\setminus \mathbf{X}}$ formulae.

In the proofs, we need six assumptions which are all satisfied by the syntactic definitions and well-formedness conditions of HelenaLight except the last two:

(1) State labels are not the first actions of branches of nondeterministic choice constructs in HelenaLight role behavior declarations. This is a well-formedness condition of HelenaLight process expressions in Def. 6.11.

(2) Recursive process invocation may occur in a HelenaLight role behavior declaration rt at most for rt. This is a syntax restriction in Def. 6.12.

(3) Recursive process invocation may not occur immediately at the beginning of a HelenaLight role behavior declaration. This is a syntax restriction in Def. 6.12.

(4) All state labels in HelenaLight role behavior declarations are not of the form start_{rt}. This is an additional assumption which is not part of the syntax restriction or well-formedness conditions, but is required for stutter trace equivalence between HelenaLight and PromelaLight.

(5) The first action of a HelenaLight role behavior declaration is not a state label. This is an additional assumption which is not part of the syntax restriction or well-formedness conditions, but is required for stutter trace equivalence between HelenaLight and PromelaLight.

(6) In any nondeterministic choice in a HelenaLight role behavior declaration, process invocation is not one of the branches. This is an additional assumption which is not part of the syntax restriction or well-formedness conditions, but is required for divergence-sensitivity of the relation $\approx$.

Table 6.2: Overview of proof for $LTL_{\setminus \mathbf{X}}$ preservation between HelenaLight and PromelaLight

Lemma	Proof Obligation	Applied Lemmata	Assumptions	Assumptions satisfied by
Lemma A.1 on page 286	If $trans_{\text{proc}}(P) \xrightarrow{\tau^*} \pi$ and $label \neq \text{start}_{rt}$, then $P = label.P'$ iff $\pi = label : \mathbf{true}; \pi'$.		state label: not first action of nondet. choice branches	Def. 6.11
			recursive process invocation at most for rt	Def. 6.12
			state label: not first action of role behavior declarations	—
Prop. A.2 on page 289	If $\sigma \approx \gamma$, then $F_{\text{HEL}}(\sigma) = F_{\text{PRM}}(\gamma)$.	Lemma A.1	state label: not first action of nondet. choice branches	Def. 6.11
			recursive process invocation: at most for rt	Def. 6.12
			no state label start_{rt} in role behavior declarations	—
			state label: not first action of role behavior declarations	—
Lemma A.3 on page 291	If $\sigma \approx \gamma$ and $\sigma \xrightarrow{a}_{\text{HEL}} \sigma'$, then $\sigma' \not\approx \gamma$.		state label: not first action of nondet. choice branches	Def. 6.11
			no state label start_{rt} in role behavior declarations	—
			state label: not first action of role behavior declarations	—
Lemma A.4 on page 294	If $\sigma \approx \gamma$ and $\gamma \xrightarrow{a}_{\text{PRM}} \gamma'$, then $\sigma \approx \gamma'$ iff $a = \tau$.			
Prop. A.5 on page 300	$\approx$ is divergence-sensitive for $K(T_{\text{HEL}})$ and $K(T_{\text{PRM}})$.	Lemma A.3, Lemma A.4	recursive process invocation: not immediately	Def. 6.12
			no state label start_{rt} in role behavior declarations	—
			state label: not first action of role behavior declarations	—
			process invocation: not a branch in nondet. choice	—
Prop. A.6 on page 302	For any admissible initial state σ of a HelenaLight ensemble specification, it holds that $\sigma \sim trans_{\text{init}}(\sigma)$.			

Table 6.2 continued: Overview of proof for $LTL_{\setminus \mathbf{X}}$ preservation between HelenaLight and PromelaLight

Lemma	Content	Dependencies	Assumptions	Assumptions satisfied by
Lemma A.7 on page 303	If $P \overset{a}{\hookrightarrow} P'$, then $trans_{\text{proc}}(P) \xrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(P')$.		no state label start_{rt} in role behavior declarations	—
Prop. A.8 on page 307	$\sim$ is a $\approx$-stutter simulation of $K(T_{\text{HEL}})$ by $K(T_{\text{PRM}})$.	Lemma A.4, Lemma A.7	no state label start_{rt} in role behavior declarations	—
Lemma A.9 on page 317	If $\pi \overset{\tau^*}{\hookrightarrow} \pi' \xrightarrow{trans_{\text{act}}(a)\setminus\tau} \pi''$ and $\pi = trans_{\text{proc}}(P)$, then $P \overset{a}{\hookrightarrow} P'$ and $\pi'' = trans_{\text{proc}}(P')$.		no state label start_{rt} in role behavior declarations	—
Prop. A.10 on page 322	$\approx^{-1}$ is a $\approx^{-1}$-stutter simulation of $K(T_{\text{PRM}})$ by $K(T_{\text{HEL}})$.	Lemma A.4, Lemma A.9	no state label start_{rt} in role behavior declarations	—
Thm. A.11 on page 328	$K(T_{\text{HEL}})$ and $K(T_{\text{PRM}})$ are stutter-trace equivalent.	Thm. 6.5, Prop. A.2, Prop. A.5, Prop. A.6, Prop. A.8, Prop. A.10	state label: not first action of nondet. choice branches	Def. 6.11
			recursive process invocation: at most for rt	Def. 6.12
			recursive process invocation: not immediately	Def. 6.12
			no state label start_{rt} in role behavior declarations	—
			state label: not first action of role behavior declarations	—
			process invocation: not a branch in nondet. choice	—
Cor. A.12 on page 329	$K(T_{\text{HEL}}) \models \phi \Leftrightarrow K(T_{\text{PRM}}) \models \phi$.	Thm. 6.6, Thm. A.11	state label: not first action of nondet. choice branches	Def. 6.11
			recursive process invocation: at most for rt	Def. 6.12
			recursive process invocation: not immediately	Def. 6.12
			no state label start_{rt} in role behavior declarations	—
			state label: not first action of role behavior declarations	—
			process invocation: not a branch in nondet. choice	—

6.6 Correctness of the Full Translation

The correctness proof of the simplified translation is extended to the full translation. In the following, we first summarize which are the main extension points in the proof to reflect full HELENA. Afterwards, we informally explain for each of the main propositions, how the proof is extended to full HELENA.

6.6.1 Main Extension Points of the Proof

To recap the additional features which HELENA offers in contrast to HELENALIGHT, we summarize the main points of Table 6.1 once again: (1) Full HELENA extends the role-based model of HELENALIGHT by components which adopt roles and serve as computing and storage resources. (2) Data can be stored on components and roles as well as exchanged by messages between roles. Based on the data, we also introduce an if-then-else construct selecting between branches based on guards opposed to nondeterministic choice. (3) Messages allow lists of role instance parameters and data parameters instead of a single role instance parameter. (4) The number of allowed instances per role type is restricted by a minimal and maximal count. (5) Arbitrary process invocation is allowed instead of recursive role behavior invocation only.

For the correctness proof, these additional features mainly require to introduce more silent actions and particular data structures to include data. Some steps have to be composed to an atomic sequence of actions to particularly reflect the semantics of nondeterministic choice and the if-then-else construct. Furthermore, **goto**-jumps to the beginning of a translated role behavior have to be generalized to jumps to arbitrary points in a translated role behavior. (1) More specifically, some additional steps are required in PROMELA to realize role-to-component communication for role creation, retrieval, and termination as well for access of component attributes and for component operation calls, e.g., packing and unpacking a variable of type "Op" and exchanging messages between the role and the component (cf. Fig. 5.10 and Fig. 5.16). These steps are considered as silent actions in the proof which just execute some auxiliary steps needed in PROMELA in contrast to HELENA to finally progress the state of the overall PROMELA system similarly to the ensemble state in HELENA. However, we make all actions executed by the component process atomic by using the **atomic**-block of PROMELA such that they can be considered as one step and therefore do not influence the formal proof. (2) Stored data in attributes of components and roles is translated to values of parameters and local variables in PROMELA (cf. Fig. 5.10 and Fig. 5.12). This just extends the representation of states and any changes to the stored data directly results in changes in the state. To send and receive messages containing data, we extend the transferred message by more parameters to describe messages with a payload. To represent the HELENA if-then-else construct in PROMELA, we use the nondeterministic **if**-construct in PROMELA where the first statement is the translated guard and the following statements are the translated process expressions. To guarantee atomicity of the evaluation of the guard and the execution of the first action, we encapsulate the whole **if**-construct in PROMELA in an **atomic**-block. Furthermore, we have to make the assumption that the first action of each branch has to be executable (cf. Sec. 5.1). Without this assumption, the PROMELA specification could select a branch which would later on not be executable while the HELENA specification would immediately single out this non-executable branch (cf. Sec. 5.1 for an explanation of this problem). (3) To allow lists of parameters in message exchange, we extend each message by a fixed number of parameters reflecting the maximum number of parameters which a message has in

the HELENA specification. If the actual message to be exchanged has fewer parameters than the maximum number, dummy parameters are introduced. (4) The restriction on the number of allowed role instances is expressed in HELENA by side-conditions of the semantic rules for role creation and the **quit** process (cf. Fig. 3.3). In PROMELA, the restriction is expressed by the same boolean expression used as a statement prefixing spawning a new role process and quitting to play a role in the component process (cf. Fig. 5.10). Thereby, we exploit the fact that a boolean expression as a statement can only be executed if it evaluates to **true**. For the proof, this means that the boolean expression is considered as a silent action which establishes the side-condition of the semantic rule in HELENA. (5) Lastly, we allow arbitrary process invocations by inlining the translated process declarations into the translated role behavior and jumping to the beginning of the translated process declaration with a **goto**-statement for every process invocation (cf. Fig. 5.13). This is a straightforward extension of the idea of recursive role behavior invocation which we allowed in HELENALIGHT. Therefore, in the proof of the full translation, arbitrary process invocation is handled analogously to recursive role behavior invocation. However, this requires to generalize the condition that no role behavior declaration may start with a state label to process declarations, i.e., for the full proof, no role behavior declaration and no (local) process declaration in HELENA may start with a state label.

6.6.2 Overview of the Proof Structure

After having summarized the main extension points of the proof, we explain in this subsection how these extension points influence the proof of each main proposition.

Extended Simulation Relations: We first informally define the relations $\sim$ and $\approx$ for full HELENA based on Def. 6.31 for HELENALIGHT. Firstly, the mapping in both relations between the local states of roles in HELENA and the local states of the corresponding role processes in PROMELA must be extended to capture the additional features of roles in the full version of HELENA: In both relations $\sim$ and $\approx$ for the simplified versions of HELENA and PROMELA, the local environment function v for role instance variables in HELENALIGHT is mapped to the local environment function β for local variables in PROMELALIGHT. This mapping is extended such that also all values of role attributes and all data variables in HELENA are reflected by the local environment function β in PROMELA. Thereby, we take into account that the values in PROMELA are typed as channels for role instance variables and by one of the built-in PROMELA data types for role attributes and data variables. Furthermore, the local state of a role in HELENA also stores a reference to its owning component. In both relations $\sim$ and $\approx$, this reference must be reflected in the local state of the corresponding role process in PROMELA by a reference to the **self** channel of the owning component process. Secondly, both relations $\sim$ and $\approx$ must map the local states of components in HELENA to the local states of the corresponding component processes in PROMELA. We define the mapping analogously to roles in Def. 6.31, but extend it similarly as before with a mapping between the local environment functions for component attributes and associations.

Entire Set of Silent Actions: Furthermore, we have to assume some of the statements in the PROMELA translation as silent. We rigorously consider all statements in the component process as silent (cf. Fig. 5.10) as well as all statements in the role process (cf. Fig. 5.12, Fig. 5.13, and Fig. 5.16) except the last statement (possibly a whole

atomic block) of each translated HELENA action. All PROMELA statements before the last one can be considered silent since they do not change equivalence of the HELENA ensemble state and the corresponding global PROMELA state according to the extended relation $\approx$.

Entire Set of Assumptions: As listed in Table 6.2, we require a set of assumptions during the proof. These conditions apply to full HELENA as well while some have to be lifted to full HELENA:

- The following assumptions are either guaranteed by well-formedness conditions (cf. Def. 2.10): State labels are not the first action of any branch of nondeterministic choice or the if-then-else construct. Role behavior declarations and process declarations do not immediately call themselves. The last assumption also prohibits a chain of direct process invocations in which one process invocation occurs two times, e.g., N is called which immediately calls M and this immediately calls N again.
- Role behavior declarations do not contain any start labels start_N being N the name of a role type or a process. This assumption is not guaranteed by the syntax of HELENA or any well-formedness conditions, but it is needed to show property-preservation of the relation $\approx$.
- No role behavior declaration and also no process declaration may start with a state label. This assumption is not guaranteed by the syntax of HELENA or any well-formedness conditions, but it is needed to establish stutter trace equivalence.
- Process invocation is not allowed as one branch in any nondeterministic choice or if-then-else construct in a role behavior in the HELENA specification. This assumption is not guaranteed by the syntax of HELENA or any well-formedness conditions, but it is needed to establish divergence-sensitivity of the relation $\approx$.
- The **create**-action is only allowed as first action of a branch in nondeterministic choice or an if-then-else construct if the multiplicity of instances of the role type to be created is not yet exceeded and the owning component instance does not yet play the role. This assumption is not guaranteed by the syntax of HELENA or any well-formedness conditions, but it is needed to establish divergence-sensitivity of the relation $\approx$ (cf. Sec. 5.1 for a detailed explanation).
- The **get**-action is only allowed as first action of nondeterministic choice or an if-then-else construct if the requested owning component is guaranteed to currently adopt the requested role. This assumption is not guaranteed by the syntax of HELENA or any well-formedness conditions, but it is needed to establish divergence-sensitivity of the relation $\approx$ (cf. Sec. 5.1 for a detailed explanation).
- Sending a message is only allowed as first action of a branch of an if-then-else construct if the capacity of the message queue of the receiving role is not yet exceeded. This assumption is not guaranteed by the syntax of HELENA or any well-formedness conditions, but it is needed to establish divergence-sensitivity of the relation $\approx$ (cf. Sec. 5.1 for a detailed explanation).

With the extended relations $\sim$ and $\approx$, silent actions, and assumptions, we can now transfer the proofs of each main proposition to full HELENA:

Satisfaction of $LTL_{\setminus \mathbf{X}}$ Formulae in $\approx$-Equivalent States: Prop. A.2 and its auxiliary lemma Lemma A.1 show that $\approx$-equivalent states satisfy the same set of

$LTL_{\setminus \mathbf{X}}$ formulae, i.e, the relation $\approx$ is property-preserving. The proof can directly be transferred to full HELENA. Its argumentation is extended to the full set of HELENA process expressions and actions. It just requires to lift the assumptions to full HELENA and to extend the set of silent actions as explained before.

Divergence-Sensitivity of the Relation $\approx$: Prop. A.5 and its auxiliary lemmata Lemma A.3 and Lemma A.4 show that the relation $\approx$ is divergence-sensitive. Here again the proof can directly be transferred to full HELENA if the assumptions are lifted to full HELENA and the set of silent actions is extended. In the argumentation, special care has to be taken for nondeterministic choice and the if-then-else construct. As in HELENALIGHT, process invocation is not allowed as branch of nondeterministic choice which is now also extended to the if-then-else construct. Furthermore, we have to introduce some conditions when a **create**-action, a **get**-action or message reception are allowed as first action of a branch of nondeterministic choice or the if-then-else construct (cf. Sec. 5.1 for a detailed explanation of the conditions). These restrictions are necessary to avoid that the PROMELA translation can get stuck while the HELENA specification cannot. For a HELENA specification, a branch of nondeterministic choice is not selected for execution if its first action is not executable (and similarly for the if-then-else construct). In PROMELA, this HELENA action might be translated to several steps where the first one is executable, but a later one is not. Thus, in the PROMELA translation, a branch might be selected for execution which is actually not executable. This problem cannot be avoided by encapsulating all PROMELA steps into an indivisible sequence of actions with the PROMELA **atomic**-block because executability is still decided based on the first action of the **atomic**-block. Therefore, we have to restrict nondeterministic choice and the if-then-else construct as described above to guarantee divergence-sensitivity of the relation $\approx$.

$\sim$-Equivalence of Initial States: Prop. A.6 shows that an admissible initial state of a HELENA specification and its PROMELA translation can be related by the relation $\sim$. With a simple extension of the translation for initial states, the proof is trivial.

$\approx$-Stutter Simulation of HELENA Specifications: Prop. A.8 and its auxiliary lemma Lemma A.7 show that the relation $\sim$ is a $\approx$-stutter simulation of the induced Kripke structure of a HELENALIGHT specification by the induced Kripke structure of its PROMELALIGHT translation. The proof argumentation is again extended to the full set of HELENA process expressions and actions which again requires to lift the assumptions to full HELENA and to extend the set of silent actions as explained before. Two points in the proof require special care: Firstly, the if-then-else construct is translated to the nondeterministic **if**-construct in PROMELA where the first statement is a boolean expression and reflects the guard of the branch. Similarly to the realization of multiplicities of role instances in PROMELA, we again exploit the fact that a boolean expression as a statement can only be executed if it evaluates to **true**. For the proof, this means that the boolean expression is considered as a silent action which establishes the side-condition of the semantic rule for guarded choice in HELENA. Secondly, we rely on atomicity of the if-then-else construct and all actions of the component process to show that the PROMELA translation takes some silent steps compared to the original HELENA specification which does not change $\approx$-equivalence of states.

$\approx$-Stutter Simulation of Promela Translations: Prop. A.10 and its auxiliary lemma Lemma A.9 show that the relation $\approx^{-1}$ is a $\approx^{-1}$-stutter simulation of the induced Kripke structure of a Promela translation by the induced Kripke structure of its corresponding Helena specification. The proof argumentation is extended in the same points as the previous paragraph.

Lastly, we want to mention why we did not use the **atomic** keyword to make all translation of actions atomic, but rather introduced silent actions to hide additional steps in Promela. This would have led to a more scattered Promela specification while we wanted to keep it as clean and directly relatable to Helena as possible. Furthermore, the **atomic**-block only encapsulates actions into an indivisible sequence of actions, but the executability of the whole block is still decided based on the first action of the block.. However, to actually gain an advantage from the usage of the **atomic**-block, it would be nice if the Promela semantics and Spin allowed to check the executability of the **atomic**-block as a whole instead of just based on the first action.

6.7 Publication History

The idea and the structure for the correctness proof has already been presented for HelenaLight and its translation to PromelaLight in [HKW15]. This chapter augmented by the Appendix A presents the proof for HelenaLight in full detail. All theorems and auxiliary lemmata are precisely stated and proven correct. Particular care is taken for the edge case of divergence-sensitivity. For that reason, the translation had to be changed such that channel declarations are shifted to the beginning of the translated role behavior in PromelaLight.

The extension of the correctness proof to full Helena has already shortly been discussed in [Kla15b]. This chapter describes the extension in more detail, though not formally showing the correctness.

Chapter 7

Implementation
Vivifying HELENA with jHELENA

To realize ensemble-based systems following the role-based modeling approach HELENA, this chapter describes the prototypic implementation and execution framework jHELENA. It is realized in Java and transfers the concepts of roles and collaborations in ensembles to an object-oriented implementation. Roles are implemented as Java threads on top of a component. Role objects are bound to specific ensembles while components can adopt many roles in different, concurrently running ensembles.

The goal of the framework is twofold: jHELENA implements the structural and dynamic rules enforced by the formal modeling concepts of ensemble specifications and their semantics. It furthermore provides an interface for the developer to realize concrete ensemble-based applications according to the HELENA approach and to allow to execute them. To support both goals, the framework contains two layers, a `metadata` layer and a `developer interface`, and an orthogonal system manager.

- With help of the `metadata` layer, the developer can define ensemble structures. For that, the `metadata` classes must be instantiated by objects which represent the various kinds of types that can occur in an ensemble structure, like role types, message types, etc. Thus an ensemble structure is represented by a net of objects which are linked in accordance with the general rules for ensemble structures.
- The `developer interface` contains abstract base classes to implement concrete components, roles, messages, etc. They are related to the `metadata` classes by associations determining their types. The abstract classes of the `developer interface` must be extended by the developer to implement concrete ensembles in accordance with a particular ensemble structure (defined on the `metadata` level). Most importantly, for each concrete role class the behavior of the instances of that role must be realized. The framework prescribes that any role instance is an active object implemented as a thread whose `run`-method executes the role behavior.
- The system manager is responsible to instantiate ensemble structures, to create the underlying component-based platform, and to create and run ensembles on top of it. For a particular ensemble-based application, this class has to be extended to prescribe the contributing types of the ensemble structure, the particular components forming the component-based platform, and the ensemble to be run on top of that.

In the following, we first present the architecture of the jHELENA framework in Sec. 7.1. The next three sections, Sec. 7.2, Sec. 7.3, and Sec. 7.4, cover the two layers and

the abstract system manager of the framework in detail. To illustrate the application of the framework, Sec. 7.5 walks through the implementation of the p2p example. We finish with a discussion of related work in Sec. 7.6 and concluding remarks in Sec. 7.8.

The complete implementation of the jHELENA framework can be found on the attached CD in the project `eu.ascens.helena`. The application of this framework to the p2p example is exercised in the project `eu.ascens.helena.p2p`.

7.1 Architecture

The jHELENA framework is implemented in Java and consists of two layers, the `metadata` layer and the `developer-interface`, which both are used by a system manager. Its architecture is shown in Fig. 7.1.

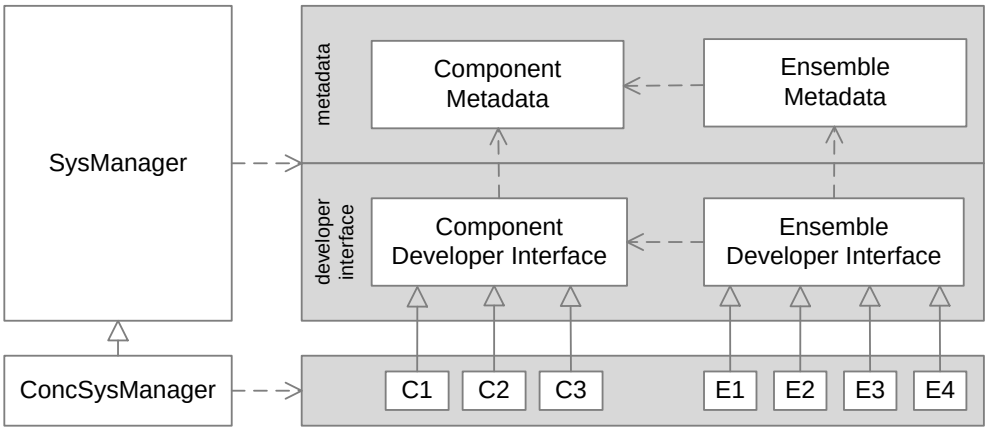


Figure 7.1: Architecture of the jHELENA framework

The `metadata` layer allows to define the meta model of ensemble specifications in terms of component types and ensemble structures (and thus role types etc.) according to the definitions in Chap. 2. Thereby, the ensemble-related parts built upon the component-related parts as indicated by the dependency arrow from left to right, e.g., to define the possible owning component type of a role type. The classes of this layer provide the means to describe the structural aspects of an ensemble structure, the internals of the framework guarantee that all syntactical restrictions from Chap. 2 are respected during the definition of an ensemble structure. For example, this layer takes care that a role type $rt = (rtnm, rtcomptypes, rtattrs, rtmsgs_{out}, rtmsgs_{in})$ is reflected in jHELENA by a role type class which is associated to a set of component type classes which can play this role, to a set of role-specific attribute classes, and to a set of outgoing and incoming message classes for message exchange. Other associations are forbidden by the framework.

The `developer-interface` provides the basic functionality to realize an actual ensemble-based application and implements the execution semantics of HELENA. The developer extends the abstract base classes of this layer to implement concrete components, indicated by `C1,C2,C3`, as well as concrete ensembles including roles and their behavior, indicated by `E1,E2,E3,E4`. As in the previous layer, the ensemble-related parts built upon the component-related parts as indicated by the dependency arrow from left to right. Furthermore, this layer provides the means

to describe the dynamic aspects of an ensemble. A further set of classes allows to define role behaviors according to the syntax of HELENA ensemble specifications in Chap. 2. By relying on these classes, the developer is forced to adhere to the basic structure of role behaviors and it is guaranteed that the execution semantics of HELENA is preserved.

The system manager and its concrete, application-dependent extension are responsible for the configuration of the component-based platform and ensemble structures, the creation of initial ensemble states, and the launch of concrete ensemble-based applications running ensembles concurrently on top of the component-based platform. Most certainly, the system manager thereby relies on the (component-related and ensemble-related) infrastructure of the jHELENA framework. Extending the abstract `SysManager` class guides the developer through its main activities: The developer first needs to implement the method `configureTypes()` (cf. Fig. 7.2) which configures all structural types for the application, i.e., component types and operation types, role types and message types as well as ensemble structures formed from role types relying on component types. Afterwards, the method `createComponents()` initializes all component instances providing the component-based platform for the application-specific ensembles. Lastly, the initial state of each ensemble is established and the ensembles themselves are launched in the method `startEnsembles()`. With this method, many concurrently running ensembles can be started one after the other.

In following sections, we discuss in detail how the formal definitions of ensemble structures are realized in the `metadata` layer of jHELENA, which infrastructure the `developer-interface` provides for the implementation of actual ensemble-based applications, and how the system manager handles the initialization of concrete ensemble-based applications. The complete implementation of the jHELENA framework can be found on the attached CD in the project `eu.ascens.helena`.

7.2 Metadata Layer

The upper package of Fig. 7.2 gives an overview of the `metadata` layer. All types used to provide a component-based platform and to build ensemble structures are realized by corresponding `metadata` classes; the relationships between types are represented by associations in the `metadata` layer of the jHELENA framework. Hence, this layer defines the meta model of a component-based platform and the ensemble structures building on top of it. Concrete instances of classes on this layer represent the types contributing to the ensemble-based system (and not the actual instances of the types). We walk through all classes of this layer in unison with their counterparts of the formal HELENA syntax.

7.2.1 Component-Based Platform

The main element forming the component-based platform in HELENA is a component type $ct = (ctnm, ctattrs, ctassocs, ctops)$. Abstractly, such a component type is represented in jHELENA by the class `ComponentType`:

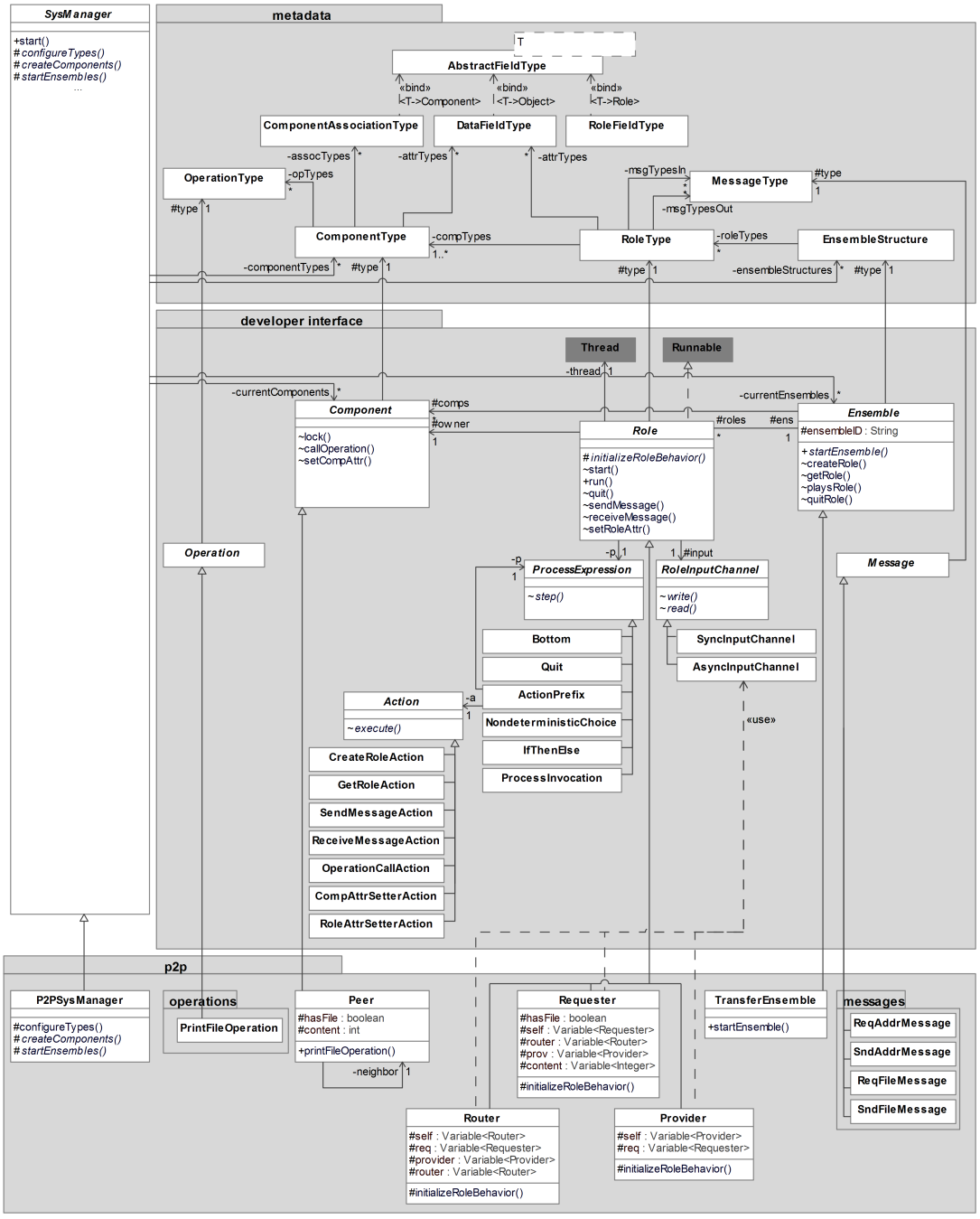


Figure 7.2: jHELENA framework and its application to the p2p example (for readability some associations are omitted and only mentioned in the following descriptions, e.g., for the class `NondeterministicChoice`)

- The name *ctnm* of the component type *rt* is stored in an attribute `name` of the class `ComponentType` (not shown in the diagram). The attribute `name` has the type `Class<? extends Component>`. This ensures, using the reflection mechanism of Java, that only those objects of the class `ComponentType` can be created whose `name` attribute refers to a component class extending the abstract class `Component` of the `developer-interface` (cf. Sec. 7.3).

- The component type attributes *ctattrs* are determined by the association with end

`attrTypes` directed from the class `ComponentType` to the class `DataFieldType`. The super class `AbstractFieldType<T>` of the class `DataFieldType` represents abstract fields (like known as attributes of Java classes) which have a name and a certain type `T`. The class `DataFieldType` instantiates the type parameter `T` by the class `Object` such that an instance of the class `DataFieldType` represents an arbitrary data attribute.

- Similarly, the component association types *ctassocs* of the component type *ct* are determined by the association with the end `assocTypes` directed from the class `ComponentType` to the class `ComponentAssociationType`. This class instantiates the type parameter `T` of its super class `AbstractFieldType` by the class `Component` such that an instance of the class `ComponentAssociationType` represents only associations to other components.
- Finally, the operation types *ctops* provided by the component type *ct* are determined by the association with the end `opTypes` directed from the class `ComponentType` to the class `OperationType`. To represent an operation type $op = opnm(\overrightarrow{x : dt})$, the class `OperationType` stores the name of the operation type in an attribute `name` of type `Class<? extends Operation>` analogously to the name of a component type. Furthermore, it requires a list $\overrightarrow{x}$ of data parameters represented by an ordered list of instances of the class `DataFieldType` (the association from `OperationType` to `DataFieldType` is not shown in Fig. 7.2).

Particular component types are represented by objects of the class `ComponentType`. They are constructed with the static factory method `createType` of the class `ComponentType` (not shown in the diagram) such that the actual parameters point to objects representing the constituent parts of a component type like its association types.

7.2.2 Ensemble Structures

Components team up in ensembles to perform certain tasks. Each participant of an ensemble is described by a role type. The class `RoleType` represents such a role type $rt = (rtnm, rtcomptypes, rtattrs, rtmsgs_{out}, rtmsgs_{in})$:

- The name *rtnm* of the role type *rt* is stored in an attribute `name` of the class `RoleType` (not shown in the diagram). Analogously to the attribute `name` of the class `ComponentType`, this attribute has the type `Class<? extends Role>`. This ensures, using the reflection mechanism of Java, that only those objects of the class `RoleType` can be created whose `name` attribute refers to a role class extending the abstract class `Role` of the `developer-interface` (cf. Sec. 7.3).
- The set *rtcomptypes* of component types, which are able to adopt the role type *rt*, is represented by an association with end `compTypes` directed from the class `RoleType` to the class `ComponentType` which was already described before.
- Similarly to component attributes for component types, the role type attributes *rtattrs* of a role type *rt* are determined by the association with end `attrTypes` directed from the class `RoleType` to the class `DataFieldType`.
- Lastly, the sets of message types $rtmsgs_{out}$ and $rtmsgs_{in}$ representing outgoing and incoming messages supported by the role type *rt* are modeled as associations with end `msgTypesOut` and `msgTypesIn` directed from the class `RoleType` to the class `MessageType`. Analogously to the representation of operation types, a message type $msg = msgnm(\overrightarrow{X : rt})(\overrightarrow{x : dt})$ is represented by the class `MessageType`.

The class stores the name of the message type in an attribute `name` of type `Class<? extends Message>`. The list $\overrightarrow{X}$ of role instance parameters is represented by an ordered list of instance of the class `RoleFieldType`, the list $\overrightarrow{x}$ of data parameters by an ordered list of instances of the class `DataFieldType` (both associations from `MessageType` are not shown in Fig. 7.2).

Particular role types used in an ensemble structure are represented by objects of the class `RoleType`. They are constructed with the static factory method `createType` of the class `RoleType` (not shown in the diagram) such that the actual parameters point to objects representing the constituent parts of a role type.

Several role types collaborate in an ensemble. The structural relationships between the collaborating role types are described by an ensemble structure $\Sigma = (nm, roletypes, roleconstraints)$ which is represented in jHELENA by an object of the corresponding class `EnsembleStructure`. The set *roletypes* of role types contributing to the ensemble is determined by the association with end `roleTypes` directed from the class `EnsembleStructure` to the class `RoleType`. The role constraints *roleconstraints* furthermore determine how many instances per role type may and have to contribute to the ensemble and how many messages the input queue of each role type can store. The multiplicity constraints are represented by an additional parameter of the list of role types of the class `EnsembleStructure`. However, the capacity of the input queue is stored per role type (not shown here). Again, particular ensemble structures are represented by objects of the class `EnsembleStructure`. They are constructed with the static factory method `createType` of the class `EnsembleStructure` (not shown in the diagram) such that the actual parameters point to objects representing the constituent parts of an ensemble structure.

7.3 Developer Interface

The goal of the `developer interface` is to facilitate the implementation of concrete ensemble applications by providing abstract base classes for all formal HELENA concepts and to guarantee that the execution semantics of HELENA is preserved by these abstract classes. In contrast to the `metadata` layer, concrete instances of classes on this layer represent actual instances of HELENA types. For instance in our p2p example, the role type `Router` is a type instance on the `metadata` layer while the router with ID 1 is an instance on the `developer interface` layer whose type is defined by the role type `Router`.

In the following, we divide the set of classes provided by the `developer interface` into two categories: classes to represent the instances of component types, role types and ensemble structures and their current state according to the HELENA semantics (cf. Sec. 3.2) and classes to represent role behaviors and their execution according to the structural operational semantic rules of HELENA (cf. Sec. 3.3).

7.3.1 Instances and Their Current State

The first part concentrates on classes which represent the HELENA instances of the HELENA types. As depicted in Fig. 7.2, the `developer interface` offers the abstract classes `Component`, `Role`, and `Ensemble` for the corresponding `metadata` classes; the subclasses of `AbstractFieldType` do not need any counterparts in the `developer interface` since attribute instances are implicitly represented by Java instance variables and their values associated to component and role instances. Each abstract class has an association with

end `type` to the corresponding `metadata` class such that the type of an instance can be determined.

As already mentioned, instances of the classes of the `metadata` layer represent HELENA types while instances of the classes of the `developer interface` represent HELENA instances like role instances. These two layers are necessary to guarantee compliance of an ensemble-based system developed in jHELENA with the formal rules and restrictions of the HELENA modeling approach. The `metadata` layer prescribes how the different HELENA types can or must be connected. The `developer interface` takes care for the definition and execution of the dynamic behavior of an ensemble-based system. Without the `metadata` layer, all type information would have to be encoded into the `developer interface`. For example, the class `Role` would have to provide static fields to store the allowed owning component types or the supported message types. Extracting this static information about the relationships between types to the `metadata` layer allows to focus with the `developer interface` just on the dynamic aspects of an ensemble, i.e., its current state and its execution.

The relationships between the abstract classes characterize the state of an ensemble according to the formal HELENA semantics in Chap. 3.

- According to Def. 3.4 on page 50, the global state σ of an ensemble is defined as a pair $(comps, roles)$ where *comps* is a function mapping each component instance contributing to the ensemble to its local state and *roles* is a function mapping each role instance participating in the ensemble to its local state. In jHELENA, an ensemble is represented by an instance of the class `Ensemble`. The set *comps* of component instances contributing to the ensemble is given by the association with end `comps` directed from the class `Ensemble` to the class `Component`. The set *roles* of role instances currently participating in the ensemble is given by the association with end `roles` directed from the class `Ensemble` to the class `Role`.

- According to Def. 3.2 on page 46, the local state of a component instance is defined as a tuple (ct, at^c, as) where ct is the component type of the instance, at^c is a function mapping attributes of the component type ct to values, and as is a function mapping component associations of the component type ct to component instances. In jHELENA, a component instance is represented by an instance of the class `Component`. The component type ct of the instance can be accessed via the association with end `type` from the `Component` to the class `ComponentType`. The attribute values at^c and the component association values as of the component instance are given by the current values of the instance variables of the concrete component class later on (cf. Sec. 7.5).

- According to Def. 3.3 on page 48, the local state of a role instance is defined as a tuple $(rt, ci, at^r, v, w, q, P)$ where rt is the role type of the instance, ci is the owning component instance of the role instance, at^r is a function mapping attributes of the role type rt to values, v is a function mapping role instance variables to values, w is a function mapping data variables to values, q is the current content of the input queue of the role instance, and P is process expression representing the current control state of the role instance. In jHELENA, the role type rt of the instance can be accessed via the association with end `type` from the class `Role` to the class `RoleType`. The association with end `owner` navigates to the unique component instance ci which currently adopts this role. The attribute values at^r of a role instance are given by the current values of the instance variables of the concrete role class later on (cf. Sec. 7.5). Similarly, the values of role

instance variables and data variables are implicitly given by the current values of the instance variables of the concrete role which are accessed when the role behavior is executed. The input queue q is represented by the association with end `input` from the class `Role` to `RoleInputChannel`. The current control state P of each role instance is represented by the association with end `p` from the class `Role` to the class `ProcessExpression`.

To implement a concrete ensemble application, the abstract classes of the `developer interface` must be extended by concrete subclasses as indicated by the inheritance arrows to the lower `p2p` layer in Fig. 7.2. The framework ensures, using Java reflection, that concrete subclasses and the attributes of concrete component and role classes fit to an ensemble structure represented by type instances on the `metadata` level.

7.3.2 Evolution of Instances

The second part of the `developer interface` concentrates on classes which represent the description of role behaviors and allow their execution in unison with the HELENA semantics.

7.3.2.1 Description of Role Behaviors

As depicted in Fig. 7.2, the `developer interface` offers the abstract classes `ProcessExpression` and `Action` together with their concrete subclasses to describe role behaviors. The subclasses of the class `ProcessExpression` represent all constructs of process expressions according to Def. 2.9 on page 25; one additional subclass `Bottom` represents the semantic extension $\perp$ describing that a role finished its role behavior (cf. Sec. 3.2.2). For these subclasses, Fig. 7.2 only shows the associations for the simplest class `ActionPrefix`. Similarly to the syntactic construct of action prefix in Def. 2.9, the class is characterized by an action (represented by the association with end `a` from the class `ActionPrefix` to the class `Action`) and the following process expression (represented by the association with end `p` from the class `ActionPrefix` to the class `ProcessExpression`). All other process expressions are analogously implemented (not all associations needed in the implementation are shown in Fig. 7.2). The subclasses of the class `Action` represent all actions according to Def. 2.9. To describe the actions, the classes `Operation` and `Message` are additionally used to represent concrete operation calls and exchanged messages.

The class `Role` offers an abstract method `initializeRoleBehavior` which must be implemented by concrete subclasses, i.e., concrete roles. The implementation of this method uses the hierarchy of classes for process expressions and actions, concrete message and operation classes as well as role instance and data variables to define the role behavior of a certain role (cf. Sec. 7.5).

7.3.2.2 Execution of Role Behaviors

The execution of the role behavior defined by the method `initializeRoleBehavior` is started with the method `start` of the class `Role`. The method is responsible to initialize and start a new `Thread` (depicted in Fig. 7.2 by the association with end `thread`). In this thread, the method `run` of the class `Role` is then executed. With that, a role instance becomes an active entity which executes its role behavior in parallel to all other currently existing role instances. The method `run` takes care to continuously execute one step of the role behavior until the role behavior has been terminated and the role

has been quit, i.e., if the role behavior in HELENA is infinite, it will also be infinitely be executed in jHELENA. This corresponds to taking one path, projected to a role instance, through the semantic labeled transition system of the ensemble-based system according to the semantics in Chap. 3. Each step represents the evolution of the ensemble by one operational semantic rule on the ensemble-level (cf. Sec. 3.3), projected on a particular role instance. The implementation of the method `run` is shown in Fig. 7.3. The method

```
1  public final void run() {
2    try {
3      while (!(this.p instanceof Bottom)) {
4        try {
5          this.p = this.p.step(this);
6        }
7        catch (ActionCurrentlyNotExecutableException e) {
8          this.log.fine(e.toString());
9          Thread.sleep(10);
10       }
11     }
12   }
13   catch (ActionNeverExecutableException
14    | WellFormednessViolatedException
15    | GuardNeverEvaluableException
16    | InvokedProcessNotSetException
17    | InterruptedException e) {
18     this.log.severe(e.toString());
19     this.quit();
20   }
21 }
```

Figure 7.3: jHELENA implementation of the method `run` of the class `Role`

implements a `while`-loop (line 3–11) which continuously evolves the process expression `p`, representing the role's current control state, by one step (line 5). If no exceptions occur, the `while`-loop evolves the process expression until the process expression is an instance of the class `Bottom` (line 3). This class represents the semantic extension $\perp$ of process expressions (cf. Sec. 3.3) which cannot evolve anymore and thus describes that a role finished its role behavior and has been quit. During the execution of one step, exceptions may occur:

- If an exception of type `ActionCurrentlyNotExecutableException` occurs (line 7), the evolution of the process expression is currently not possible, but at a later point of execution, it might become executable again. Therefore, this exception is caught and after some delay, evolution of the process expression is tried again (line 7–10). Examples for that are that a role can currently not be created since too many instances of the desired role type exist or a message can currently not be received since there is no message in the input queue.
- If any other exception occurs, the process expression will never become executable again or some assumptions are violated such that the evolution of the process expression has to be abnormally terminated (line 13–20). Each caught exception can have different reasons: The exception `ActionNeverExecutableException` is raised whenever the next action to be executed will never be executable, e.g., a message should be received, but another message is the first item in the input queue of the role. The exception `WellFormednessViolatedException` is raised if some syntactic well-formedness condition of process expressions according to Def. 2.10 on page 28 was violated, e.g., a message was sent which is actually not supported as outgoing

message by the sending role. The exception `GuardNeverEvaluableException` is raised if the guard of an if-then-else construct was not well-formed according to Def. 2.10, e.g., a role attribute was requested which was not declared in the corresponding role type. The exception `InvokedProcessNotSetException` is thrown if a process was invoked, but internally the invoked process could not be determined. Finally, the exception `InterruptedException` is thrown whenever the execution of the role gets abnormally interrupted. In all these cases, the role behavior cannot be executed from this point on. Thus, an error is logged and the role is quit (line 19–20).

The subclasses of the class `ProcessExpression` and `Action` then implement the single step according to the semantic operational rules of the HELENA semantics in Sec. 3.3 on page 53. The rules for the evolution of process constructs in Fig. 3.1 on page 54 and for the process construct **quit** in Fig. 3.3 on page 57 are represented by the method `step` of each jHELENA counterpart of a process expression. The rules for the evolution of ensembles by particular actions in Fig. 3.3 on page 57, Fig. 3.4 on page 58, and Fig. 3.5 on page 60 are represented by the method `execute` of each jHELENA counterpart of an action. In the following, we walk through all process constructs and show the implementation of the method `step` of each jHELENA counterpart of a process expression. Furthermore, we discuss the implementation of the method `execute` of each jHELENA counterpart of an action when we reach the actual execution of an action (during action prefix).

Role Termination: According to the rule *quit* in Fig. 3.3 on page 57, role termination quits the current role and evolves the process expression representing the current control state to the semantics extension $\perp$. The corresponding jHELENA implementation of the method `step` in the class `Quit` is shown in the code snippet of Fig. 7.4. It quits the role executing the process expression by calling the method `quit` of the class `Role` (line 2) and returns the singleton instance of the class `Bottom` (line 3). The implementation of the method `quit` in the class `Role` thereby takes care to close the input channel of the role and to remove the role from the set `roles` of currently existing role instances which is linked to the class `Ensemble` by a corresponding association.

```
1  ProcessExpression step(Role source) {
2    source.quit();
3    return Bottom.getInstance();
```

Figure 7.4: jHELENA implementation of the method `step` of the class `Quit`

Action Prefix: According to the rule *action prefix* in Fig. 3.1 on page 54, action prefix simply evolves by executing the current action to the remaining process expression. In jHELENA, action prefix is described by the class `ActionPrefix` which represents the action to be executed by the association with end `a` to the class `Action` and the remaining process expression by the association with end `p` to the class `ProcessExpression`. The implementation of the method `step` in the class `ActionPrefix` uses these two associations to realize the semantics of action prefix. Its implementation is shown in the code snippet in Fig. 7.5. Most importantly, the method executes the action `a` by calling the method `execute` of the class `Action` in line 6 (the implementation of the method `execute` is discussed in the next paragraph) and returns the remaining process expression `p`

(line 17). However, it might happen that an exception is raised during the execution of the action. The implementation then takes care to categorize the raised exceptions: Either they violate the well-formedness conditions of process expressions in Def. 2.10 on page 28 (which raises an exception of type `WellFormednessViolatedException` in line 8–10) or the action is just currently not executable (which raises an exception of type `ActionCurrentlyNotExecutableException` in line 11–13) or the action will never be executable (which raises an exception of type `ActionNeverExecutableException` in line 14–16).

```
1  ProcessExpression step(Role source)
2       throws WellFormednessViolatedException,
3       ActionCurrentlyNotExecutableException,
4       ActionNeverExecutableException {
5    try {
6      this.a.execute(source);
7    }
8    catch (MessageNotAllowedAsInputException | ... e) {
9      throw new WellFormednessViolatedException(this.a, source, e);
10   }
11   catch (NoMessageException | ... e) {
12     throw new ActionCurrentlyNotExecutableException(this.a, source, e);
13   }
14   catch (RoleInputChannelClosed | ...s e) {
15     throw new ActionNeverExecutableException(this.a, source, e);
16   }
17   return this.p;
18 }
```

Figure 7.5: jHELENA implementation of the method `step` of the class `ActionPrefix`

An action is represented by the class `Action` with appropriate subclasses for all possible HELENA actions. The implementation of the method `execute` in each subclass realizes the semantics of the corresponding action of in Fig. 3.3 on page 57, Fig. 3.4 on page 58, and Fig. 3.5 on page 60. The implementation thereby always follows the same pattern: All static well-formedness criteria for the action according to Def. 2.10 on page 28 are checked if they are not yet guaranteed by the `metadata` layer. Then, the ensemble, the issuing role or the owning component is invoked with an appropriate method call to actually realize the action, e.g., for role creation the method `createRole` in the class `Ensemble` is called. In this method, the side-conditions of the semantic rules are checked and finally the action is taken to effect. In the following, we summarize for each HELENA action all well-formedness criteria and side-conditions which are checked and which method implements the actual effect of the action. We furthermore discuss which actions can interfere with each other and how atomicity is guaranteed:

A create action is represented in jHELENA by the class `CreateRoleAction` which maintains attributes for the type of the role instance to be created, the desired owning component, and the variable used to store the reference of the created role instance. Its method `execute` first checks the well-formedness criteria that the type of the role instance to be created is allowed for the underlying ensemble structure and the type of the given component instance is allowed as owner of the role instance to be created. Furthermore, it checks whether the desired owning component instance is not `null`. If a criterion is not satisfied, an appropriate exception is thrown. Afterwards, the method `createRole` of the class `Ensemble` is invoked which returns a reference to the created role instance if it could be created (otherwise, an exception is thrown). This reference is stored in the given variable.

The actual role creation according to the rule *create* in Fig. 3.3 is implemented by the method `createRole` of the class `Ensemble`. It first checks the side-conditions of the semantic operational rule that the given component instance is actually contributing to the ensemble (i.e., that it is contained in the set `comps` associated to the class `Ensemble`, cf. item (1) of the side-condition), that the given component instance does not yet adopt an instance of the desired role type (item (2)), and that the multiplicity bounds for the desired role type are not yet exceeded (item (3)). If a side-condition is not satisfied, an appropriate exception is thrown. Afterwards, the role instance is created, added to the set `roles` of the class `Ensemble` representing the currently existing role instances, and finally its behavior is started by calling the method `start` of the class `Role`.

The execution of this action can interfere with other role creations and role termination on the same owning component and in the same ensemble because the set of adopted roles for the component and/or the multiplicity of currently existing role instances in the ensemble is changed. Therefore, role creation as well as role termination have to acquire the single lock of the owning component and the owning ensemble to guarantee exclusive execution. Furthermore, role creation can interfere with plays-queries in guard of the if-then-else construct. Therefore, also plays-queries have to acquire the same lock of the owning component.

A get action is represented in jHELENA by the class `GetRoleAction` which maintains attributes for the type of the role instance to be retrieved, the desired owning component, and the variable used to store the reference of the retrieved role instance. Its method `execute` first checks the well-formedness criteria that the type of the role instance to be created is allowed for the underlying ensemble structure and the type of the given component instance is allowed as owner of the role instance to be created. Furthermore, it checks whether the desired owning component instance is not `null`. If a criterion is not satisfied, an appropriate exception is thrown. Afterwards, the method `getRole` of the class `Ensemble` is invoked which returns a reference to the retrieved role instance if it already exists (otherwise, an exception is thrown). This reference is stored in the given variable.

The actual role retrieval according to the rule *get* in Fig. 3.3 is implemented by the method `getRole` of the class `Ensemble`. It first checks the side-conditions of the semantic operational rule that the given component instance is actually contributing to the ensemble (i.e., that it is contained in the set `comps` associated to the class `Ensemble`, cf. item (1) of the side-condition) and that the given component instance adopts an instance of the desired role type (item (2)). If a side-condition is not satisfied, an appropriate exception is thrown. Afterwards, the role instance is returned.

The execution of this action can interfere with other role termination on the same owning component because the set of adopted roles for the component is changed. Therefore, role retrieval as well as role termination have to acquire the single lock of the owning component to guarantee exclusive execution.

A send action is represented in jHELENA by the class `SendMessageAction` which maintains attributes for the message to be sent and the target role. Its method `execute` first checks the well-formedness criteria that sending role type supports the given message as outgoing and the receiving role type supports the given message as

incoming. If a criterion is not satisfied, an appropriate exception is thrown. Afterwards, the method `sendMessage` of the class `Role` is invoked.

The actual transmission of the message according to the rule *send* in Fig. 3.4 is implemented by the method `sendMessage` of the class `Role`. It first checks the side-condition of the semantic operational rule that the sending role instance and the receiving role instance are in the same ensemble (item (1) and (2). Afterwards, it calls the method `write` of the input channel `input` of the target role instance. This method checks the another side-condition that the capacity of the input channel of the target role instance is not yet exceeded (item (2)). If a side-condition is not satisfied, an appropriate exception is thrown. Afterwards, the message is written to the input channel. It depends on the implementing subclass whether this method returns immediately (for the class `AsyncInputChannel`) or waits until the message is received (for the class `SyncInputChannel`).

The execution of this action can interfere with other message transmissions on the input channel of the same target role instance. Therefore, the method `write` of the class `RoleInputChannel` is synchronized such that only one thread can exclusively execute it.

A receive action is represented in jHELENA by the class `ReceiveMessageAction` which only maintains an attribute for the message to be received. Its method `execute` first checks the well-formedness criterion that the receiving role type supports the given message as incoming. If the criterion is not satisfied, an appropriate exception is thrown. Afterwards, the method `receiveMessage` of the class `Role` is invoked which returns the received message if it could be received (otherwise, an exception is thrown). The received parameters are stored in the message to be returned.

The actual reception of the message according to the rule *receive* in Fig. 3.4 is implemented by the method `receiveMessage` of the class `Role`. The method directly calls the method `read` of the input channel `input` of the receiving role instance. This method checks the side-condition that a message fitting to the expected message is actually first in the input channel (item (1)). If the side-condition is not satisfied, an appropriate exception is thrown. Afterwards, the message is retrieved from the input channel.

The execution of this action can interfere with other message receptions on the input channel of the same role instance. Therefore, the method `read` of the class `RoleInputChannel` is synchronized such that only one thread can exclusively execute it.

A component operation call is represented in jHELENA by the class `OperationCallAction` which maintains attributes for the operation to be called and the variable to store the return value of the operation call. Its method `execute` first checks the well-formedness criterion that the owning component instance of the issuing role supports the given operation. If the criterion is not satisfied, an appropriate exception is thrown. Afterwards, the method `callOperation` of the class `Component` is invoked which returns the return value of the operation call. The return value is stored in the variable.

The actual call of the operation according to the rule *op call 1* and *op call 2* in Fig. 3.5 is implemented by the method `callOperation` of the class `Component`.

The method calls the method which implements the operation on the concrete component class via reflection.

The execution of this action cannot interfere with other actions.

A component attribute setter is represented in jHELENA by the class `CompAttrSetterAction` which maintains attributes for the attribute to be set and the value to be set. Its method `execute` first checks the well-formedness criterion that the owning component type of the issuing role supports the given attribute. If the criterion is not satisfied, an appropriate exception is thrown. Afterwards, the method `setCompAttr` of the class `Component` is invoked.

The actual update of the value of the component attribute according to the rule *comp attr* in Fig. 3.5 is implemented by the method `setCompAttr` of the class `Component`. The method sets the attribute of the concrete component class via reflection.

The execution of this action can interfere with the evaluation of guards of the if-then-else construct. Therefore, setting a component attribute as well as evaluation of a guard containing a component attribute have to acquire the single lock of the component to guarantee exclusive execution.

A role attribute setter is represented in jHELENA by the class `RoleAttrSetterAction` which maintains attributes for the attribute to be set and the value to be set. Its method `execute` first checks the well-formedness criterion that the issuing role type supports the given attribute. If the criterion is not satisfied, an appropriate exception is thrown. Afterwards, the method `setRoleAttr` of the class `Role` is invoked.

The actual update of the value of the role attribute according to the rule *role attr* in Fig. 3.5 is implemented by the method `setRoleAttr` of the class `Role`. The method sets the attribute of the concrete role class via reflection.

The execution of this action cannot interfere with other actions.

A state label is not represented in jHELENA since it is only introduced for model-checking purposes only.

Nondeterministic Choice: According to the rules *nondet. choice 1* and *nondet. choice 1* in Fig. 3.1 on page 54, nondeterministic choice can evolve by either branch which is currently executable. In jHELENA, nondeterministic choice is described by the class `NondeterminsticChoice` which represents both branches by associations with end `p1` and `p2` to the class `ProcessExpression` (not shown in Fig. 7.2). The implementation of the method `step` in the class `NondeterministicChoice` uses these two associations to realize the semantics of nondeterministic choice. Its implementation is shown in the code snippet in Fig. 7.6. The idea of the method is to randomly select one branch and to try to execute it. If the first action of the first branch is currently not executable, we try to execute the second branch. In line 8–18, one branch is randomly selected to be executed first (assignment to the local variable `choice1`) and the other to be executed in case that the first was currently not executable (assignment to the local variable `choice2`). Afterwards, we try to execute the first choice in line 21. If the execution of the first action of the first branch raised an exception of type `ActionCurrentlyNotExecutableException` or `ActionNeverExecutableException` (line 23–24), the second choice is executed in line 25

```
1  ProcessExpression step(Role source)
2       throws WellFormednessViolatedException,
3       ActionCurrentlyNotExecutableException,
4       ActionNeverExecutableException,
5       GuardNeverEvaluableException,
6       InvokedProcessNotSetException {
7
8    ProcessExpression choice1 = null;
9    ProcessExpression choice2 = null;
10
11   if (Math.random() < 0.5) {
12     choice1 = this.p1;
13     choice2 = this.p2;
14   }
15   else {
16     choice1 = this.p2;
17     choice2 = this.p1;
18   }
19
20   try {
21     return choice1.step(source);
22   }
23   catch (ActionCurrentlyNotExecutableException
24           | ActionNeverExecutableException e) {
25     return choice2.step(source);
26   }
27 }
```

Figure 7.6: jHELENA implementation of the method `step` of the class `NondeterministicChoice`

since the first choice was not executable. If any other exception was raised during execution of the first choice, we do not continue with the second choice since some fatal exception occurred like well-formedness of process expressions was violated which has to be reported to the developer of the ensemble-based system. However, if also the execution of the second choice raises an exception of type `ActionCurrentlyNotExecutableException` or `ActionNeverExecutableException`, this exception is handed over to the caller of the method `step`.

If-Then-Else: According to the rules *if-then-else 1* and *if-then-else 2* in Fig. 3.1 on page 54, the if-then-else construct can evolve by the first branch if its guard evaluates in the current state to **true** and the first branch can evolve; the if-then-else construct can evolve by the second branch if its guard evaluates to **false** and the second branch can evolve; otherwise, the whole if-then-else construct cannot evolve. For example, if the guard evaluates to **true**, but the first branch cannot evolve, the if-then-else construct as a whole cannot evolve. In jHELENA, the if-then-else construct is described by the class `IfThenElse` which represents both branches by associations with end `p1` and `p2` to the class `ProcessExpression` (not shown in Fig. 7.2) as well as the guard by an association with end `guard` to the class `Guard` expressing arbitrary guards over an ensemble specification (not shown in Fig. 7.2). The implementation of the method `step` in the class `IfThenElse` uses these three associations to realize the semantics of the if-then-else construct. Its implementation is shown in the code snippet in Fig. 7.7. The idea of the method is to evaluate the guard (line 17) and to try to execute the appropriate branch according to the guard's evaluation (line 17–22). If the selected branch cannot evolve, an exception is raised during the execution of the selected branch (i.e., in line 18 or 21) which is handed over to the caller of the method `step`.

```
1  ProcessExpression step(Role source)
2       throws WellFormednessViolatedException,
3       ActionCurrentlyNotExecutableException,
4       ActionNeverExecutableException,
5       GuardNeverEvaluableException,
6       InvokedProcessNotSetException {
7
8    Set<Component> locks = this.guard.lockObjects(source);
9    try {
10     for (Component lock : locks) {
11       if (lock = null) {
12         throw new ComponentIsNullException();
13       }
14       lock.lock();
15     }
16
17     if (this.guard.isTrue(source)) {
18       return this.p1.step(source);
19     }
20     else {
21       return this.p2.step(source);
22     }
23   }
24   catch (ComponentIsNullException | PropertyNotDeclaredInClassException
25     | ReflectionException | NoBooleanValueException e) {
26     throw new GuardNeverEvaluableException(this.guard, source, e);
27   }
28   finally {
29     for (Component lock : locks) {
30       if (lock != null) {
31         lock.unlock();
32       }
33     }
34   }
35 }
```

Figure 7.7: jHELENA implementation of the method `step` of the class `IfThenElse`

In addition to that, the HELENA semantics prescribes that the evaluation of the guard and the evolution of the appropriate branch must occur as one step. That means that it is not allowed that the value of the guard changes during the execution of the first action of the selected branch. Basically, a guard is built from boolean primitives, data variables, component or role attributes, and plays-queries (and arbitrary compositions of these atomic propositions). Boolean primitives, data variables and role attributes cannot change while the if-then-else construct is evaluated. The reason is that only the role itself can access these variables and attributes and no other role can change them. However, the values of component attributes and plays-queries can change if any other role owned by the same component sets the component attributes from its role behavior or a role which concerns the plays-queries is created or quit. Therefore, in jHELENA, we have to take care that for all component attributes and plays-queries in the guard the corresponding component is locked for changes until the first action of the selected branch was executed. By calling the method `lockObjects` on the `guard` (line 8), all components are retrieved which occur in the guard. All retrieved components are temporarily locked for modifications in line 8–15 by calling the method `lock` of the class `Component` (if the component to be locked is `null`, an exception is raised). Then, the if-then-else construct is evolved and finally, the locks for all components are revoked (line 28–35). This locking mechanism is in-line with the locking of the actions for role creation, role termination, and component attribute setters discussed previously.

Process Invocation: According to the rule *process invocation* in Fig. 3.1 on page 54, process invocation can evolve if the invoked process can evolve. In jHELENA, process invocation is described by the class `ProcessInvocation` which represents the invoked process by an association with end `n` to the class `ProcessExpression` (not shown in Fig. 7.2). The implementation of the method `step` in the class `ProcessInvocation` uses this association to realize the semantics of process invocation. Its implementation is shown in the code snippet in Fig. 7.8. The method just calls the method `step` for the invoked process if the invoked process is not . Otherwise, an appropriate exception is thrown.

```
1  ProcessExpression step(Role source)
2       throws WellFormednessViolatedException,
3       ActionCurrentlyNotExecutableException,
4       ActionNeverExecutableException,
5       GuardNeverEvaluableException,
6       InvokedProcessNotSetException {
7    if (this.n == null) {
8      throw new InvokedProcessNotSetException();
9    }
10   return this.n.step(source);
11 }
```

Figure 7.8: jHELENA implementation of the method `step` of the class `ProcessInvocation`

7.4 System Manager

It remains to mention the abstract `SysManager` class which provides a template method `start` to set-up and start an ensemble system. The method is responsible for the configuration of the component-based platform and ensemble structures, the creation of initial ensemble states, and the launch of concrete ensemble-based applications running ensembles concurrently on top of the component-based platform. Therefore, it sequentially calls the methods `configureTypes` to construct ensemble structures, `createComponents` to create the underlying component instances and `startEnsembles` to start all concurrently running ensembles of the ensemble-based system. All three methods have to be implemented by the developer in a manager subclass when implementing a concrete ensemble-based application.

7.5 Framework Application

We illustrate the use of the framework by implementing our running p2p file transfer ensemble. We perform the implementation in two major steps concerning the structural aspects of the ensemble, i.e., its contributing types and their conceptual relationships described by and ensemble structure, and the dynamic behavior of the ensemble, i.e., the role behaviors of the participating roles. Using the jHELENA framework, the implementation of the p2p example was straightforward and could easily be derived from the formalization in HELENA. Different file transfer ensembles could be instantiated and run concurrently. The complete implementation of the p2p example relying on the jHELENA framework can be found on the attached CD in the project `eu.ascens.helena.p2p`.

7.5.1 Structural Aspects

To create all contributing types and their conceptual relationships of the ensemble, we first extend the classes of the `developer-interface` for each type in the example as shown in the package `p2p` in Fig. 7.2: `Peer` extends `Component`, `Requester`, `Router`, and `Provider` extend `Role`, `TransferEnsemble` extends `Ensemble`, `PrintFileOperation` extends `Operation`, and `ReqAddrMessage`, `SndAddrMessage`, `ReqFileMessage` and `SndFileMessage` extend `Message`. We define component and role attributes as instance variables of the concrete component and role classes, operations as methods of component classes, and parameters of messages as attributes of the particular message classes (not shown in Fig. 7.2). However, we do not realize the role behaviors yet.

Afterwards, we extend the abstract class `SysManager` by the class `PeerSysManager` and implement the method `configureTypes` to configure all types of the p2p example. This method instantiates all type classes of the `metadata` layer and connects them appropriately to represent the ensemble structure $\Sigma_{transfer}$ in Fig. 2.5 on page 23. An excerpt of the implementation is shown in Fig. 7.9. The method first has to create all component types underlying the ensemble-based system. For the p2p example, we instantiate only one component type for peers (instantiation of attribute types and component association types is shown inline as well as for the single operation type) and add it to the set `componentTypes` of the system manager by calling the method `addComponentType` (cf. line 2–8 in Fig. 7.9). Furthermore, we create instances for all message types occurring in the ensemble structure (line 10–24). Afterwards, we create instances for all types of the ensemble structure and connect them accordingly. Line 26–33 in Fig. 7.9 exemplify this for the role type of a requester. Lastly, we compose all types to the desired ensemble structure and add it to the set of ensemble structures `ensembleStructures` for the system (line 35–37).

7.5.2 Dynamic Behavior

The second step is to add dynamic behavior such that the ensemble-based system fulfills its goal-directed behavior. For this purpose, we realize the ensemble specification by implementing the method `initializeRoleBehavior` of all concrete role classes together with a set of attributes representing the variables of the role behavior in the corresponding role class and by implementing the methods representing operations of components. Afterwards, we indicate how to concretely start an ensemble by implementing the method `startEnsemble` of the class `TransferEnsemble`. Lastly, we realize a concrete application by implementing the methods `createComponents` and `startEnsembles` of the class `P2PSysManager`.

Role Behaviors: To realize the role behavior of a role, we implement the method `initializeRoleBehavior` in its concrete role class. Each process expression and action is directly translated to its jHELENA representation. Concrete exchanged messages and called operations are expressed by their jHELENA counterparts. However, the representation of role instance variables and data variables needs special care. For each variable in the role behavior of a certain role including the predefined constant **self**, we define an attribute in the jHELENA class for the role. However, we do not directly type it with the type `T` of the variable used in the role behavior. We rather wrap the variable in the wrapper class `Variable<T>` which just stores the type `T` and the value of the variable. This wrapper class is necessary since Java implements call-by-value. At the moment of initialization of the role behavior, most of the variables occurring in the role

```
1  protected void configureTypes() {
2    ComponentType peerType = ComponentType.createType(Peer.class,
3        getAsSet(DataFieldType.createType("hasFile", Boolean.class),
4           DataFieldType.createType("content", Integer.class)),
5        getAsSet(ComponentAssociationType.createType("neighbor", Peer.class)),
6        getAsSet(OperationType.createType(
7          PrintFileOperation.class, new ArrayList<DataFieldType>(), Void.class)));
8    this.addCompType(peerType);
9
10   MessageType reqAddrMsg = MessageType.createType(
11       ReqAddrMessage.class,
12       getAsList(RoleFieldType.createType("req", Requester.class)),
13       new ArrayList<DataFieldType>());
14   MessageType sndAddrMsg = MessageType.createType(
15       SndAddrMessage.class,
16       getAsList(RoleFieldType.createType("prov", Provider.class)),
17       new ArrayList<DataFieldType>());
18   MessageType reqFileMsg = MessageType.createType(
19       ReqFileMessage.class,
20       getAsList(RoleFieldType.createType("req", Requester.class)),
21       new ArrayList<DataFieldType>());
22   MessageType sndFileMsg = MessageType.createType(
23       SndFileMessage.class, new ArrayList<RoleFieldType>(),
24       getAsList(DataFieldType.createType("content", Integer.class)));
25
26   Set<ComponentType> reqCompTypes = getAsSet(ComponentType.getType(Peer.class));
27   Set<DataFieldType> reqAttrTypes =
28       getAsSet(DataFieldType.createType("hasFile", Boolean.class));
29   Set<MessageType> reqMsgsOut = getAsSet(reqAddrMsg, reqFileMsg);
30   Set<MessageType> reqMsgsIn = getAsSet(sndAddrMsg, sndFileMsg);
31   RoleType req = RoleType.createType(
32       Requester.class, reqCompTypes, reqAttrTypes, reqMsgsOut, reqMsgsIn);
33   ...
34
35   EnsembleStructure transferEnsemble =
36       EnsembleStructure.createType(TransferEnsemble.class, ...);
37   this.addEnsembleStructure(transferEnsemble);
38 }
```

Figure 7.9: Instantiation of types in the method `configureTypes` of the class `P2PSysManager`

behavior have not been set yet. Their values will be received via message receptions or role creations. However, since Java implements call-by-value, the role behavior is initialized with empty values at all places even if the variable is used afters its initialization. However, if we wrap the variables in the wrapper class `Variable<T>`, the role behavior no longer refers to the value of the variable, but to an object which contains the value. Thus, if the variable is initialized during the execution of the role behavior, this value is changed and all later invocations of the variable refer to this new value.

Fig. 7.10 shows the implementation of the class `Router` in our p2p example. The foundation for the implementation is the role behavior given in Fig. 2.7 on page 31. The class `Router` declares attributes for the predefined constant **self** and all role instance variables `router`, `req` and `provider` (line 2–5) which are used in the role behavior.

```
1  public class Router extends Role {
2    protected final Variable<Router> self = new Variable<>(Router.class);
3    protected final Variable<Requester> req = new Variable<>(Requester.class);
4    protected final Variable<Provider> prov = new Variable<>(Provider.class);
5    protected final Variable<Router> rout = new Variable<>(Router.class);
6    ...
7
8    protected ProcessExpression initializeRoleBehavior() throws ... {
9      ProcessInvocation createInvocation = new ProcessInvocation();
10     ProcessInvocation provideInvocation = new ProcessInvocation();
11     ProcessInvocation fwdInvocation = new ProcessInvocation();
12     ProcessInvocation recursion = new ProcessInvocation();
13
14     ProcessExpression routerProc =
15       new ActionPrefix(
16         new ReceiveMessageAction(new ReqAddrMessage(this.req)),
17         new IfThenElse(
18           new CompAttrGetter<>("hasFile", Boolean.class),
19           provideInvocation,
20           fwdInvocation));
21
22     ProcessExpression provide =
23       new ActionPrefix(
24         new CreateRoleAction<>(this.prov, Provider.class, this.getOwner()),
25         new ActionPrefix(
26           new SendMessageAction(this.req, new SndAddrMessage(this.prov)),
27           Quit.getInstance()));
28
29     ProcessExpression fwd =
30       new IfThenElse(
31         new PlaysQuery(
32           new CompAssociationGetter("neighbor").getValue(this),
33           Router.class),
34         Quit.getInstance(),
35         createInvocation);
36
37     ProcessExpression create =
38       new ActionPrefix(
39         new CreateRoleAction<>(
40           this.rout,
41           Router.class,
42           new CompAssociationGetter("neighbor").getValue(this)),
43         new ActionPrefix(
44           new SendMessageAction(this.rout, new ReqAddrMessage(this.req)),
45           recursion));
46
47     provideInvocation.setInvocatedProcess(provide);
48     fwdInvocation.setInvocatedProcess(fwd);
49     createInvocation.setInvocatedProcess(create);
50     recursion.setInvocatedProcess(routerProc);
51
52     return routerProc;
53   }
54 }
```

Figure 7.10: jHELENA implementation of the class `Router`

The method `initializeRoleBehavior` realizes the role behavior relying on the jHELENA representations of process expressions and actions. The process expression `routerProc` is the starting point of the role behavior (line 14). The original role behavior starts by action prefix with the action *reqAddr*. The implementation translates this to a new object of the class `ActionPrefix` (line 15) with two parameters. The first parameter represents the reception of the message *reqAddr* by instantiating a new object of the class `ReceiveMessageAction` with an object for the message to be sent as parameter

(line 16). The second parameter of action prefix is the remaining process expression, in this case an if-then-else construct. The if-then-else construct is expressed by a new object of the class `IfThenElse` (line 17) where the first parameter in line 18 represents the guard of the if-then-else construct (evaluating whether the attribute `hasFile` of the owning component instance is **true**), the second parameter in line 19 the **if**-branch and the third parameter in line 20 the **else**-branch. Both branches invoke another process which is expressed by the local variables of type `ProcessInvocation`. The invoked processes are defined only after this with the process expression `provide` (line 22–27) and `fwd` (line 29–35). Thus, we cannot set the invoked process directly in the definition of the process expression `routerProc`, but we have to set it afterwards in line 47 and 48. The remaining process is translated analogously and depicted in Fig. 7.10.

Operations: Operations of components are implemented as methods of the corresponding (subclass of the) class `Component`. They have to take the parameters of the operation as input. The body of the method implements the behavior of the operation which was not yet part of the ensemble specification, but has now to be added by the developer. In our p2p example, the component type *Peer* has just one operation *printFile* which is implemented by the method `printFileOperation` of the class `Peer`. As shown in Fig. 7.11, the operation just prints the String `PRINT FILE` to the Java output console. However, more sophisticated behavior could be added, e.g., printing to a real printer.

```
1  public void printFileOperation() {
2    System.out.println("PRINT_FILE");
3  }
```

Figure 7.11: jHELENA implementation of the method `printFileOperation` in the class `Peer`

Initial State of an Ensemble: The method `startEnsemble` of the class `TransferEnsemble` actually starts an instance of the ensemble (cf. Fig. 7.12). The method gets an initial component as input where the file was initially requested. It creates a role instance of type `Requester` adopted by the initial (peer) component, thus starting to execute the requester's behavior.

```
1  public void startEnsemble(Component initialComponent) throws ... {
2    this.createRole(Requester.class, initialComponent);
3  }
```

Figure 7.12: jHELENA implementation of the method `startEnsemble` in the class `TransferEnsemble`

Lastly, a concrete scenario needs to be set up. The system is populated by concrete peers in the method `createComponents` of the `P2PSysManager` (cf. Fig. 7.13). Five peers are initialized as indicated in line 2–6. All peers do not have the requested file except the fourth peer. The network of peers as a ring structure is set up (line 8–12), and each peer is added to the set `currentComponents` of the class `P2PSysManager` (line 14–18). Afterwards, concrete ensemble instances are created and run in the method `startEnsembles` (cf. Fig. 7.14).

```
protected void createComponents() {
  Peer peer1 = new Peer("p1", false, 0);
  Peer peer2 = new Peer("p2", false, 0);
  Peer peer3 = new Peer("p3", false, 0);
  Peer peer4 = new Peer("p4", true, 12345);
  Peer peer5 = new Peer("p5", false, 0);

  peer1.setNeighbor(peer2);
  peer2.setNeighbor(peer3);
  peer3.setNeighbor(peer4);
  peer4.setNeighbor(peer5);
  peer5.setNeighbor(peer1);

  this.addComponent(peer1);
  this.addComponent(peer2);
  this.addComponent(peer3);
  this.addComponent(peer4);
  this.addComponent(peer5);
}
```

Figure 7.13: Instantiation of peers in the method `createComponents` of the class `P2PSysManager`

```
protected void startEnsembles() throws ... {
  Ensemble ens1 = new TransferEnsemble("ens1", this.getComponents());
  this.addEnsemble(ens1);
  ens1.startEnsemble(this.getComponent());

  Ensemble ens2 = ...
}
```

Figure 7.14: jHELENA implementation of the `startEnsembles` in the class `P2PSysManager`

7.6 Related Work

When it comes to implementation, the HELENA approach shares its foundation with frameworks from different areas: Ensemble-based systems which particularly deal with groups of autonomic entities, role-based modeling which introduces the notion of roles for only focus on a certain perspective of an object, and implementations of communication groups which consider the architecture and protocol throughout the collaboration of a group.

7.6.1 Implementations of Ensemble-Based Systems

The EU project ASCENS [WHKM15] develops foundations, techniques and tools to support the whole life cycle for the construction of Autonomic Service Component ENSembles. In this context, several approaches to formalize and implement ensemble-based systems have been developed. SCEL and its implementation jRESP [DLPT14] provide a kernel language for abstract programming of autonomic systems, whose components rely on knowledge repositories, and models interaction by knowledge exchange. In SCEL and jRESP, ensembles are understood as communication groups which are defined by predicates determining the participants of the group. The participants communicate by

putting knowledge items to knowledge repositories of communication groups. In contrast, HELENA relies on directed message exchange between participants of ensembles and introduces a second role layer on top of a component-based platform to allow a more flexible mechanism for dynamic ensemble composition.

DEECo and its implementation jDEECo [BGH+13] introduces an explicit specification artifact for ensembles dynamically formed according to a given membership predicate. Interaction is realized by implicit knowledge exchange managed by DEECo's runtime infrastructure. However, HELENA is more concrete since we include an explicit notion of interaction and collaboration.

Compared to both implementations, jHELENA introduces a clear separation between type level and instance level. The `metadata` level implements the formal syntax of HELENA and thus the type level. By using the abstractions introduced in this layer, the developer is forced to respect the syntactic restrictions of HELENA when introducing component types, role types and ensemble structures. The `developer interface` implements the formal semantics of HELENA and thus the instance level. By relying on the abstractions provided by this layer, the developer defines concrete instances of component types, role types and ensembles. He furthermore specifies role behaviors using the jHELENA abstractions of the formal HELENA process constructs. jHELENA takes care to execute the roles and their behavior according to the semantic rules of HELENA.

7.6.2 Implementations of Roles

With HELENA, we offer a rigorous approach for developing goal-oriented groups on the basis of roles. Modeling evolving objects with roles as perspectives on objects has been proposed by various authors [GSR96, KØ96, Ste00b, Ste00a], but they do not see them as autonomic entities with behavior as we do in HELENA.

Gottlob et al. [GSR96] propose role hierarchies to complement object-oriented systems for evolving objects. Their implementation in Smalltalk follows the same concept as the HELENA framework by binding role instances to objects. However, they do not consider any collaboration between roles to perform cooperative tasks.

Kristensen et al. [KØ96] define roles as perspectives of some objects sharing the basic ideas with Gottlob et al. Like the HELENA framework, their implementation in BETA and Smalltalk emphasizes that objects can only be accessed through their role references (or in the case of sets of roles, subject references). In their approach, roles can be transferred between objects without interrupting role-specific behavior. This idea could be interesting to integrate into HELENA to complement the idea of ensembles.

Steimann [Ste00b, Ste00a] proposes a formal model for roles and relationships between roles. His "model specifications" are comprised of signature, static model, and dynamic model similarly to HELENA, but they do not specify any collaborations or object interactions. Based on this formal model for roles, Steimann defines the rudimentary modeling language LODWICK. Compared to HELENA, this modeling language is very high-level and not supported by an execution framework like jHELENA. However, he proposes to indicate by interface realization which (component) types can adopt which roles. Hence, roles correspond to interfaces and do not provide behavior implementations.

Steegmans et al. [SWHB05] propose a role model where agents commit themselves to roles and therefore execute the associated behavior given by action diagrams. However, they do not transfer the idea of roles to the implementation level as we do it with jHELENA, but rather rely on free-flow architectures for realization.

7.6.3 Implementations of Collaboration Groups

The idea to describe structures of interacting objects without having to take the entire system into consideration was already introduced by several authors [Her03, BSI07, Ree96, TUI07], but they do not consider roles as autonomic entities and do not tackle concurrently running ensembles as we do in HELENA.

Herrmann [Her03] introduces "teams" in his framework ObjectTeams/Java, Baldoni et al. [BSI07] "institutions" in their framework powerJava, and Reenskaug [Ree96] and Andersen [And97] "role models" in their OOram method. Like in HELENA, they define the structural model of a collaboration by participating roles, but they handle behavior very differently. In ObjectTeams/Java and powerJava, collaboration between roles is initiated through operation calls while in the OOram method, roles exchange message like in HELENA. In ObjectTeams/Java and powerJava, roles are not active themselves, but can only react to operation calls. The OOram method pursues our idea of roles as being autonomic entities which start their behavior based on an external stimulus (like a file being requested from the outside). However, while in HELENA we model concurrently running ensembles, in the OOram method overlapping role models are composed into a single composite role model. Therefore state spaces only represent composite behaviors while we explicitly run behaviors in parallel.

The modeling approach Macodo [HWH14] introduces a set of role-based abstractions to define collaborations. It is supported by a proof-of-concept middleware which provides appropriate programming concepts to map the role-based abstractions to Web service technologies. However, their focus is only on the collaboration-level and does not include the concrete realization of individual role behaviors.

Related approaches have also been developed in the context of multi-agent systems and multi-party session types with the Scribble framework [YHNN13]. It provides a high-level language to describe collaborations or, in terms of the authors, session types which consist of a prescribed scenario of interactions. In contrast to jHELENA, Scribble does not allow the dynamic creation of new participants and the concurrent execution of ensembles which is built-in in the HELENA semantics and its implementation.

7.7 Publication History

This chapter extends and improves the jHELENA framework already presented in [KH14]. Compared to this publication, the jHELENA framework in its current implementation considers all syntactic constructs of Chap. 2 and realizes the formal SOS semantics in Chap. 3 which was not available when the first version of jHELENA has been presented in [KH14]. Especially, the implementation of role behaviors has been improved with special care to match the SOS rules in Sec. 3.3. Furthermore, this chapter describes the jHELENA framework in full detail while in [KH14] we focused only on the most important ideas.

7.8 Present Achievements and Future Perspectives

Present Achievements: For the implementation of ensemble-based systems, we provide the Java framework jHELENA. The construction of the framework was rigorously guided by the abstract notions of ensemble specifications and their semantics used for modeling ensembles in the HELENA approach. HELENA extends component-based systems with the notion of roles and ensembles to focus on capabilities of a component

needed for particular collaborations. Our framework transfers this concept to an object-oriented platform and directly implements the formal foundations and the execution model of the HELENA approach. The classes provided by the jHELENA framework can be extended for particular ensemble-based applications. The developer is forced by the framework to respect the formal restrictions of the HELENA approach. If an ensemble-based application is executed with the jHELENA framework, it is guaranteed that the execution of the system follows the semantics of HELENA.

As the reader might have noticed, the derivation of the implementation follows a systematic translation from the formal HELENA ensemble specification to jHELENA code. In the following chapter, we will exploit this systematic translation to provide an automated code generator. It takes a HELENA ensemble specification described with the domain-specific language HELENATEXT as input and automatically generates a Java implementation based on the jHELENA framework.

Future Perspectives: The jHELENA framework is a first prototype which can be extended in several directions:

Communication Styles: Further communication styles like broadcast messaging or knowledge exchange as envisioned in SCEL and DEECo could be supported. This would extend the set of actions by broadcasting and knowledge repository access. At the same time, the way of message transmission which is currently realized via message queues must be reconsidered and special data structures for knowledge repositories have to be included.

Distributed Deployment: To allow real distribution, the framework could be based on a component infrastructure which supports distributed deployment of components. So far, all components are initialized in the same Java virtual machine; only roles are run in different threads, but on the same machine. A distribution framework would allow to set up a distributed network of components which collaborates in ensembles across different machines. However, new issues have to be addressed like limited communication range or message loss.

Proof of Preservation of the HELENA *Semantics:* Throughout this chapter, we argued that the jHELENA framework preserves the HELENA semantics by construction. To formally guarantee that, the semantic equivalence of a HELENA ensemble specification with its Java implementation following the jHELENA framework should be shown similarly to the semantic equivalence with the PROMELA translation. However, this would be a challenging proof which had to rely on a formal Java semantics including threads like in [CKRW99]. Anyway, it could be an interesting option to use Java PathFinder [Lau16] for verification instead of Spin as proposed in Chap. 5. Java PathFinder allows to analyze executable Java programs for properties like deadlocks, unhandled exceptions, and data races, but also for LTL properties if given in the form of a Büchi automaton.

Chapter 8

HELENA Workbench
Working with HELENA

When developing ensemble-based system according to HELENA from specification through verification to implementation, the developer may experience some pitfalls. Without any editor support, the developer has to ensure himself that his ensemble specifications conform to HELENA and respect all constraints formulated in the formal definitions. To verify the satisfaction of goals for an ensemble, the specification has to be translated to PROMELA by hand to be able to model-check the resulting verification model with Spin. Although concrete formal rules accurately determine the translation, it is inherently error-prone due to its manual execution. To implement an ensemble, the specification must also be translated to Java code relying on the jHELENA framework by hand. Thus, it cannot be guaranteed that the manual implementation indeed respects the formal specification, in particular, that role behaviors are implemented correctly.

We therefore provide the HELENA workbench supporting the whole development process of ensemble-based systems with HELENA. The domain-specific language HELENATEXT serves as concrete syntax for HELENA ensemble specifications supporting roles and ensemble structures as first-class citizens. Relying on the XTEXT workbench, Eclipse integration of the domain-specific language is offered which features a full HELENATEXT editor including syntax highlighting, content assist, and validation. Moreover, we define a set of rules for the automatic generation of the PROMELA verification model and the Java implementation from an ensemble specification. The rules are directly derived from their formal counterparts and therefore allow a reliable translation. Both code generators are integrated into Eclipse and the fully-fledge editor.

Table 8.1 gives an overview about the development of and with the HELENA workbench. In Sec. 8.2, we introduce HELENATEXT, the domain-specific language realizing the formal syntax rules of HELENATEXT, and the HELENATEXT editor integrated into Eclipse. The two generators to PROMELA and Java are presented in Sec. 8.3 and Sec. 8.4. We finally conclude with future work in Sec. 8.6.

The complete implementation of the HELENA workbench can be found on the attached CD in the projects `eu.ascens.helenaText`, `eu.ascens.helenaText.sdk`, `eu.ascens.helenaText.tests`, and `eu.ascens.helenaText.ui`. The p2p example is exercised with the HELENA workbench in the project `eu.ascens.helenaText.p2p`.

8.1 Overview of the HELENA Workbench

The HELENA workbench is implemented as a plug-in for the Eclipse development environment[1]. Fig. 8.1 shows a screenshot of the final HELENA workbench during the implementation of the p2p example. The XTEXT workbench of Eclipse provides the means to define the domain-specific language (DSL) for the HELENA workbench, to generate and customize a fully-fledged editor for the DSL, and to define code generators from the DSL to any other language. We give a short overview about the XTEXT workbench in the next subsection. Afterwards, we focus on the workflow how the HELENA workbench itself has been implemented and on the workflow how to implement ensemble specifications with the HELENA workbench.

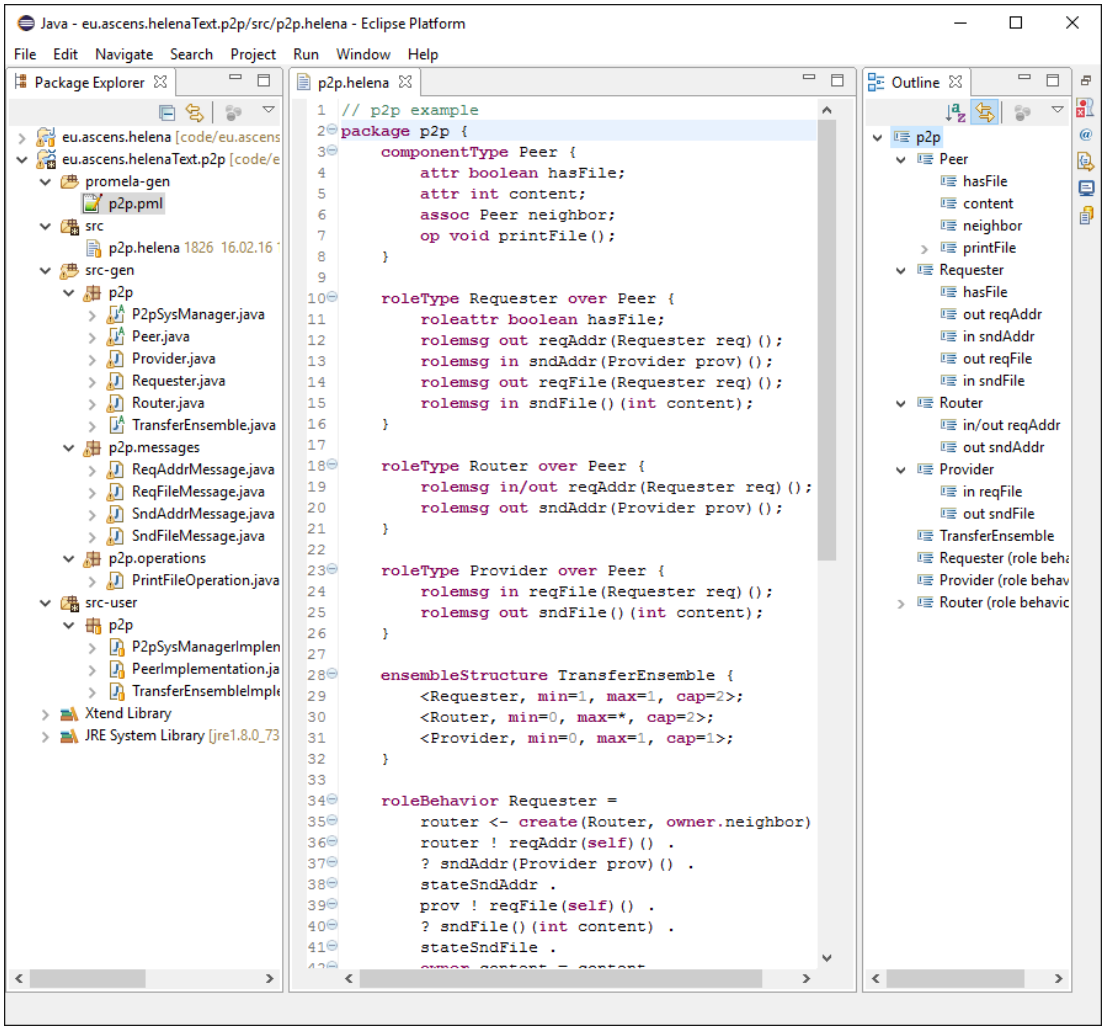


Figure 8.1: Screenshot of the HELENA workbench during development of the p2p example

8.1.1 Eclipse's XTEXT Workbench

XTEXT[2] is a framework for the development of DSLs fully integrated into Eclipse. The user of the XTEXT workbench can define a custom grammar for his DSL in a BNF-like

[1] http://www.eclipse.org/
[2] https://eclipse.org/Xtext/

notation. XTEXT generates a complete language infrastructure for this DSL including parser, linker, compiler, interpreter, and a fully-fledged Eclipse editor. All provided tools can be extended to fit the user's needs. In particular, special validation rules can be defined to detect errors in specifications written in the new DSL. The appearance and provided features of the editor can be adapted, e.g., formatting of code, content assist, or code outlines can be customized. Furthermore, code generators can be defined which translate specifications written in the DSL to any other language. To allow adaptation and extension of the language infrastructure, the XTEXT workbench internally creates an EMF (Eclipse Modeling Framework) metamodel which builds the foundation for the editor and all code generators. This metamodel allows the integration with other EMF framework of Eclipse, e.g., with the Graphical Modeling Framework (GMF) to provide a graphical syntax and editor for the DSL.

An integral extension to the XTEXT workbench is the programming language XTEND[3]. Formerly part of the XTEXT project, it now resides as a separate project in the Eclipse context. XTEND is a dialect of Java which also directly compiles to Java. However, compared to Java, it offers some important features which make it a perfect companion in the context of language development:

- **Type interference and lambda expressions** allow efficient programming. Types can automatically be derived and thus method signatures can be left unspecified or variable declarations untyped. Lambda expressions allow amongst others to map transformation functions to a whole collection of objects.
- **Extension methods** enhance types generated from the DSL grammar by new methods without modifying the types themselves.
- **Template expressions** allow a code generator to specify abstract translation rules which are easy to read. They define placeholders which the code generator replaces at runtime by concrete values to gain the final translation. Furthermore, template expression provide control structures like conditional branching or loops to control the composition of the template expression.

The XTEXT workbench finally relies on Java to provide the Eclipse editor for the new DSL with all its features like syntax highlighting, content assist, validation, formatting, and code generation. The editor is implemented as an Eclipse plug-in which can be imported into any Eclipse installation.

8.1.2 Workflow of the Implementation of the HELENA Workbench

The XTEXT workbench guides the development process of the HELENA workbench as an Eclipse plug-in. Fig. 8.2 gives an overview about the steps which are necessary to create the HELENA workbench. Boxes with rounded corners denote activities in the workflow and boxes with sharp corners input and output artifacts of these activities. The gray artifacts provide the HELENA workbench as an Eclipse plug-in consisting of the domain-specific language HELENATEXT and an appropriate Eclipse editor with syntax highlighting, validation and code generators.

The workflow starts with the creation of an empty XTEXT project in Eclipse. The project mainly contains a stub for the grammar of the DSL to be defined as an `.xtext`-file. The stub for the grammar is extended with the definition of the domain-specific language HELENATEXT, carefully capturing all conditions from the formal definitions of the HELENA syntax (we will discuss the grammar in Sec. 8.2). From the fully-specified

[3] `http://www.eclipse.org/xtend/`

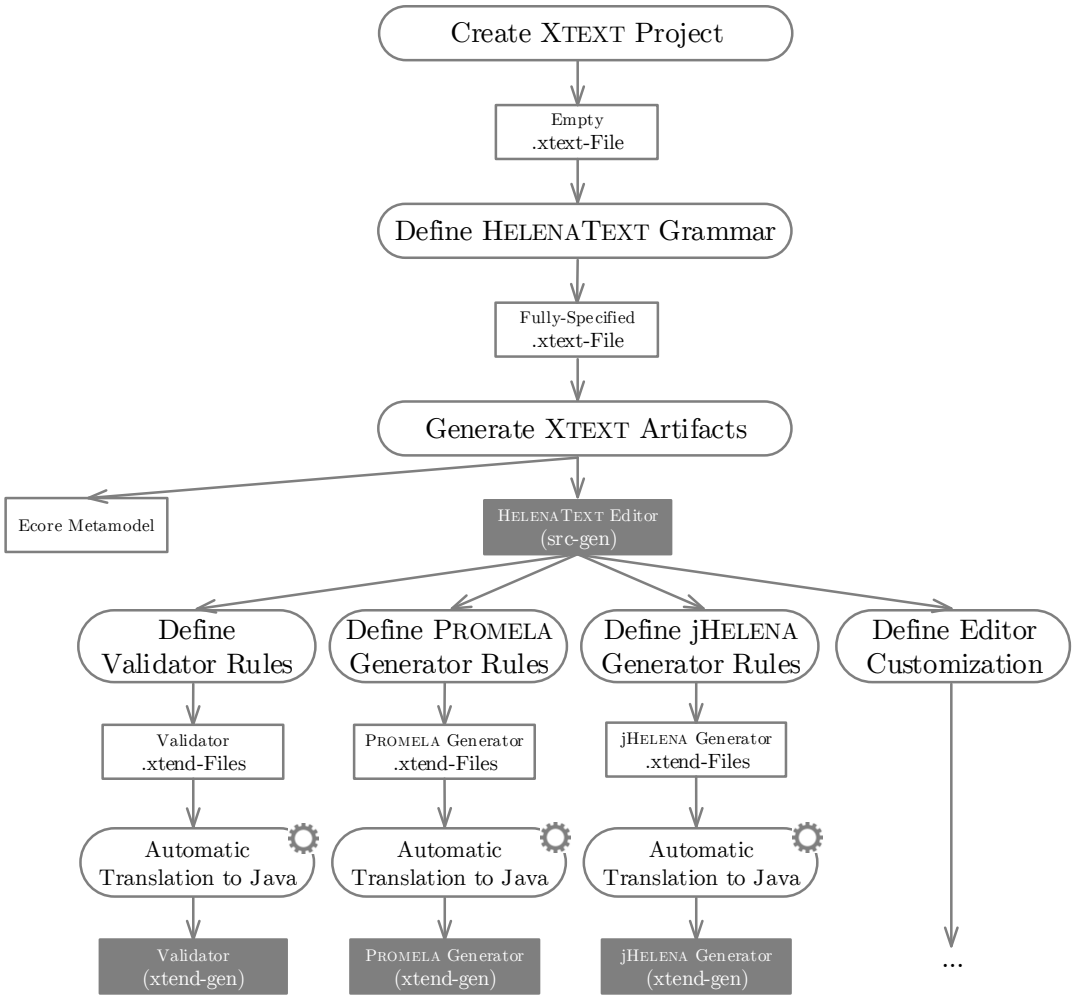


Figure 8.2: Workflow of the implementation of the HELENA workbench

grammar, the generation of all artifacts required for the textual HELENATEXT editor is triggered. The editor is offered by a set of generated Java classes which all reside in the folder `src-gen`. On the one hand, they represent all constructs specified in the HELENATEXT grammar in Java. On the other hand, they provide a basic Eclipse editor which supports the DSL HELENATEXT as input language. To this end, the generated Java code contains a parser, lexer and linker for HELENATEXT and the implementation of the HELENATEXT editor with basic syntax highlighting, content assist, outline proposals etc. In addition to the Java classes representing the HELENATEXT constructs and the editor, the generation of artifacts from the HELENATEXT grammar creates an ecore metamodel of the newly specified DSL. This metamodel is not further used in the current implementation of the HELENA workbench. However, if a graphical editor should be provided in addition to the textual HELENATEXT editor, this metamodel serves as foundation for the editor's development with the Graphical Modeling Framework of Eclipse.

The next steps in the workflow adapt and extend the basic (textual) HELENATEXT editor to support the whole HELENA development process. Firstly, some restrictions of the HELENA syntax cannot be expressed in the HELENATEXT grammar, e.g., a message can only be sent to a role which supports it as incoming message. Such restrictions are defined as validation rules using XTEND (we will discuss an excerpt of the validation

rules in Sec. 8.2). During the usage of the final HELENATEXT editor, the rules will be evaluated on-the-fly on concrete HELENATEXT specifications. To integrate the validation rules in the HELENATEXT editor, the XTEXT workbench automatically (and without any interaction with the user) translates the XTEND rules to Java classes in the folder `xtend-gen`. Furthermore, two generators are defined. The first generator translates a HELENATEXT specification to PROMELA and the second to Java code relying on the jHELENA framework. The translation rules for both generators are defined with XTEND. During the usage of the final HELENATEXT editor, the rules will automatically translate valid HELENATEXT specifications to PROMELA and Java resp. To integrate the generators the HELENATEXT editor, the XTEXT workbench again automatically translates the XTEND rules to Java classes in the folder `xtend-gen`. Finally, the appearance and features of the HELENATEXT editor can be further customized, e.g., to include user-defined formatting. This customization is also specified with XTEND and integrated in the HELENATEXT editor. A user-guide how to implement and provide the HELENA workbench to ensemble developers can be found in Appendix B.1.

To make the dependencies between the created and generated artifacts of the HELENA workbench clear, Fig. 8.3 only considers the relationships between all artifacts. The main input artifact is the `.xtext`-file. The developer of the HELENA workbench defines the grammar of the DSL HELENATEXT in this file. From the `.xtext`-file, two artifacts are generated: The ecore metamodel represents all constructs of the HELENATEXT grammar as a metamodel according to the Eclipse Modeling Framework. It can be used as a foundation for the development of a graphical HELENA editor with the Graphical Modeling Framework of Eclipse. The (textual) HELENATEXT editor is defined by a set of Java classes in the folder `src-gen`. The classes represent all constructs of the HELENATEXT grammar in Java and provide the basic HELENATEXT editor. To adapt and extend the basic HELENATEXT editor, a validator, the PROMELA generator and the jHELENA generator are defined by XTEND rules. These rules rely on the representation of all constructs of the HELENATEXT grammar in Java, i.e., on the set of Java classes in the folder `src-gen`. From the XTEND files, their representations in Java are automatically generated in the folder `xtend-gen` and integrated into the basic HELENATEXT editor. Thus, the HELENA workbench consists of the two Java artifacts shown in gray in Fig. 8.3: The basic HELENATEXT editor in the folder `src-gen` and the user-defined customizations and extensions in the folder `xtend-gen`.

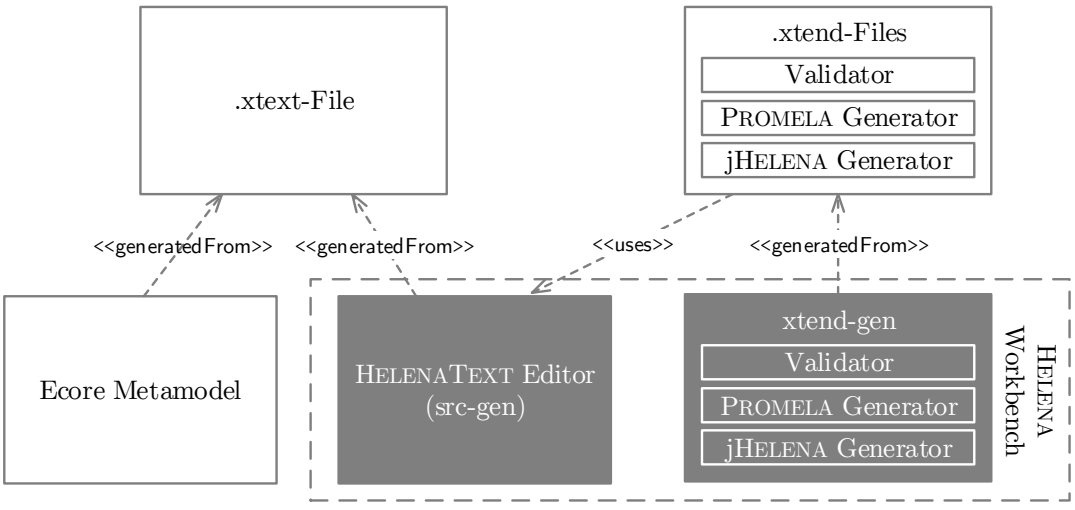


Figure 8.3: Dependencies between artifacts of the HELENA workbench

8.1.3 Workflow of the Usage of the HELENA Workbench

The HELENA workbench is installed in an Eclipse instance as an Eclipse plug-in. Appendix B.2 explains the installation process. Once the HELENA workbench is set up, the user can develop ensembles from specification through verification to implementation fully-integrated into in Eclipse. Fig. 8.4 gives an overview about the workflow of developing ensembles with the HELENA workbench. The user creates a HELENA-project and in particular a `.helena`-file. This file is automatically opened in the HELENATEXT editor provided by the HELENA workbench. In the editor, an ensemble specification can be written in HELENATEXT with full tool support like syntax highlighting, code completion, validation, and code generation. The particular features of the HELENA workbench are triggered by typing any letter in the HELENATEXT editor or by saving the `.helena`-file: Whenever a letter is added to the `.helena`-file, the HELENATEXT editor takes care to present the ensemble specification in the `.helena`-file with syntax highlighting, content assist, code outline to the user. At the same time, the validator evaluates its validation rules and gives feedback about incorrect parts annotated as warnings and errors to the `.helena`-file. On any save action of the HELENATEXT specification, the two automatic code generators are started: The PROMELA generator translates the `.helena`-file to a `.pml`-file. The translation follows the rules described in Sec. 5.2. The resulting `.pml`-file has to be enhance by an initial states and goals to be checked to be used for verification with Spin. The jHELENA generator translates the `.helena`-file to Java code as described in Chap. 7. The generated code is split into two parts: All files in the package `src-gen` implement the ensemble specification given in HELENATEXT. However, HELENATEXT does not yet allow to specify the effect of component operations and an initial state for the ensemble specification. Code stubs for these two open points are generated to the folder `src-user`. They are only created once to not overwrite any user-defined code and need to be implemented by the user.

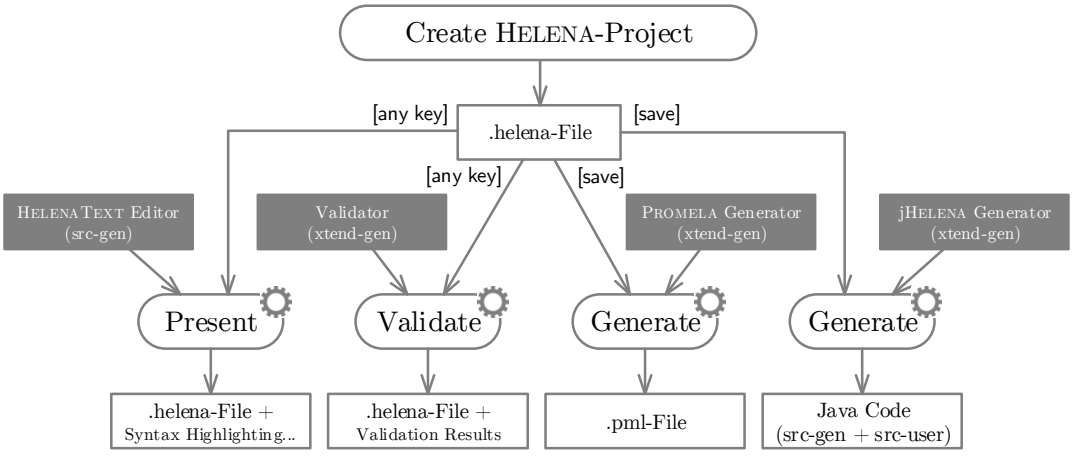


Figure 8.4: Workflow of the usage of the HELENA workbench

8.2 The Domain-Specific Language HELENATEXT

To allow the user of the HELENA workbench to specify ensemble specifications with the HELENA concepts as first-class citizens, we introduce the domain-specific language HELENATEXT. The grammar of the DSL HELENATEXT is defined in the BNF-like notation of XTEXT. It follows the formal definitions of the HELENA modeling elements

like component and role types, ensemble structures and role behaviors. Constraints which cannot be included into the DSL grammar are formulated as validation rules written in XTEND. The grammar rules for all syntactic constructs of HELENATEXT can be found in Appendix B.3. In this section, we focus on the representation of structural aspects, in particular role types, as well as on the representation of dynamic behavior, in particular role behaviors. We show grammar rules written in XTEXT as well as examples of validation rules written in XTEND.

8.2.1 Structural Aspects

To exemplify the derivation of the grammar rules for types, let us revisit the definition of a role type from Chap. 2: A role type rt over a given set of component types CT is a tuple $rt = (rtnm, rtcomptypes, rtattrs, rtmsgs_{out}, rtmsgs_{in})$. Fig. 8.5 shows the corresponding grammar rule. A role type declaration in HELENATEXT must start with the keyword `roleType` followed by its `name` referring to *rtnm*. The set *rtcomptypes* of component types which can adopt the role are reflected by the list `compTypes` after the keyword `over`. It is a list of references to already defined component types which is expressed by the square brackets, the cross reference concept of XTEXT. In curly braces, the two sets `roleattrs` referring to *rtattrs* and `rolemsgs` referring to *rtmsgs* are defined in arbitrary order.

```
1  RoleType:
2  'roleType' name=ValidID 'over' compTypes+=[ComponentType]
3                                 (','compTypes+=[ComponentType])* '{'
4     (
5           roleattrs += ('roleattr' type=JvmTypeReference  name=ValidID ';')
6        |  rolemsgs  += ('rolemsg'  direction=MsgDirection name=ValidID
7                                    formalRoleParamsBlock=FormalRoleParamsBlock
8                                    formalDataParamsBlock=FormalDataParamsBlock ';')
9     )*
10 '}'
11 ;
```

Figure 8.5: XTEXT grammar rule for role types in HELENATEXT

However, the DSL grammar rule cannot express that the lists `compTypes`, `roleattrs`, and `rolemsgs` (depending on the direction of the message) all have to be duplicate-free to represent the sets *rtcomptypes*, *rtattrs*, $rtmsgs_{out}$ and $rtmsgs_{in}$. For that, a validation rule in XTEND is added (cf. Fig. 8.6). Each set of elements is handled separately, for messages we even split the set according to whether they are incoming or outgoing messages (cf. line 5-6). For each set, we call the method `findDuplicates` which reports an error in line 13 if an element with the same name exists in the investigated set.

Fig. 8.7 illustrates the application of the grammar rule for role types in the HELENA workbench on the p2p example. We rely on the declaration of the role type of a router which was already presented in Fig. 2.2 on page 21 in abstract notation and in Fig. 2.3b on page 22 graphically. Fig. 8.7 defines the same role type in HELENATEXT.

```
1  @Check
2  def check_rt_hasDuplicates(RoleType rt) {
3    findDuplicates(rt.compTypes);
4    findDuplicates(rt.roleattrs);
5    findDuplicates(rt.rolemsgs.
6         filter[direction == MsgDirection.OUT || direction == MsgDirection.INOUT]);
7    findDuplicates(rt.rolemsgs.
8         filter[direction == MsgDirection.IN  || direction == MsgDirection.INOUT]);
9  }
10
11 private def void findDuplicates(Iterable<T extends AbstractDuplicateFreeObject> list) {
12   var Set<String> nameSet = new TreeSet();
13   for (AbstractDuplicateFreeObject elem : list.filterNull) {
14     if (!nameSet.add(elem.name)) {
15       error('Duplicate declaration of ' + elem.name, ...)
16     }
17   }
18 }
```

Figure 8.6: XTEND validation rule for role types in HELENATEXT

```
1  roleType Router over Peer {
2    rolemsg in/out reqAddr(Requester req)();
3    rolemsg out sndAddr(Provider prov)();
4  }
```

Figure 8.7: Role type of a router in the p2p example in HELENATEXT

Dynamic Behavior Besides capturing the structural aspects of an ensemble specification with component types, role types and ensemble structures, the dynamic behavior is defined as role behaviors. Role behaviors are composed from process expressions with the process constructs **quit** for role termination, action prefix, nondeterministic choice, if-then-else, and process invocation. The grammar rule for defining such role behaviors (cf. Fig. 8.8) directly follows the inductive definitions in Def. 2.9 on page 25 and Def. 2.12 on page 30. A role behavior can either directly declare its defining process expression (line 2–3) or it can invoke another process (line 4–5) from the set `processes` which it exclusively defines for itself (line 6). The definition of a process (line 9) thereby only differs from the definition of a declaring role behavior insofar that it is declared with the keyword `process` instead of the keyword `roleBehavior`. Furthermore, a process can only be defined in the scope of a certain role behavior which is not further shown here. The different process constructs which can be used as process expression in HELENA according to Def. 2.9 on page 25 are captured by appropriate counterparts in the HELENATEXT grammar (line 11–18). Their composition is a direct representation of the abstract syntax defined in Chap. 2.

The actions for role instance creation and retrieval, sending and receiving messages, operations calls, setting values of attributes, and state labels which can be executed in a role behavior are similarly expressed in the HELENATEXT grammar. The rules can be found in Appendix B.3. They follow the inductive definition in Def. 2.9 on page 25 and directly transfer the abstract syntax to a concrete intuitive notation. Instead of listing all rules here, we will rather illustrate the application of the HELENATEXT grammar rules for role behaviors and actions at the p2p example later on.

In Sec. 2.4.2, we stated all conditions which a role behavior has to satisfy to be well-formed. Those well-formedness criteria cannot be expressed in the DSL grammar.

```
1  RoleBehavior:
2       {DeclaringRoleBehavior} 'roleBehavior'
3           roleTypeRef=[RoleType] '=' processExpr=ProcessExpression
4     |  {InvokingRoleBehavior} 'roleBehavior'
5           roleTypeRef=[RoleType] '=' processInvocation=ProcessInvocation
6           '{' (processes+=Process)* '}';
7  Process: 'process' name=ValidID '=' processExpr=ProcessExpression;
8
9  ProcessExpression:
10      {QuitTerm} 'quit'
11    |  {ActionPrefix} (action=Action '.' processExpr=ProcessExpression)
12    |  {NondeterministicChoice}
13         '(' first=ProcessExpression '+' second=ProcessExpression ')'
14    |  {IfThenElse} 'if' '(' guard=Guard ')' '{' ifProcessExpr=ProcessExpression '}'
15                    'else' '{' elseProcessExpr=ProcessExpression '}'
16    |  {ProcessInvocation} process=[Process];
```

Figure 8.8: XTEXT grammar rule for role behaviors, processes and process expressions in HELENATEXT

Therefore, we add validation rules written in XTEND. We explain two validation rules. The rule in Fig. 8.9 expresses that in any nondeterministic choice construct, the first actions of the two branches are either incoming messages or any other action than an incoming message. Line 3 recursively retrieves all first actions of possibly nested nondeterministic choice or if-then-else constructs of the first branch. The condition of the if-statement in line 4–5 then checks whether the first actions are either all incoming messages (line 4) or any other action (line 5). If so, an error is shown in the HELENATEXT editor. Similarly, the first actions of the second branch are checked in line 9–13.

```
1  @Check
2  def check_rb_noMixedStates(NondeterministicChoice term) {
3    var actions = term.first.firstActions;
4    if (! (actions.forall[it instanceof IncomingMessageCall] ||
5           actions.forall[! (it instanceof IncomingMessageCall)]) ) {
6      error('In nondeterministic choice, mixed states are not allowed.', ...);
7    }
8    var actions2 = term.second.firstActions;
9    if (! (actions2.forall[it instanceof IncomingMessageCall] ||
10          actions2.forall[! (it instanceof IncomingMessageCall)]) ) {
11     error('In nondeterministic choice, mixed states are not allowed.', ...);
12   }
13 }
```

Figure 8.9: XTEND validation rule for nondeterministic choice in HELENATEXT

We illustrate the use of the DSL HELENATEXT for role behaviors at the role behavior of a router in the p2p example. The role behavior is given in abstract notation in Fig. 2.7 on page 31. Fig. 8.10 defines the same role behavior in HELENATEXT. Apparently, most formal constructs occurring in role behaviors are directly represented in an intuitive concrete syntax.

```
1  roleBehavior Router = RouterProc {
2    process RouterProc =
3      ? reqAddr(Requester req)() .
4      if ( owner.hasFile ) { Provide }
5      else { Fwd }
6
7    process Provide =
8      prov <- create(Provider, owner) .
9      req ! sndAddr(prov)() . quit
10
11   process Fwd =
12     if ( plays(Router, owner.neighbor) ) { quit }
13     else { Create }
14
15   process Create =
16     router <- create(Router, owner.neighbor) .
17     router ! reqAddr(req)() .
18     RouterProc
19 }
```

Figure 8.10: Role behavior of a router in the p2p example specified in HelenaText

8.3 Automated Promela Code Generator

To assure that an ensemble specification actually achieves its goals, we proposed in Chap. 5 to translate it to a Promela verification model and to check the translation with the model-checker Spin [Hol03] against its goals specified as LTL formulae. Table 5.2 already proposed a formal translation function from Helena to Promela and Chap. 6 proved (a simplified version of) the translation semantically correct. However, though formally defined, a manual translation according to this formal translation function is error-prone. Thus, we support the user of the Helena workbench for his verification job with an automated code generator translating a HelenaText specification to Promela. The Promela generator consists of rules written in Xtend similarly to the validation rules for the DSL. The rules are directly derived from the formal translation function proposed in Sec. 5.2. They take a HelenaText file containing a particular ensemble specification as input and generate the corresponding Promela file containing the translated process definitions.

In the following, we show an excerpt of the translation rules expressed in Xtend. We focus on role types and their behaviors only. The translation function is formally defined in Sec. 5.2. Its complete implementation in Xtend has a size of 2568 lines of code and can be found on the attached CD in the project `eu.ascens.helenaText` in package `eu.ascens.generator.promela`.

8.3.1 Generator Rule for Role Types

Helena role types are translated to Promela processes which actively communicate with component processes to rely on the capabilities of the components, e.g., to create other role instances or call operations, and with other role processes to exchange messages. They furthermore store the values of role attributes and represent the corresponding role behavior. The formal translation function for a role type is given in Fig. 5.12 on page 94.

For the code generator, this function is directly expressed with template expressions in the Xtend function `compileProctype` as shown in Fig. 8.11. The function `compileProctype` is called for any role type given in a HelenaText specification and

generates the corresponding process type in PROMELA. Basically anything in the function `compileProctype` between `"'` and `"'` is written to the generated PROMELA file except text enclosed in tag brackets «» which must be evaluated first. For example, in line 4 the head of the process type declaration is built. The name of the process type is dynamically evaluated from the expression «`rt.name`». This is a function of `RoleType` which is called for the first parameter `rt` of the function (see line 1) and retrieves the name of the role type `rt` (the resulting head of the process type for the role type `Router` is shown in line 376 of Appendix C.2).

The rest of the template expressions of the function `compileProctype` directly correspond to the formal translation function in Fig. 5.12 on page 94: Line 5–7 declare local variables for all role attributes of the role type, line 9–11 for all role instance variables used in the role behavior of the role type, line 13–15 for all data parameters of the role behavior, and line 17–19 for all return values of operations. Finally, the start state label is generated in line 21, the complete role behavior is translated to PROMELA by calling the function `compileRoleBehavior` in line 23, and the end state label is added in line 25. The complete generated process type for the role `Router` can be found in Appendix C.2.

```
1  def compileProctype(RoleType rt, Model model) {
2    var rb = model.headPkg.roleBehaviors.findFirst[it.roleTypeRef == rt]
3    '''
4    proctype «rt.name»(chan owner, self) {
5      «FOR attr:rt.roleattrs»
6      «attr.type» «attr.name»;
7      «ENDFOR»
8
9      «FOR inst : rb.abstractRoleInstances»
10     chan «inst.name»;
11     «ENDFOR»
12
13     «FOR param : rb.formalDataParams»
14     «param.type» «param.name»;
15     «ENDFOR»
16
17     «FOR op : rb.operationCalls»
18     «op.operationType.returnType» «op.variable.name»;
19     «ENDFOR»
20
21     «rt.startLabel» : true;
22
23     «rb.compileRoleBehavior»;
24
25     «rt.endLabel» : false
26   }
27   '''
28 }
```

Figure 8.11: XTEND generation rule for role types from HELENATEXT to PROMELA

8.3.2 Generator Rule for Role Behaviors

The role behavior of a role type is generated as body of the PROMELA process type declaration for the role type. Each HELENA process construct is expressed by PROMELA representations and actions are simulated by message exchange on channels either between role and component or between two roles. A role behavior is translated to its PROMELA representation by the function `compileRoleBehavior` which only translates its defining process expression and is thus not shown here. The translation of process expressions is however inductively defined as shown in excerpts in Fig. 8.12. Each

`dispatch` variant of the function `compileProcExpr` specifies the translation rule for a specific process construct. The correspondence of the XTEND functions to the formal translation functions in Fig. 5.13 on page 96 can easily be seen: For role termination with the **quit**, the `dispatch` function in line 1–8 generates the corresponding PROMELA code. Line 3 generates a new local variable for the request to the component to quit the role, line 4 sets the type of the request to quitting the role and line 5 generates the transmission to the owning component. Finally, a `goto`-statement to the end label of the role is created in line 6. For action prefix, the `dispatch` function in line 9–14 advises the generator to first compile the action with a closing semicolon (line 11) and then to compile the remaining process expression (line 12) similarly to the formal translation function in Fig. 5.13. The translation of all further process expression constructs is not shown here since they can be directly derived from he formal translation function and expressed in XTEND. Similar functions are also defined for the translation of actions and guards.

```
1  private def dispatch CharSequence compileProcExpr(RoleType rt, QuitTerm expr) {
2    '''
3    «rt.ownerComponentType.operationTypeName» op;
4    op.optype = «rt.quit»;
5    owner!op;
6    goto «rt.endLabel»
7    '''
8  }
9  private def dispatch CharSequence compileProcExpr(RoleType rt, ActionPrefix expr) {
10   '''
11   «expr.action.compileAction»;
12   «expr.processExpr.compileProcExpr»
13   '''
14 }
15 ...
```

Figure 8.12: XTEND generation rule for process expressions from HELENATEXT to PROMELA

The translation functions for role behaviors, process expressions, actions and guards together generate the body of the PROMELA process type for a role type according to its role behavior. The complete generated process type for the role `Router` can be found in Appendix C.2. As a side-note, the generated PROMELA file corresponds to the formal translation function given in Sec. 5.2, but it uses abbreviation macros at some points, e.g., for message exchange such that the generated PROMELA file is better readable.

Although the code generator translates the complete ensemble specification to PROMELA, it still remains to prepare the PROMELA translation for model-checking with Spin. As explained in Sec. 5.3.1, an initial state has to be established in the dedicated `init`-process and HELENA LTL formulae have to be translated to PROMELA LTL.

8.4 Automated jHELENA Code Generator

In Chap. 7, we proposed the Java framework jHELENA to make HELENA ensemble specifications executable. To facilitate the realization of HELENA ensemble specifications with jHELENA even further, this subsection introduces an automatic code generator translating a HELENATEXT specification to Java code relying on the jHELENA framework. The jHELENA generator consists of rules written in XTEND similarly to the

validation rules for the DSL. It takes a HELENATEXT file containing a particular ensemble specification as input and generates a package for the ensemble application which is split into two parts, the (sub)packages `src-gen` and `src-user`. The package `src-gen` is already complete and must not be touched anymore while the package `src-user` offers templates which must be implemented by the user. In the following, we first explain the idea of the splitting of the generated code into two packages before finally introducing the rules defining the JHELENA generator. The complete implementation of the JHELENA generator in XTEND has a size of 3368 lines of code and can be found on the attached CD in the project `eu.ascens.helenaText` in package `eu.ascens.generator.jHelena`.

8.4.1 Package `src-gen`

Nearly all parts of the realization of an ensemble specification relying on the JHELENA framework can be generated from its HELENATEXT specification. On the one hand, we can generate all contributing types and their conceptual relationships described by an ensemble structure. We introduce corresponding classes for all contributing types and a system manager which initializes all types and ensemble structures. On the other hand, role behaviors which represent the dynamic behavior of the ensemble can also be automatically translated to JHELENA. The generated role behaviors use the JHELENA representations of process constructs and instantiate the JHELENA actions appropriately relying on the introduced classes for component and roles as well as messages and operations.

For the p2p example, the generated package `p2p` with is shown in Fig. 8.13. In comparison to Fig. 7.2 on page 160 where we explained the implementation of the p2p example by hand, the package `p2p` is now split into two parts: the package `src-gen` contains only classes which is completely generated from the HELENATEXT specification; the package `src-user` provides base classes where the user implements the parts which cannot be generated from the ensemble specification. Let us focus on the package `src-gen`. It contains the generated subclasses for the abstract base classes of the `developer-interface`. These subclasses, like `Peer`, `Requester`, `Router`, and `Provider`, correspond to the types of the given ensemble structure. They implement the structural composition of a `TransferEnsemble` as well as the dynamic behavior of all roles as explained in Sec. 7.5. The generated `P2PSysManager` implements the method `configureTypes` to create objects for the `metadata` classes which represent types and the ensemble structure in accordance with the p2p ensemble specification (cf. Fig. 7.9 on page 175).

8.4.2 Package `src-user`

Only two parts which are necessary for the implementation of an ensemble-based system cannot be specified with HELENATEXT: the effect of operations and the initial state for an ensemble (note that the effect of operations is left unspecified in the formal HELENA syntax as well). To allow the user of the HELENA workbench to implement these two missing parts, we generate an additional package `src-user` which contains implementation classes for the missing parts. For each component type underlying the ensemble specification, we generate an implementation class which contains empty stubs for the implementation of its operations offered to the adopted roles of the component. Furthermore, implementation classes for the system manager and for the ensemble are created. They allow to implement a concrete initial state for an ensemble and to start an ensemble.

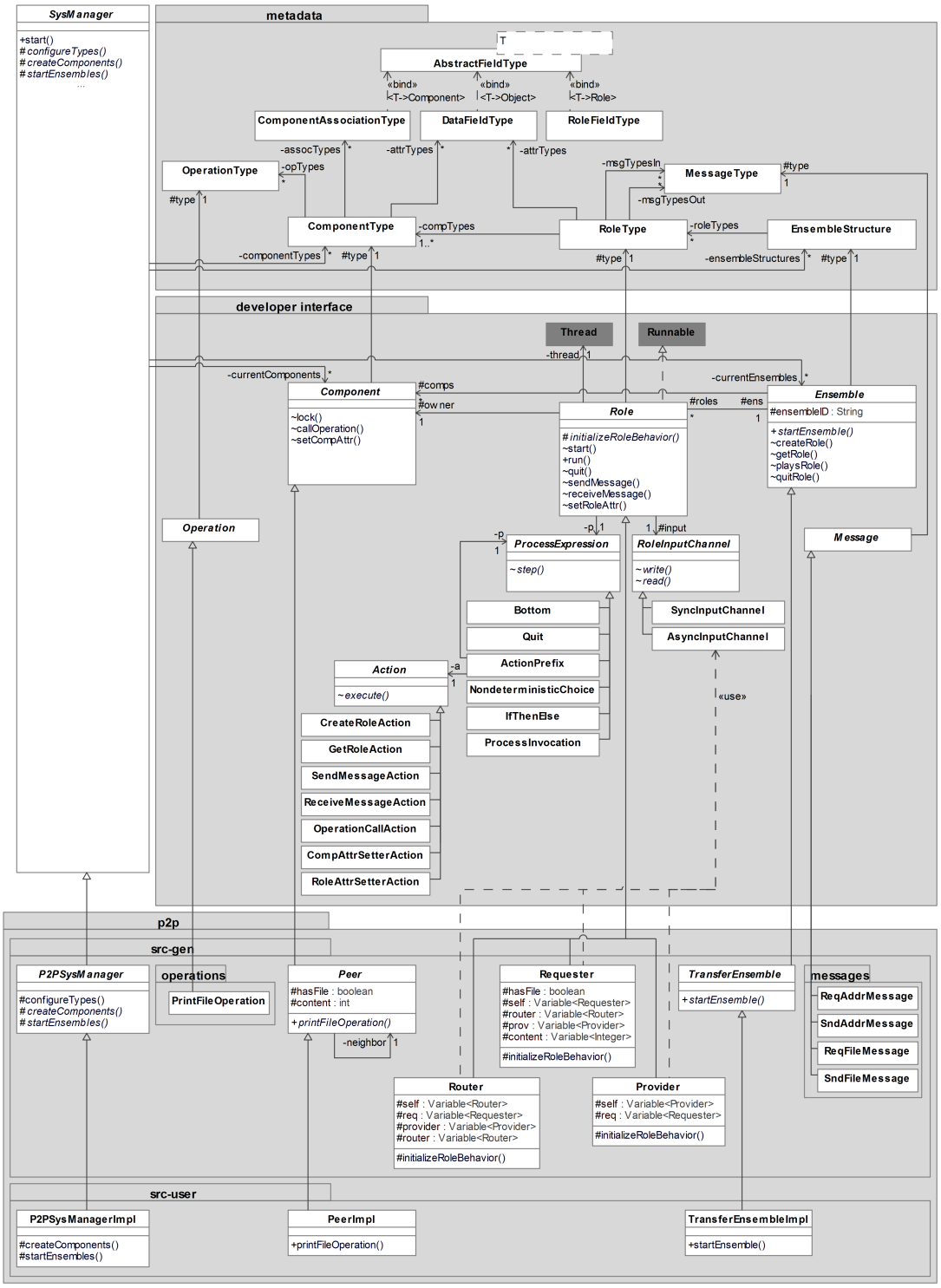


Figure 8.13: jHELENA and its application to the p2p example (generated)

Let's revisit the p2p example once again. The lower part of the `p2p`-package in Fig. 8.13 contains the subpackage `src-user`. It contains three classes: `P2PSysManagerImpl`, `PeerImpl`, and `TransferEnsembleImpl`. They are generated with empty code stubs for all declared methods and need to be implemented by the user of the HELENA workbench. In the class `PeerImpl`, the user has to add the behavior of the operation `printFileOperation` which is not defined in the HELENA (and resp. HELENA-

TEXT) specification. Additionally, he has to implement the methods `createComponents`, `startEnsembles`, and `startEnsemble` in the classes `P2PSysManagerImpl` and `TransferEnsembleImpl` to initialize the ensemble for file transfer as explained in Sec. 7.5.

8.4.3 Generator Rules

The rules for the translation to jHELENA were not formalized in the previous chapters like for the translation to PROMELA. However, the ideas of implementing a concrete ensemble-based application can be easily seen from the implementation of the p2p example in Sec. 7.5. Following these ideas, we specify the rules of the jHELENA generator as template expressions in XTEND. As shown in the previous section about the PROMELA generator, the XTEND rules are verbose enough to serve as formal translation rules (at least a draft of them). Similarly to the previous sections, we will not present all translation rules here, but focus on roles and their behaviors.

Fig. 8.14 shows an excerpt of the XTEND translation rule for the generation of role classes. The operation `compile` is called for any role type given in a HELENATEXT

```
1   def compile(RoleType it) {
2     '''
3     public class «it.classname» extends Role {
4
5       «FOR field : it.roleattrs»
6       protected «field.type» «field.name»;
7       «ENDFOR»
8
9       protected final Variable<«it.classname»> self =
10      new Variable<>(«it.classname».class);
11      «FOR inst : it.roleBehavior.abstractInstances»
12      protected final Variable<«inst.type»> «inst.name» =
13      new Variable<>(«inst.name».class);
14      «ENDFOR»
15
16      public «it.classname»
17      (Component comp, Ensemble ens, Integer capacity) {
18        super(comp, ens, capacity);
19        this.self.setValue(this);
20      }
21
22      @Override
23      protected ProcessExpression initializeRoleBehavior() throws ... {
24        «it.roleBehavior.compileRoleBehavior»
25      }
26      '''
27    }
```

Figure 8.14: XTEND generation rule for role types from HELENATEXT to jHELENA

specification and generates the corresponding class declaration in jHELENA. Basically anything in the operation `compile` is written to the generated class file except text enclosed in tag brackets «» which must be evaluated first. For example, in line 3 the class-header is built. The name of the class is dynamically evaluated from the expression «`it.classname`». This is a function of `RoleType` which is called for the first parameter `it` of the operation (see line 1) and retrieves the name of the role type `it` (the resulting class-header for the role type `Router` is shown in line 1 of Fig. 7.10 on page 176). Afterwards, in line 5-7 of the XTEND rule, all attributes of the role type are generated as instance variables (which are none for the role type `Router` in Fig. 7.10). Lines 9-14 declare additional instance variables for the **self**-reference and any parameters of

incoming messages or created role instances in the role behavior of the role type such that their values can be accessed throughout the execution of the role behavior. For example, for the role behavior of the `Router`, we need instance variable to store the values of the **self**-reference, of the parameter `req` of the incoming message `reqAddr` as well as of the two created role instances `prov` and `rout` (cf. line 2–5 in Fig. 7.10). A constructor for the class is generated in line 16–19 which sets the variable for the **self**-reference accordingly. For the role behavior itself the function `initializeRoleBehavior` is generated in line 21–24. To this end, the method `compileRoleBehavior` is called whose idea is explained in the following together with the generation of process expressions.

The rule for the generation of role behaviors is not shown here, but its basic idea is to generate its defining process expression possibly with some auxiliary processes (an example for those auxiliary processes are the processes `Provide`, `Fwd`, and `Create` of a router in Fig. 8.10). Though, we show the translation function for process expressions in Fig. 8.15. It is inductively implemented according to the definition of process ex-

```
1   private def dispatch String compileProcExpr(QuitTerm nil) {
2     '''Quit.getInstance()'''
3   }
4   private def dispatch String compileProcExpr(ActionPrefix actionPrefix) {
5     '''
6     «IF actionPrefix.action instanceof Label»
7     «actionPrefix.processExpr.compileProcExpr»
8     «ELSE»
9     new ActionPrefix(
10    «actionPrefix.action.compileAction»,
11    «actionPrefix.processExpr.compileProcExpr»
12    )
13    «ENDIF»
14    '''
15  }
16  private def dispatch String compileProcExpr(IfThenElse condSel) {
17    '''
18    new IfThenElse(
19    «condSel.guard.compileRelation»,
20    «condSel.ifProcessExpr.compileProcExpr»,
21    «condSel.elseProcessExpr.compileProcExpr»
22    )
23    '''
24  }
25  private def dispatch String compileProcExpr(NondeterministicChoice choice) {
26    '''
27    new NondeterministicChoice(
28    «choice.first.compileProcExpr»,
29    «choice.second.compileProcExpr»
30    )
31    '''
32  }
33  private def dispatch String compileProcExpr(ProcessInvocation procInvocation) {
34    '''invoc«procInvocation.hashCode»'''
35  }
```

Figure 8.15: XTEND generation rule for process expressions from HELENATEXT to jHELENA

pressions in Def. 2.9 on page 25. Depending on the type of the process construct, i.e. `QuitTerm`, `ActionPrefix`, `IfThenElse`, `NondeterministicChoice`, or `ProcessInvocation`, the XTEND rules prescribe to create a new instance of the representation of the process construct in jHELENA. The parameters of the new instances are the compiled actions, guards or process expressions used in the process construct. The jHELENA generator

defines similar functions for them as for role behaviors and process expressions such that the translation can rely on them. In summary, the function `compileProcExpr` and similar variants for actions and guards generate the method `initializeRoleBehavior`. An excerpt of this method for the router of the p2p example is shown in Fig. 7.10 on page 176. As a side-note, the jHELENA code generator produces exactly the same implementation of the class `Router` as we proposed it by manual implementation in Sec. 7.5.

8.5 Publication History

This chapter brings the master thesis [Cic14] and the two publications [Kla15b], and [KCH14] together. The overview of the HELENA workbench is based on [Cic14], but goes more into detail about the different activities and artifacts.

The basis for the domain-specific language HELENATEXT has been developed in the master thesis [Cic14] under the supervision of Annabelle Klarl. However, this thesis extends HELENATEXT to be able express all syntactic constructs presented in Chap. 2. Especially, role behaviors are now described by process expressions and not by transitions of a labeled transition system.

The automated PROMELA code generator has already been presented in [Kla15b]. Compared to this publication, this thesis improves the code generation to precisely follow the formal translation proposed in Sec. 5.2. Special care has been taken to represent atomic steps correctly.

The automated jHELENA code generator has already been presented in [Cic14] and [KCH14]. Since the jHELENA framework (and especially the representation of role behaviors) has fundamentally been improved in this thesis, the code generator has rudimentary been revised in this thesis. The translation of ensemble structures is only slightly adapted, but the translation of role behaviors has completely been reimplemented.

8.6 Present Achievements and Future Perspectives

Present Achievements: This chapter proposed the HELENA workbench, an Eclipse plug-in which supports the developer of ensemble-based systems throughout the whole development process. The HELENA workbench provides the domain-specific language HELENATEXT to be able to specify ensembles with appropriate first-class concepts. HELENATEXT relies on the XTEXT workbench of Eclipse which allows to automatically provide an Eclipse editor for writing HELENATEXT specifications. It gives the user assistance for writing ensemble specifications and checks the user-defined models for validity according to the formal HELENA definitions. Additionally, two code generators are implemented which take a HELENATEXT specification as input and translate them to PROMELA and Java. The two generators realize the translations proposed as manual translations in Chap. 5 and Chap. 7, but allow reliable and fast development through the automation of the translation.

Future Perspectives: Future work will be driven by extension of the domain-specific language HELENATEXT.

Syntactic Extension of HELENATEXT*:* At the moment, it is only possible to specify an ensemble specification, but not its initial state or its goals. If we offered these

possibilities, the two code generators could be extended such that the developer of the ensemble does not have to specify anything in generated files anymore. For the translation to PROMELA, it would be possible to generate the initialization of processes given in the dedicated `init`-process (cf. Sec. 5.3.1) as well as to translate goals given as the HELENA LTL formulae directly to PROMELA LTL. For the translation to Java relying on the jHELENA framework, it would be possible to generate the set-up of an initial state in the system manager of the ensemble-bases system. Furthermore, the whole HELENA methodology could allow the specification of effects of component operations. If they were included in a HELENATEXT specification, it would be possible to also generate the implementation of operations of components such that the Java implementation could be completely generated without manual intervention by the developer of the ensemble-based system.

Syntactic Sugar for HELENATEXT*:* During the development of the p2p example as well as our case study which will be presented in Chap. 10, we noticed some handy extension of the domain-specific language which would ease specification. On the one hand, some syntactic sugar could be added to allow an **if**-construct or nondeterministic choice with several options. On the other hand, the graphical representation of ensemble structures used throughout this thesis helps to get a better overview of the structure of an ensemble. The HELENA workbench already provides the metamodel of HELENATEXT according to the Eclipse Modeling Framework. Thus, it should be easy to extend it with a graphical notation based on the Graphical Modeling Framework of Eclipse.

Representation of Counterexamples in HELENATEXT*:* Finally, it would be interesting to map the results of verification and execution back to the original HELENATEXT specification. This is particularly interesting for counterexamples generated during verification which are currently represented as traces of the PROMELA verification model. They have to be mapped back to HELENA by hand which requires in-depth knowledge of the translation from HELENA to PROMELA.

Chapter 9

HELENA Development Methodology
Developing with HELENA

The HELENA techniques and tools support different activities of engineering ensemble-based systems. This chapter is concerned with bringing all techniques and tools presented in the previous chapters together and integrates them in a holistic development methodology. Fig. 9.1 gives an overview of the HELENA development methodology. The methodology splits development work into five distinct phases (shown as dashed boxes): *domain modeling*, *goal specification*, *design*, *verification*, and *implementation*. Each phase is concerned with certain activities (shown as boxes with rounded corners) and produces certain artifacts (shown as boxes with pointed corners). The intentions of the phases in the HELENA development methodology do not differ from other classical development methodologies since each activity is similarly relevant for ensemble-based systems as for other software applications. That means that there is no need to introduce a whole new methodology for the development of ensemble-based system, but rather to tailor the techniques and tools for each phase to the particular characteristics of ensemble-based systems. Thus, we introduce the HELENA development methodology as a classical model-driven approach for engineering ensemble-based systems with systematic transitions between all phases. Most of the phases and transitions are tool-supported and (except verification with Spin) seamlessly integrated into one single tool, the HELENA workbench (cf. Chap. 8).

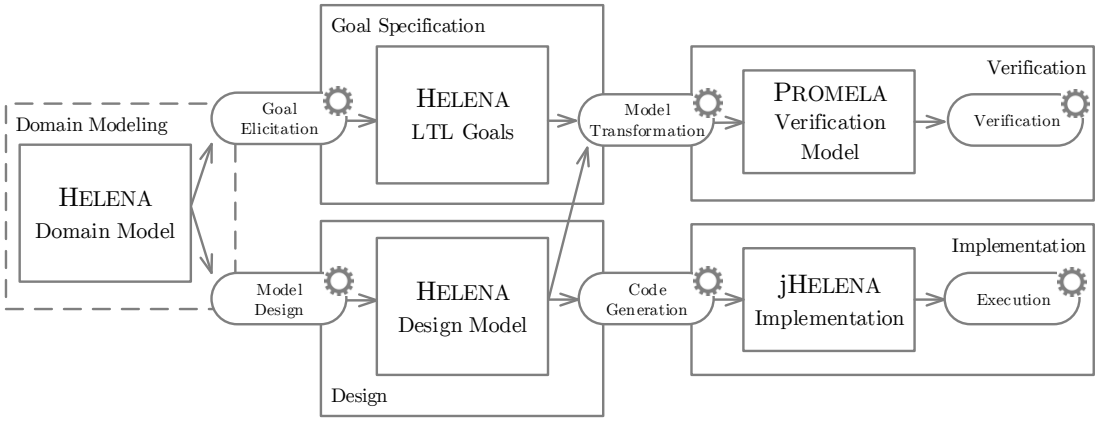


Figure 9.1: The HELENA development methodology for ensemble-based systems (repeated from Sec. 1.5)

The HELENA methodology advises to start with modeling the domain of the ensemble-based application. An ensemble structure captures the participating entities of the ensemble and serves as the HELENA *domain model*. Based on this model, goals for the envisioned ensemble are elicited and specified as HELENA *LTL formulae*. Afterwards (or possibly in parallel), a goal-oriented HELENA *design model* is derived. It enhances the initial HELENA domain model by further role types which might be needed to accomplish the global goal of the ensemble, but are not explicitly relevant for the domain description. Furthermore, all component types of the initial HELENA domain model are enhanced by operations to allow use of their resources, all role types are enhanced by messages to allow communication between roles and by goal-directed role behaviors which use the aforementioned messages for collaboration between roles. This enhancement produces a complete HELENA ensemble specification serving as the HELENA design model. To assess whether the designed ensemble specification actually achieves the specified goals, the HELENA LTL goals and the HELENA design model are transformed to a PROMELA *verification model*. This PROMELA verification model is verified with the model-checker Spin to get feedback on the satisfaction of goals in the HELENA design model and to improve the model further to finally achieve all goals. Furthermore, a jHELENA implementation is generated from the HELENA design model. This implementation allows to execute the ensemble-based system in Java relying on the jHELENA framework.

When examining the overview diagram of the HELENA development methodology, one might be tempted to envision it as a classical waterfall methodology. We rather have an *iterative* and *incremental* development in mind, but did not show feedback loops and increments in the picture.

- Iterative development means that we continuously improve the system design. In the case of HELENA, a first draft of the goal specifications and the design model is created and analyzed with Spin. As long as verification does not return the desired positive results concerning goal satisfaction, goal specifications and the design model are refined until the goals specifications finally express the intended goals and the design model achieves all intended goals. Similarly, the design model is improved until the implementation satisfies all execution demands like performance requirements.

- Incremental development means that we develop the overall system in smaller parts or increments. In the case of HELENA, we incrementally design the ensemble-based system such that each increment of the system tries to achieve a certain subset of all intended goals and to fulfill its dedicated performance requirements. On ensemble-level, we focus on a particular subset of goals and develop only the roles and their behaviors which contribute to the achievement of these goals. On system-level, each ensemble is considered as an increment which runs on top of the underlying component-based platform and is developed individually.

- By combining iterative and incremental development, the system is created in increments each of which undergoes several refinement iterations until goal satisfaction is reached and execution requirements are fulfilled.

In the following sections, we describe the five phases of the HELENA development methodology separately. For each phase, we summarize (and repeat from the previous chapters) the activities, created artifacts, and tool support. We illustrate each step with our running example of the p2p network for exchanging files.

The complete specification of the p2p example in HelenaText can be found in Appendix C.1 and on the attached CD in the project `eu.ascens.helenaText.p2p` in the file `src/p2p.helena`. The generated Promela specification for verification together with all goals is shown in Appendix C.2 and on the attached CD in the project `eu.ascens.helenaText.p2p` in the file `promela-gen/p2p-check.pml`. The generated implementation of the p2p example based on the jHelena framework can be found on the attached CD in the project `eu.ascens.helenaText.p2p` in packages `src-gen` and `src-user`.

9.1 Domain Modeling

The first phase in the Helena development methodology is domain modeling. This phase is based on the techniques and tools of Chap. 2 and Chap. 8.

Activities: We capture the domain of the ensemble application to be developed. The domain model represents all entities which are important in the problem domain space and therefore provides the vocabulary to describe the ensemble application. It does not specify any behavior of the entities, but only their basic properties.

Artifacts: The domain is characterized by the underlying component-based platform providing its resources to the application. The component-based platform is represented by a set of component types with their mutual associations and the basic attributes they provide to store data. On top of that, the ensemble to be developed is described by its essential role types. The role types and their relationships to the underlying component-based platform are represented in an ensemble structure as introduced in Sec. 2.3.

We only model the problem domain in this phase. Thus, it is enough to name the contributing component types and essential role types together with their attributes. Component types are not yet equipped with operations and role types are not equipped with messages since only the specification of goals can later on guide the introduction of particular actions and behaviors.

Tool Support: The domain model with component types, role types and the ensemble structure can textually be specified in HelenaText. To this end, the Helena workbench provides a HelenaText editor with syntax highlighting, content assist and validation. Although the editor allows to specify operations for component types as well as messages and behaviors for role types, they are left unspecified since they are added during goal-oriented solution design.

Example: We illustrate the Helena development methodology at our p2p example. The employed p2p network supports the distributed storage of files that can be retrieved upon request. Therefore, the underlying component-based platform is described by a single component type `Peer` as graphically shown in Fig. 9.2 (for the textual representation, we refer to Appendix C.1). For simplicity, we only consider a single file to be stored in the network. Thus, the component type `Peer` only has two attributes `hasFile` and `content` to indicate whether it stores the file and to represent its content.

The problem to be solved by the envisioned ensemble is to work together to allow a requesting peer to retrieve the file from the network. This means that the role of a `Requester` is essential to describe the problem domain of the ensemble. It just has one attribute `hasFile` to indicate whether it already retrieved the file from the network. All

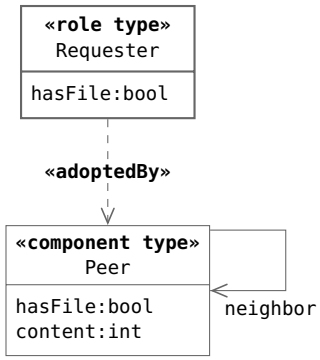


Figure 9.2: Domain model for the p2p example in graphical notation

other role types which are needed to transfer the file to the requesting peer are part of the solution design and are hence not represented in the domain model in Fig. 9.2.

9.2 Goal Specification

Following up is the phase of goal specification. This phase is based on the techniques and tools of Chap. 4.

Activities: In this phase, we use the domain model and the informal intuition about the problem to elicit formal goals. The goals formally represent the intentions of the ensemble under development. Requirements methods like KAOS [vL09] can be applied to systematically derive goals. Requirements are gathered by interviewing stakeholders and carefully analyzing the problem domain. Different requirements patterns [DvLF93, DAC99] and decomposition into subgoals [vL01] lead to a set of system goals which systematically describe the requirements of the system and allow to deduce the system behavior in the next phase.

A common tool for the actual goal description is linear temporal logic. It allows to specify temporal properties which can be achieved at some point in the runtime of the ensemble-based system (so-called *achieve goals*) or which are maintained during the whole runtime of the ensemble-based system (so-called *maintain goals*).

Artifacts: Each goal of the ensemble is specified as a linear temporal logic (LTL) formula. LTL formulae are built from a set of HELENA-specific atomic propositions and the usual relational, propositional, and temporal operators (cf. Sec. 4.2). There are two different types of HELENA-specific atomic propositions about which we can reason: Attribute expressions refer to the current value of an attribute of a component instance or role instance. They allow to specify goals for the problem domain. The second type of atomic propositions are state label expressions. They refer to certain states in the progress of executing a role behavior. Therefore, they cannot be used to initially describe goals since the domain model does not specify any role behaviors. However, they can later on be used as auxiliary goals when verifying and refining the design model.

Tool Support: The specification of goals is so far not supported by the HELENA workbench. It remains future work to allow the formulation of LTL properties based on the HELENA-specific atomic propositions.

Example: For our p2p example, we derive two goals. The first goal is an achieve goal and expresses that the `Requester` finally receives the file if the file is present in the network (cf. Fig. 9.3). The formalization makes the drawback of using LTL as the logic for goal specification apparent. Since LTL does not support quantifiers, we have to enumerate all peers in the network to specify the precondition that the file has to exist in the network. This means that we must already assume during goal specification that the system only employs three peers in the case of our p2p example.

$$(Peer[1]{:}hasFile \vee Peer[2]{:}hasFile \vee Peer[3]{:}hasFile) \Rightarrow \Diamond Requester{:}hasFile.$$

Figure 9.3: Achieve goal for the p2p example in HELENA LTL

The second goals is a maintain goal (cf. Fig. 9.4) and expresses that the file is never removed from the network if the file is present in the network. As before, we have to enumerate all peers of the underlying platform to be able to describe this goal in LTL.

$$\begin{aligned}&(Peer[1]{:}hasFile \vee Peer[2]{:}hasFile \vee Peer[3]{:}hasFile)\\ &\quad \Rightarrow \Box(Peer[1]{:}hasFile \vee Peer[2]{:}hasFile \vee Peer[3]{:}hasFile).\end{aligned}$$

Figure 9.4: Maintain goal for the p2p example in HELENA LTL

9.3 Design

The main phase for the derivation of the solution is the design of the envisioned ensemble-based system. This phase is based on the techniques and tools of Chap. 2 and Chap. 8.

Activities: This phase is concerned with the design of the goal-directed behavior of the system. To this end, we have to equip component types with operations which allow to use the resources of the components. Furthermore, role types have to be provided with messages which they can exchange to collaborate on the task of the ensemble. To be able to achieve the specified goals, it might be necessary to even include more role types. These role types are not essential to the description of the domain and were therefore not part of the initial domain model. They only contribute to the solution of the task of the ensemble. For each role type, we additionally specify how many instances are allowed and have to contribute to the ensemble and how many message can be held by the input queue of each instance.

Lastly, a role behavior for each role type needs to be designed such that the overall ensemble works towards its intended goals in collaboration and interaction. We recommend to prevent communication errors between the collaborating roles by visualizing the interaction behavior of the ensemble in a notation similar to sequence diagrams. This interaction diagram allows to design the collaboration between all contributing roles of the ensemble together. Projection on a single role gains the desired role behavior for each role type.

Artifacts: Resulting from the extension of component types by operations and of role types by messages as well as the inclusion of further role types and the specification of

role behaviors is a HELENA design model in the form of a complete ensemble specification in the sense of Chap. 2. It relies on a set of component types with operations. On top of that, an ensemble structure specifies the conceptual relationships between collaborating roles and a set of role behaviors determines the behavior of the ensemble.

Tool Support: The HELENA design model can completely be specified textually in HELENATEXT. The corresponding editor in the HELENA workbench provides once again syntax highlighting, content assist and validation of the specified model.

Example: Fig. 9.5 depicts the full ensemble structure for our p2p example in graphical notation (for the textual representation, we refer to Appendix C.1). The design model includes the component type `Peer` and the role type `Requester` from the initial domain model, but it enhances them by operations and messages (depicted by arrows between role types) as well as by a multiplicity and a capacity of the input queue for the role type `Requester`. Furthermore, the ensemble employs two more role types `Router` and `Provider` to work towards the goal of file transfer in collaboration.

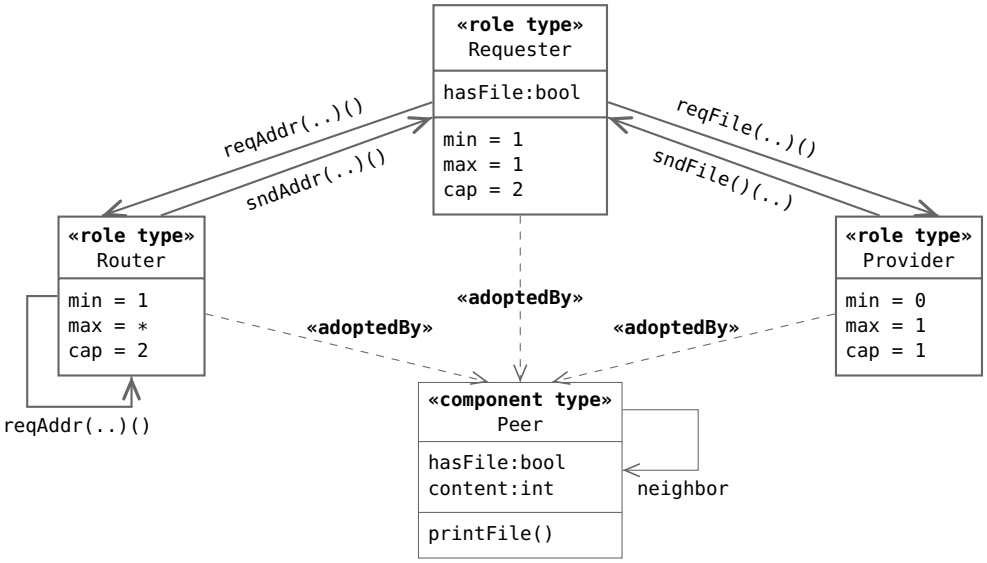


Figure 9.5: Ensemble structure in the HELENA design model for the p2p example in graphical notation

Apart from the ensemble structure, the design model is also comprised of a role behavior for each role type in the ensemble. These interacting role behaviors can be developed by visualizing them in a graphical notation similar to sequence diagrams (cf. Fig. 9.6). Afterwards, they are projected on each role type to derive the individual role behaviors. Thereby, the two instances of role type `Router` are integrated into one role behavior which calls itself recursively as indicated by the recursion box in Fig. 9.6. The projected role behaviors were already presented in Sec. 2.4. The complete specification in HELENATEXT has 72 lines of code and is listed in Appendix C.1.

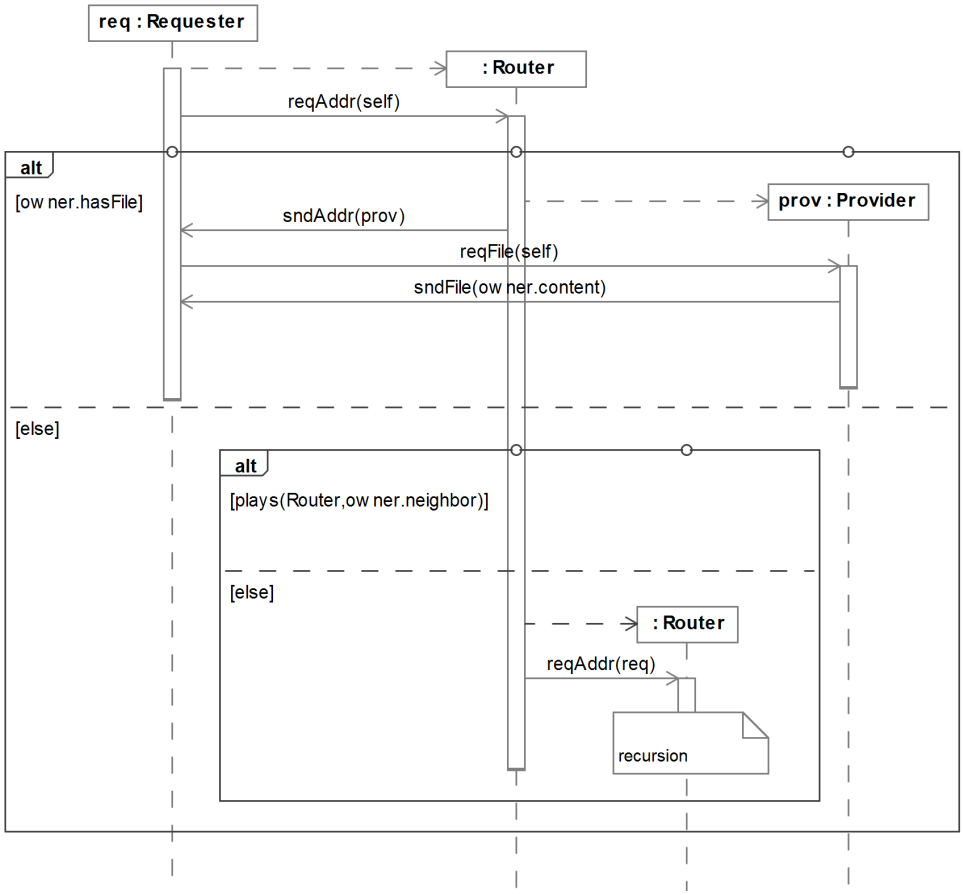


Figure 9.6: Role behaviors in the HELENA design model for the p2p example in graphical notation

9.4 Verification

The design model for the envisioned ensemble-based system is developed having the specified goals in mind. To actually guarantee that it achieves the specified goals, the model has to be verified against its goals. For this verification phase, we rely on the well-established explicit state model-checker Spin [Hol03] instead of providing a HELENA-specific model-checker. The phase is based on the techniques and tools of Chap. 5 and Chap. 6.

Activities: The verification phase is mainly characterized by two activities: Firstly, the HELENA design model and its goals specified as HELENA LTL formulae are transformed to a PROMELA verification model, the input for the model-checker Spin. We support this activity by a formal translation function from HELENA to PROMELA (cf. Sec. 5.2) which is proven semantically correct in Chap. 6.

Secondly, the PROMELA verification model is checked for a specific initial state against the translated goals with the model-checker Spin. This verification step gives feedback to the phase of goal specification on the one hand and to the phase of design on the other hand. Goals are refined if they are not strong enough or restrict the solution design too much. The design model is revised to get rid of design flaws which prevent the model from achieving the specified goals. This feedback loop emphasizes

the iterative character of the HELENA development methodology insofar that the results of model-checking help to improve the artifacts of previous phases.

Artifacts: Two artifacts are required as input: The goals for the envisioned ensemble-based system are described as HELENA LTL formulae. The structure and the goal-directed behavior of the ensemble-based system are captured as a HELENA design model consisting of an ensemble structure with associated role behaviors on top of a component-based platform.

These two artifacts are transformed to a PROMELA verification model. For the HELENA design model, the two layers of components and roles are represented as processes in PROMELA. Instead of directed message exchange, channels allow the exchange of data between role processes. Furthermore, the goals for the system specified as HELENA LTL formulae are translated to PROMELA LTL formulae. Their main difference is to refer to properties of the PROMELA process types instead of HELENA components and roles.

Tool Support: Based on the design model specified in HELENATEXT, the HELENA workbench offers an automated model transformation to PROMELA. The transformation implements the formal translation function from Sec. 5.2 as described in Sec. 8.3. Its output is a PROMELA file representing the HELENA design model which can be used as input for the model-checker Spin. So far, the translation of HELENA LTL formulae to PROMELA LTL formulae is not supported by the HELENA workbench and has to be done by hand.

Model-checking with Spin is not integrated into the HELENA workbench. However, the generated PROMELA file can directly be checked for a specific initial state against the hand-coded PROMELA LTL formulae with Spin. The output of Spin provides results and statistics about the verification run. Counterexamples can be examined on the level of the PROMELA verification model if goals are not satisfied. It remains future work to map these counterexamples back to the original HELENA design model and therefore to integrate the verification run into the HELENA workbench.

Example: For verification, the HELENA design model for the p2p example is transformed to a PROMELA verification model with four process types representing the component type `Peer` and the role types `Requester`, `Router`, and `Provider`. Excerpts were already shown and explained in Fig. 5.5 on page 85 and Fig. 5.6 on page 87. The complete generated PROMELA verification model has 512 lines of code and is listed in Appendix C.2.

Furthermore, both goals for the p2p example are translated to PROMELA as already discussed in Sec. 5.3.1. To represent the HELENA LTL formulae in PROMELA, very few adaptations are needed: All operators are expressed by their ASCII representatives. To refer to the role attribute *hasFile*, it is prefixed with `roleAttr_`. Everything else remains unchanged. The translated goals are listed in Fig. 5.18 on page 104 and Fig. 5.19 on page 104 or at the end of Appendix C.2 and are not repeated here.

To model-check the generated PROMELA verification model against its translated PROMELA goals, we finally set-up an initial state for the model as already shown in Fig. 5.17 on page 103. The `init`-process in Fig. 5.17 creates three peers (line 2–19), each possibly having the desired file. Afterwards, it starts the envisioned ensemble by creating a requester on the first peer (line 21–27).

With the initial state from Fig. 5.17, both goals of the p2p example are satisfied for the design model. Due to the small size of the example, it was possible to design a correct

solution model from the beginning such that no improvement iterations were needed. However, this changes quickly if the example gets more complex. When developing the case study of the Science Cloud platform in Chap. 10, we will soon benefit from the verification of the design model. Design flaws and deadlocks in the structural model and the collaboration behaviors become apparent before implementation. They can be eliminated at early stages and the later implementation is guaranteed to achieve its goals. Some of the errors, we detect in the design of the case study, are corner-cases which most certainly would not have been uncovered by simple tests of the final implementation.

9.5 Implementation

The final phase of the HELENA development methodology realizes the verified design of the ensemble-based system and allows to execute it. This phase is based on the techniques and tools of Chap. 7 and Chap. 8.

Activities: The main activities of this phase are implementation and execution of the ensemble-based system. To ease implementation, we proposed the Java framework jHELENA in Chap. 7. It expresses the two-layered HELENA concepts with object-orientation in Java and realizes the execution semantics of HELENA. A specific ensemble-based system is thus easily implemented by relying on the general classes provided by jHELENA.

To execute the implemented system, a specific initial configuration of the system is set up, i.e., it is defined which components exist and how the ensemble is started. The jHELENA framework then takes care to execute the implemented system according to the HELENA semantics.

Artifacts: As for verification, the HELENA design model serves as input for the implementation and builds the foundation for execution. The structure and the goal-oriented behavior of the ensemble described in this model are transferred to an implementation in the form of a set of classes relying on the jHELENA framework. jHELENA thereby represents all syntactical concepts of HELENA in Java, may it be entities like components, roles and ensemble structures or behavioral constructs like process expressions and actions. It furthermore realizes the execution semantics of HELENA for all represented concepts. By building on jHELENA, we thus assure that the main artifact of this phase, the implementation as a set of classes, only refers to HELENA concepts and that its execution respects the semantics of HELENA.

Tool Support: Based on the design model specified in HELENATEXT, the HELENA workbench offers an automated code generator to Java. The code generation translates all HELENA concepts to their counterparts in the jHELENA framework as described in Sec. 8.4. Its output is a set of classes representing the HELENA design model relying on the jHELENA framework.

As for verification, a specific configuration of the ensemble-based system has to be established before the ensemble can be started. So far, it has to be implemented by hand which components exist and how the ensemble is started. The HELENA workbench supports the implementation of an initial state in Java (or rather jHELENA) relying on the standard Java editor of Eclipse with syntax highlighting and content assist. It

remains future work to support the specification of an initial state in HelenaText in the Helena workbench.

The generated set of classes enhanced by an initial state can then be executed with the Java virtual machine. By relying on the Java virtual machine of Eclipse and providing the jHelena framework, execution is fully integrated into the Helena workbench including the well-established Eclipse features of debugging and code inspection.

Example: The generated implementation for the p2p example is split into two parts. The first part resides in the package `src-gen` and includes all classes whose content can directly be derived from the Helena design model and which do not have to be adapted by the user anymore. It contains 11 classes: one class represents the component type `Peer`, three classes represent the role types `Requester`, `Router` and `Provider`, one class represents the ensemble structure `TransferEnsemble`, four classes represent the four messages exchanged in the ensemble, one class represents the only operation `printFile` provided by the component type `Peer` and finally one class represents the system manager which takes care to configure the ensemble structure and its contributing role types. Excerpts of the system manager and the class representing the role type `Router` were already shown in Fig. 7.9 on page 175 and Fig. 7.10 on page 176. The complete generated implementation can be found on the attached CD in the project `eu.ascens.helenaText.p2p` in package `src-gen`.

For the implementation, a second part cannot be generated from the Helena design model, but needs to be implemented by hand in stub-classes in the package `src-user`. On the one hand, the effect of operations is not defined in Helena. Therefore, the effect of an operation needs to be implemented in the generated component class (we refer the reader to Fig. 7.11 on page 177 for the implementation of the operation `printFile`). On the other hand, the initial state of the ensemble has to be defined, i.e., which components exist and how the ensemble is started. All peer components are created and connected in a ring structure in the method `createComponents` implemented by the class `P2PSysManager` which was already shown in Fig. 7.13 on page 178. To start transfer ensembles on top of this ring structure of peers, the method `startEnsembles` is implemented as already shown in Fig. 7.14 on page 178. The complete implementation can be found on the attached CD in the project `eu.ascens.helenaText.p2p` in package `src-user`.

9.6 Related Work

While the list of general development methodologies is quite long, only very few have been proposed which address the particular features of ensemble-based systems.

SCEL [DLPT14] provides linguistic abstractions to design autonomic computing systems. With its execution framework jRESP [Lor16], SCEL specifications can be implemented and simulated. A prototypic statistical model-checker integrated into jRESP allows to check satisfaction of reachability properties with a certain degree of confidence. However, the authors claim the development of a coherent methodology for future work [DLPT14].

DEECo [BGH+13] is supported by a dedicated requirements-oriented design method called Invariant Refinement Method (IRM) [BGH+15]. It guides the transition from high-level system goals to the architecture of autonomic components and ensembles. IRM captures goals and requirements of the ensemble-based system to be developed by high-level invariants. These high-level invariants are gradually decomposed to lower-level invariants until they correspond to computation activities which can be expressed

by a component process or knowledge exchange according to the DEECo component model [BGH+13]. The DEECo design model can then be implemented with the Java framework jDEECo and verified with Java PathFinder [VHB+03]. Although IRM defines a comprehensive solution how to derive a DEECo architecture from high-level goals, the further activities of verification and execution are not embedded into a coherent development methodology.

The Scribble methodology [YHNN13] supports the development of distributed systems based on multiparty session types [CDPY15] from the specification of global interaction protocols to local verification and execution. The first step in the methodology is to specify the global interaction protocol type between participants of a multiparty session with the Scribble language. Afterwards, the Scribble tools automatically generate the local projection of the global type to every single participant of the multiparty session. Such local types preserve communication safety by construction. Then, the local types are implemented in a mainstream language like Java or Python using the Scribble Conversation API and are executed by the Scribble Runtime. In theory, the implementation of the local types preserves communication safety via a strict type system between the local types and their implementations. However, in practice, calls to native functions in the host languages Java or Python make the static type checks intractable [YHNN13]. Thus, a finite state automaton is generated from each local type which represents all action traces possible in the local interaction protocol type. The correspondence of the implementation with its local interaction protocol type is dynamically verified by monitoring and comparing the traces through the implementation and the traces through the finite state automaton generated from the local type. Compared to the HELENA development methodology, multiparty session type theory in general and the Scribble methodology do not allow to define specific goals to be achieved in a multiparty session and to verify them for a particular multiparty session type. They focus on general properties likes communication safety, protocol fidelity and progress. These properties are guaranteed by construction in theory and by dynamic verification in practice. Therefore, the Scribble methodology does not include a dedicated goal specification and verification phase.

9.7 Publication History

The HELENA development methodology has newly been proposed in this thesis and has not yet been published before. It combines all techniques and tools presented in the previous chapters to a coherent methodology. The contents of the individual phases rely on the publications already named in the corresponding chapters of this thesis.

9.8 Present Achievements and Future Perspectives

Present Achievements: The HELENA development methodology is a holistic engineering process for ensemble-based systems. It is characterized by five distinct phases which are iteratively and incrementally traversed: *Domain modeling* introduces all entities to describe the problem space of the system to be developed. During *goal specification*, the goals of the envisioned ensemble-based system are elicited and formalized. The *design* phase is responsible to derive a goal-directed solution model. i.e., concrete messages exchanged between roles are introduced and role behaviors are designed. To actually guarantee goal-directed behavior, *verification* checks that the design model

achieves the specified goals and gives feedback about design flaws. The final *implementation* phase realizes the verified design and allows to execute it.

Altogether, the phases of the HELENA development methodology have the same intentions as in classical development methodologies, but we tailor the employed techniques and tools to the particular characteristics of ensemble-based systems: The HELENA modeling approach provides particular concepts targeted at dynamically formed and collaborating ensembles. With a HELENA model at the heart of the HELENA development methodology, we support systematic transitions between all phases to allow model-driven development of ensembles. Most of the phases and transitions are tool-supported and seamlessly integrated into one single tool, the HELENA workbench, upgrading the HELENA development methodology to a holistic engineering approach.

Future Perspectives: Nevertheless, the HELENA development methodology leaves some areas for future improvements.

Artifacts: The expressiveness and (textual or graphical) representation of the artifacts can be advanced further. We refer the reader to the chapters Chap. 2, Chap. 3, Chap. 4, Chap. 5, Chap. 7, and Chap. 8 for detailed future work on artifacts in each of the phases. However, we want to highlight one missing feature which became apparent during the systematical application of the HELENA development methodology to the p2p example (and the larger case study presented in Chap. 10). When specifying the dynamic behavior of an ensemble, the designer mostly has the global interaction behavior of the ensemble in mind. HELENA, however, requires to separately specify the projected role behaviors. Thus, HELENA would benefit from an interaction specification, e.g., in the style of UML sequence diagrams or multiparty session types [CDPY15], which allows to first design the interactions between roles and afterwards to project them on separate role behaviors. When using UML sequence diagrams, standard modeling concepts would have to be extended to visualize recursive role behaviors and arbitrary many instances per role type.

Activities: Some additional activities are needed to foster the information flow between the phases and nurture a deeper guideline for the completion of each activity.

- The KAOS methodology [vL09] as well as Dwyer et al. [DAC99] recommend a set of patterns for the specification of temporal properties. This very general set of patterns already provides a guideline for the formulation of goals of ensemble-based systems, but it could be specialized to the particular characteristics of ensembles. For example, maintain goals should always be expressed on the level of components since they are the persistent entities of the system while roles are volatile. On the other hand, achieve goals should capture that the roles of the ensemble achieve a certain property from an initial state expressed on the level of components.
- So far, the derivation of a solution design from a goal specification is not supported by any specific guideline, but depends on the creativity of the designer. Thus, this activity could benefit from rules about how to refine goals to a design like in the Invariant Refinement Method [BGH+15].
- For detailed future work on the activities for verification and implementation, we refer the reader to the chapters Chap. 5, Chap. 7, and Chap. 8.

Chapter 10

HELENA@Work
Applying HELENA to the Science Cloud Platform

In the last sections, we formally introduced the HELENA methodology and all its techniques and tools and illustrated it with a small example for distributed file storage. In this section, we apply the HELENA methodology to a larger software system. We take our case study from the EU project ASCENS [WHKM15]. This project investigates ensemble-based systems as we envision them in the HELENA methodology. The case study we selected from this project is the *Science Cloud Platform (SCP)* [MKH+13]. The SCP is a platform of distributed, voluntarily provided computing nodes. The nodes interact in a peer-to-peer manner to execute, keep alive, and allow use of user-defined software applications. The goal of applying HELENA to the SCP is two-fold. On the one hand, we want to evaluate whether the HELENA methodology helps to systematically develop ensemble-based systems even in a larger context. On the other hand, we want to assess whether the HELENA concept of slicing the ensemble-based system into ensembles and roles is a reasonable abstraction that serves as clear documentation, analysis model for verification, and guideline for implementation.

We experienced that the HELENA methodology helps to rigorously describe the concepts of the SCP. A HELENA ensemble structure with its roles divides the domain into intuitive entities. Based on this domain model, goals can be easily specified and goal-directed behaviors be developed. During verification of the model against its goals with Spin, collaboration mismatches can be eliminated at early stages as well as goal satisfaction guaranteed. As we shall discuss, the implementation also benefits from the introduction of ensembles and roles. The ideas from jHELENA how to realize the HELENA concepts in object-orientation can be transferred to the implementation of the SCP. However, some additional effort is required to provide an infrastructure which supports the role concept in unison with the required technology stack of the SCP (e.g., the message passing framework of the SCP). Thus, inspired by jHELENA, a HELENA SCP middleware is implemented which transfers the HELENA concepts of roles and ensembles to the distributed set-up of the SCP and can therefore be seen as a prototype of a distributed jHELENA implementation, particularly tailored for nodes of the SCP. Lastly, special care has to be taken to make the system robust against communication failures and to provide communication facilities between ensembles and the outside world which are not yet tackled in HELENA.

In the following sections, we first describe the case study and its goals informally in Sec. 10.1. Afterwards, we follow the HELENA development methodology to realize the case study with HELENA. In Sec. 10.2, we introduce the domain model of the SCP.

Using this domain model, we deduce a formal specification of all goals of the case study in Sec. 10.3. Table 10.4 describes the dynamic behavior executed by the ensemble to achieve its goals. To guarantee goal satisfaction with this dynamic behavior, we verify the complete HELENA model against its goals in Sec. 10.5. We iteratively uncover design errors and improve the HELENA model of the SCP. Table 10.6 describes the realization of the HELENA model on the technology stack of the SCP. Lastly, we summarize our experiences of applying HELENA to the SCP case study and give an outlook in Sec. 10.9.

The complete specification of the SCP case study in HELENATEXT can be found in Appendix D.1 and on the attached CD in the project `eu.ascens.helenaText.scp`. The generated PROMELA specification for verification together with all goals is shown in Appendix D.2 and on the attached CD in the project `eu.ascens.helenaText.scp` in the folder `promela-gen`. A distributed implementation of the SCP based on the HELENATEXT ensemble specification can be retrieved from `http://svn.pst.ifi.lmu.de/trac/scp`, version v3 of the node core implementation with gossip strategy.

10.1 The Science Cloud Platform

One of the three case studies in the ASCENS project is the *Science Cloud Platform (SCP)* [MKH+13]. The SCP employs a network of distributed, voluntarily provided computing nodes, in which users can deploy and use user-defined software applications. To achieve this functionality, the SCP reuses ideas from three usually separate computing paradigms: cloud computing, voluntary computing, and peer-to-peer computing. In a nutshell, the SCP implements a platform-as-a-service in which individual, voluntarily provided computing nodes interact using a peer-to-peer protocol to deploy, execute, and allow usage of user-defined applications. The SCP takes care to satisfy the requirements of the applications, keeps them running even if nodes leave the system, and provides access to the deployed applications.

To illustrate the idea of the SCP, we imagine a network of peers. Each peer can be of a different type, so for example, one might be a personal desktop computer, another one a laptop, a third one a mobile phone, and a fourth one a powerful server. They all voluntarily contribute to the network of peers by providing their storage and computing resources. This means that each peer can decide whether it wants to join or leave the network at any time. In this voluntary peer-to-peer network, users can deploy software applications like in a platform-as-a-service cloud. A software application thereby does not compute something, but provides a service itself. An example would be a software application like Google Docs[1]. It is deployed in the network and afterwards users can access the deployed software to create documents in the cloud.

As indicated before, the SCP implements its features by using a blend of three usually separate computing paradigms which we shortly discuss in the following: cloud computing, voluntary computing, and peer-to-peer computing. Afterwards, we outline the technical setup of the SCP which serves as the basic infrastructure and describe the process for executing an application on top of the SCP. The description is based on [MKH+13], for more details we refer to the very same paper.

10.1.1 Computing Paradigms

Firstly, as its name implies, the science cloud platform is a *cloud computing* platform. Cloud computing has been a hot topic in computing for some time now and refers to

[1] `https://www.google.com/docs/about/`

the ability of users to use (shared) computing resources (storage, applications, services) with minimum management effort, meaning mostly little required knowledge about how and where the resources are provisioned [MG11]. The SCP uses a specific model of cloud computing, which is the platform-as-a-service model: it provides a platform on which user-defined applications can be executed. Such an application may have requirements – for example, regarding CPU power or memory – and the platform, besides ensuring that the application keeps running in the first place, needs to fulfill these requirements as best as possible.

The second paradigm in use in the SCP is the *voluntary computing* paradigm. Voluntary computing refers to projects in which computing resources are provided voluntarily by end users or organizations instead of being provisioned by a dedicated service provider. The prototypical example of such projects are the @Home projects (especially SETI@Home [KWA+01]). Usually, a central coordinating agency uses voluntarily provided computing power over the Internet to execute small packages of computation, the results of which are then sent back to a central server and integrated. Each contributing user may add and remove his resources at any time. The SCP builds on this paradigm in that it is completely based on voluntarily provided resources (i.e., network nodes). Each node can join or leave the network as its user sees fit, with the platform ensuring that applications are kept running regardless (as far as possible). The nodes can thus be vastly heterogeneous.

Finally, the SCP uses the *peer-to-peer paradigm* [ATS04], which refers to networking infrastructures which perform their tasks without a central coordinator. Good examples are the Internet itself, or the distributed storage of data in hash tables (DHTs). The lack of a central coordinator makes such infrastructures more resilient, but introduces certain overhead for maintenance. The SCP uses the peer-to-peer paradigm for all of its communication, i.e., there is no central coordinator which manages the cloud nodes, data, and applications.

10.1.2 Technical Setup

The SCP is formed by a network of computers which are connected via the Internet, and on which the SCP software is installed (we call these *nodes*). The layout of an SCP node is shown in Fig. 10.1, along with the technologies involved. The dashed boxes are those parts which shall be developed with the HELENA methodology.

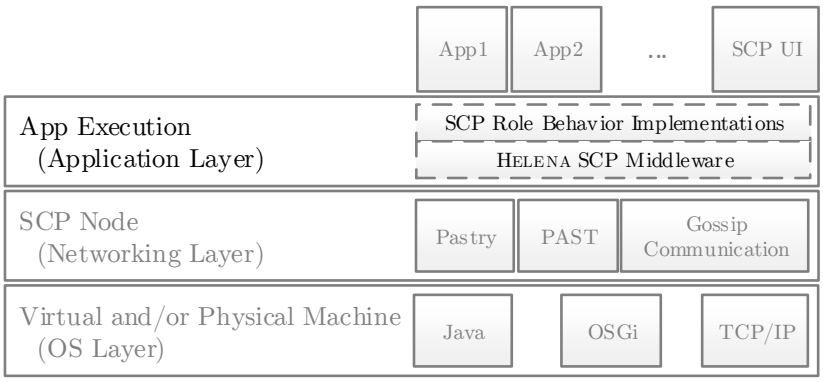


Figure 10.1: SCP architecture (HELENA parts in dashed boxes)

The bottom layer shows the infrastructure: The SCP is a Java application and thus

runs in the Java VM; it also uses the OSGi component framework[2] to dynamically deploy and run applications (as bundles). In general, plain TCP/IP networking is used to communicate between nodes on this level.

The second layer implements the basic networking logic. The SCP uses the distributed peer-to-peer overlay networking substrate Pastry [RD01b] for communication. Pastry works similarly to a Distributed Hash Table (DHT) in that each node is represented by an ID. Node IDs are organized to form a ring along which messages can be routed to a target ID. Pastry manages joining and leaving nodes and contains various optimizations for fast routing. On top of this mechanism, the DHT PAST [RD01a] allows storage of data at specific IDs. On this layer, a gossip protocol [DGH+87] is used to spread information about the nodes through the network; this information includes node abilities (CPU, RAM), but also information about applications. Each node slowly builds its own picture of the network, pruning information where it becomes outdated.

The third layer shall implement the application execution logic based on HELENA. The required functionality of the application layer is that of reliable application execution given the application requirements on the one hand and the instability of the network on the other hand. This process is envisioned as follows:

(1) **Deploying the App:** A user deploys an application (like Google Docs) using the SCP UI (top right in Fig. 10.1). The application is assigned an ID (based on its name) and stored using the DHT (PAST) at the closest node according to the ID; this ensures that exactly one node is responsible for the application, and this node can always be retrieved based on the application name (we call this node the *app-responsible node*). If this node leaves, the next adjacent node based on ID proximity takes its place.

(2) **Finding an Executor:** Since each application comes with execution requirements (e.g., a certain amount of memory) and all nodes are heterogeneous, the app-responsible node may or may not be able to execute the application. Thus, it is tasked with *finding* an appropriate executor (based on the gossiped information).

(3) **Keeping the App Alive:** Once an executor is found, it is asked to retrieve and run the application. Through a continuous exchange of keep-alive messages, the app-responsible node observes the executor and is thus able to select a new one if it fails.

(4) **Providing Access to the App:** Finally, the user may interact with the application through the SCP UI. It may request a service from the application (like writing documents with Google Docs) and receive a corresponding answer.

This third layer shall be based on the HELENA concepts. Thus, a HELENA SCP middleware is needed which allows to implement roles and their collaboration in ensembles on top of the distributed set-up of the SCP. The HELENA SCP middleware reuses the idea of the jHELENA framework from Chap. 7 and connects the HELENA concepts of roles and ensembles with the underlying SCP technology stack. The application execution logic is then realized by role behavior implementations which use the HELENA SCP middleware to employ ensembles on top of the SCP and to allow communication and collaboration between roles in the style of HELENA.

[2] https://www.osgi.org/

In the remainder of this chapter, we will be concerned with the modeling and the implementation of the application execution layer. For this layer, including the logic for deploying an app, finding an executor, and serving user requests, we will derive formal goals, model the behavior using HELENA, check satisfaction of goals in the HELENA specification, and subsequently implement the HELENA model on the technology stack of the actual SCP.

10.2 Domain Model

Following the HELENA methodology, the first step in the development of the case study is to capture the domain of the SCP, i.e., its contributing entities. We assume given the basic infrastructure for communication between nodes (Pastry), storing data (PAST), and deploying and executing applications (OSGi) (the two bottom layers in Fig. 10.1). On top of this infrastructure, we apply HELENA for modeling the whole process of application execution. Therefore, we first identify the computation nodes and their properties which serve as the individual peers contributing their storage and computing resources to the SCP. Building on these computation nodes, we derive the required role types from the informally stated requirements in Sec. 10.1.

Assumption: For simplicity, we only consider the deployment of a single application in the network and we assume all peers in the network homogeneous, but we indicate how the model could be enhanced to cope with several applications and a heterogeneous network. Furthermore, we do not model failure or coordinated leaving of nodes storing or executing the application.

(1) **Deploying the Application:** For this subtask, we envision two separate role types. The `Deployer` provides the interface for deploying an application and takes care to select the app-responsible node. The app-responsible node adopts the `Storage` role taking care for the actual storage.

(2) **Finding an Executor:** Two further roles are required for finding the appropriate execution node. The app-responsible node in the role `Initiator` determines the actual `Executor`. It selects a node as `Executor` which is able to satisfy the execution requirements of the application. The `Executor` itself takes care to keep the application running and provides access to it.

(3) **Keeping the Application Alive:** As stated before, we do not model failure or coordinated leaving of nodes. Therefore, we do not introduce any roles to observe and manage the execution of the application.

(4) **Providing Access to the Application:** Once started, the application needs to be available for user requests. The `Requester` provides the interface between the human user and the `Executor` and forwards requests and responses. The `Executor` from the previous subtask gives access to the executed application.

Fig. 10.2 summarizes the domain model consisting of the underlying component-based platform and an appropriate ensemble structure which just describes the basic entities of the problem domain without any capabilities like operations and messages so far. The individual peers contributing to the SCP (and therefore the underlying component-based platform) are represented by components of the type `Node` since we assume a homogeneous set of peers. Each `Node` has a unique ID given by the attribute

`id` identifying it in the network. Each `Node` can satisfy particular requirements for executing applications which we represent by an integer attribute `reqs`. It furthermore can store the byte code of an application. We represent the byte code by an attribute `code` of type `int` (and not of type `byte`) to be able to later on check the model with Spin. To alleviate the assumption that only one application can be stored, this attribute would have to be enhanced to an array to store the byte codes of several applications. Finally, the attribute `isExecuting` denotes whether the `Node` is currently executing the application or not. Once again, the assumption that only one application can be executed is alleviated by enhancing this attribute to an array. Note that application execution is independent from storing the application's code. Thus, the attributes `code` and `isExecuting` can independently be set. Furthermore, the attribute `isExecuting` might seem superfluous at first since we will later on introduce the role type `Executor` to express the task of executing the application. However, the attribute is important because once a role instance of type `Executor` is active, it is not guaranteed that it currently executes the application. For example, the executor might still have to retrieve the code of application before it actually executes the application.

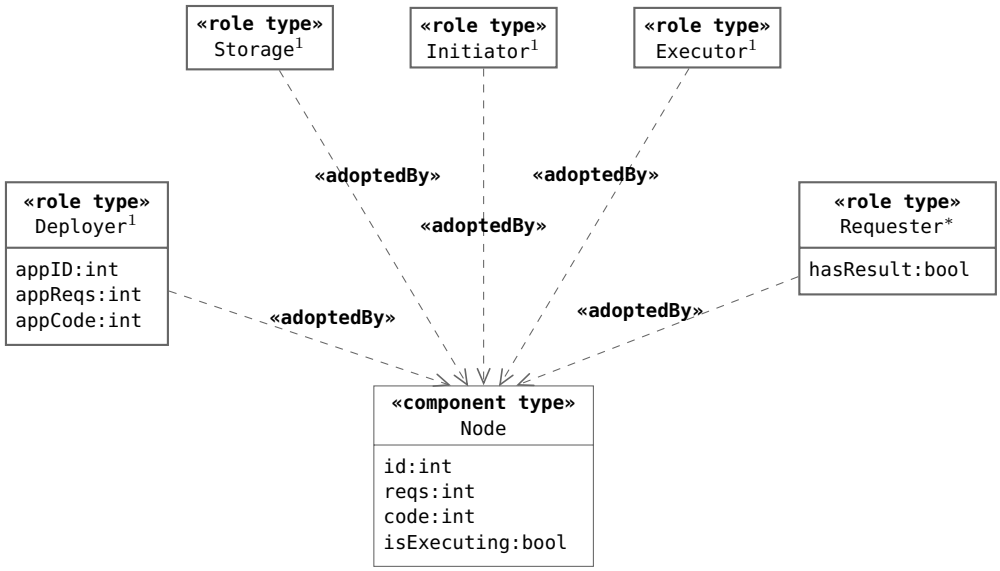


Figure 10.2: Domain model for the SCP case study

If we wanted to include heterogeneous peers in the network, we would add several component types, each modeling a different type of device contributing to the network (e.g., desktop computers, laptops, mobile phones, servers). They could differ in the properties they offer. For example, mobile phones might not be able to store applications at all, but might be able to execute applications requiring a GPS sensor. Servers in contrast might be able to store applications and to execute applications with large memory requirements.

Furthermore, the domain ensemble structure is composed of five role types as explained before. Each role can be adopted by components of type `Node`. The minimal and maximal number of instances per role type is denoted as a multiplicity and expresses that a running ensemble contains just one instance per role type except for the role type `Requester`. There can be as many `Requester`s as there are human users requesting a service from the application. Since the given ensemble structure only captures the solution domain of the SCP, but not yet the actual realization, all role types are not yet equipped with messages. Messages are added to the role types when designing the

goal-directed behavior of each roles. When specifying the domain, the role types only have attributes which describe their properties. The role type `Deployer` can store the ID (attribute `id` of type `int`), the requirements (attribute `reqs` of type `int`) and the byte code (attribute `code` of type `int`) of the application to be deployed. The role type `Storage` does not store the application's byte code itself, but rather the underlying component `Node` such that the code is permanently available even if the role `Storage` finishes its execution. Similarly, the role type `Executor` does not execute the application itself, but rather the underlying component `Node`. Lastly, the role type `Requester` denotes with the attribute `hasResult` whether it received a response from the `Executor` upon its request to the executed app.

Already when modeling the domain of the SCP, the benefits of using the HELENA concept of roles are apparent. In the SCP, distributed computing nodes interact to execute software applications. For one application, several computing nodes need to collaborate: They have to let a user deploy the application in the system, to execute the application on a node satisfying the computation requirements of the application, and to let a user request a service from the application. At design time, it is unclear which node will be assigned with which responsibility (additionally in principle, each node must also be able to take over the same or different responsibilities for the execution of different applications in parallel.) In a standard component-based design, we would have to come up with a single component type for a computing node which is able to combine the functionalities for each responsibility in one complex behavior. This is the case in the "all-in-one" implementation of the SCP presented in [MKH+13]. HELENA, however, offers the possibility to model systems in terms of collaborating roles and ensembles. Therefore, each responsibility can be encapsulated in a dedicated role while the underlying component stores all information which should be permanently available. When specifying goal-directed behaviors for each role, more benefits arise like that adopting different roles will allow components to change their behavior on demand.

10.3 Goal Specification

In Sec. 10.1, the case study and the goals of the envisioned ensemble were informally described. To derive formal goal specifications from these informal descriptions, we exploit the domain model from the previous section. It lists the main entities contributing to the ensemble-based system such that we can define the formal goals relative to these entities. Table 10.1 summarizes all goals with their type, achieve goal or maintain goal, and their formalization in LTL according to the domain model from Fig. 10.2.

The informal description of the case study names four main tasks: deploying the app, finding an executor, keeping the app alive and providing access to the app. To formally express these tasks, we derive five goals as shown in Table 10.1. The first two goals describe the task of deploying the app, the third goal the task of finding an executor, the fourth goal the task of keeping the app alive and the last goal the task of providing access to the app.

(1) **Deploying the App:** The goal "code gets stored" is an achieve goal. It requires that the byte code is eventually stored on the node with the ID closest to the app's ID. Exploiting the domain model, this is expressed by referring to the attributes *id* and *code* of a node and the attributes *appID* and *appCode* of the deployer. $Node[1]{:}code > 0$ expresses that the node with identifier 1 stores some code in its attribute *code*. Furthermore, $Node[1]{:}code == Deployer{:}appCode$

states that the node with identifier 1 stores the same content in its attribute *code* as the only deployer in the ensemble stores in its attribute *appCode*. For that first goal, it is important that the requirement $Node[1]{:}code > 0$ is added since at the beginning, the deployer might not store any code such that the goal without the requirement $Node[1]{:}code > 0$ might already be fulfilled in the initial state of the ensemble although the code of the application is not yet stored. Note thereby that the identifier 1 is not the identifier of the node in the network (i.e., the identifier stored in the attribute *id*), but the identifier of the node in the ensemble state. Furthermore, since LTL does not offer quantifiers, we cannot express that there should exists a node which stores the app's code, but we rather have to enumerate all nodes in the network. In the formalization in Table 10.1, we assume that the network only consists of two nodes, but the formula is easily extended if the network contains more nodes. Similarly, $|Node[1]{:}id{-}Deployer{:}appID| < |Node[2]{:}id{-}Deployer{:}appID|$ means that the node with identifier 1 has an ID stored in the attribute *id* which is closer to the app's ID expressed by the attribute *appID* of the deployer than the node with identifier 2 (we use $|x|$ to denote the absolute value of a number x). Note here again that the identifiers written in the square brackets are the identifiers of the node in the ensemble state, the values stored in the attribute *id* are the IDs in the network. The goal "code stays stored" is a maintain goal. It requires that whenever the byte code got stored on a node in the network, it remains in the network (not necessarily stored on the same node). As before, we refer to the attribute *code* of a node and to the attribute *appCode* of the deployer and have to enumerate all nodes in the network. The informal goal is formalized by stating that whenever (expressed by the first $\Box$ and the implication) one of the nodes in the network stores the application's byte code, then the byte code always (expressed by the second $\Box$) remains stored in the network. For this goal, we do not specify that the node must be closest to the application's ID since this is already guaranteed by the previous goal "code gets stored".

(2) **Finding an Executor:** The goal "app gets executed" is an achieve goal. It requires that if there exists at least one node in the network which satisfies the execution requirements of the application, then the application gets eventually executed on such a node. Exploiting the domain model, this is expressed by referring to the attributes *reqs* and *isExecuting* of a node and to the attribute *appReqs* of the deployer. For example, $Node[1]{:}reqs{==}Deployer{:}appReqs$ means that the node with identifier 1 provides the same requirements (expressed by the attribute *reqs* of the node) as the application requires (expressed by the attribute *appReqs* of the deployer). Likewise, $Node[1]{:}isExecuting$ means that the node with identifier 1 executes the application. One might be tempted to express the execution of the application by reasoning about whether a node currently plays the role of an executor. However, this plays query is not enough since though active the executor might not currently execute the application, e.g., because it has to retrieve the code of the application before. Nevertheless, even if not applicable here, this observation gives rise to extend the atomic propositions of HELENA LTL by plays queries.

Table 10.1: Goals for the SCP case study with two nodes in HELENA LTL

Name	LTL Formula
code gets stored (achieve goal)	$\Diamond($ $(Node[1]{:}code > 0 \wedge$ $Node[1]{:}code{==}Deployer{:}appCode \wedge$ $\lvert Node[1]{:}id{-}Deployer{:}appID\rvert < \lvert Node[2]{:}id{-}Deployer{:}appID\rvert)$ $\vee$ $(Node[2]{:}code > 0 \wedge$ $Node[2]{:}code{==}Deployer{:}appCode \wedge$ $\lvert Node[2]{:}id{-}Deployer{:}appID\rvert < \lvert Node[1]{:}id{-}Deployer{:}appID\rvert)$ $)$
code stays stored (maintain goal)	$\Box($ $($ $(Node[1]{:}code > 0 \wedge$ $Node[1]{:}code{==}Deployer{:}appCode)$ $\vee$ $(Node[2]{:}code > 0 \wedge$ $Node[2]{:}code{==}Deployer{:}appCode)$ $)$ $\Rightarrow$ $\Box($ $(Node[1]{:}code > 0 \wedge$ $Node[1]{:}code{==}Deployer{:}appCode)$ $\vee$ $(Node[2]{:}code > 0 \wedge$ $Node[2]{:}code{==}Deployer{:}appCode)$ $)$ $)$
app gets executed (achieve goal)	$(Node[1]{:}reqs{==}Deployer{:}appReqs$ $\vee$ $Node[2]{:}reqs{==}Deployer{:}appReqs)$ $\Rightarrow$ $\Diamond($ $(Node[1]{:}reqs{==}Deployer{:}appReqs \wedge Node[1]{:}isExecuting)$ $\vee$ $(Node[2]{:}reqs{==}Deployer{:}appReqs \wedge Node[2]{:}isExecuting)$ $)$
app stays executed (maintain goal)	$\Box($ $(Node[1]{:}isExecuting \vee Node[2]{:}isExecuting)$ $\Rightarrow$ $\Box(Node[1]{:}isExecuting \vee Node[2]{:}isExecuting)$ $)$
requester i gets served (achieve goal)	$\Diamond Requester[i]{:}hasResult$

(3) **Keeping the Application Alive:** Even though we do not model failure of nodes, we have to make sure that our model keeps the application alive without any node failure. Therefore, the goal "app stays executed" is a maintain goal which requires that whenever the application got executed, it remains executed. To express application execution, we refer to the attribute *isExecuting* of a node again. The informal goal is formalized by stating that whenever (expressed by the first $\Box$ and the implication) one of the nodes in the network executes the application, then the application always (expressed by the second $\Box$) remains executed in the network. For this goal, we do not specify that the node must satisfy the application's requirements since this is already guaranteed by the previous goal "app gets executed".

(4) **Providing Access to the Application:** The goal "requester i gets served" is an achieve goal. It requires that the requester with identifier i gets eventually a result after its request to the application. Exploiting the domain model, this is expressed by referring to the attribute *hasResult* of the requester. Since there can be many requesters in the network, this goal is actually a generic goal and has to be unfolded for the actual number of requesters in the network. That means, if we employ two requesters with the identifiers 4 and 5, we have to satisfy two separate goals, one naming the requester with identifier 4 and one naming the requester with identifier 5.

10.4 Design

After having formalized the goals for the SCP case study, we design the goal-directed behavior of all participating entities of the HELENA model. We use the domain model from Sec. 10.2 as a basis. All its roles are equipped with messages and a goal-directed role behavior to collaborate on the four tasks of the SCP. Furthermore, the domain model is enhanced by further roles which were not relevant to represent the domain of the SCP, but are necessary to allow appropriate collaboration in the ensemble.

Following the HELENA development methodology, we first design the interaction behavior of the complete ensemble and therefore of the collaborating roles as a whole. We visualize it in a notation similar to sequence diagrams as depicted in Fig. 10.3. The diagram shows how the different roles of the SCP interact to first store the application, then to execute the application, and finally to provide access to the application and request a service from it (we assume a single requester in the sequence diagram). By projecting the overall interaction behavior on each single role, we then gain all separate role behaviors.

In the following, we do not walk through the whole sequence diagram, but rather use it as an accompanying visualization. We individually focus on the four tasks of deploying the application, finding an executor, keeping the application alive, and providing access to the application and describe the roles and their behaviors for each of the four tasks in the following subsections.

The complete ensemble specification has 314 lines of code and is given in Appendix D.1 (the ensemble specification in the appendix is slightly extended to allow model-checking with Spin, extensions are described in Sec. 10.5.2). It can also be found on the attached CD in the project `eu.ascens.helenaText.scp` in the file `src/scp.helena`.

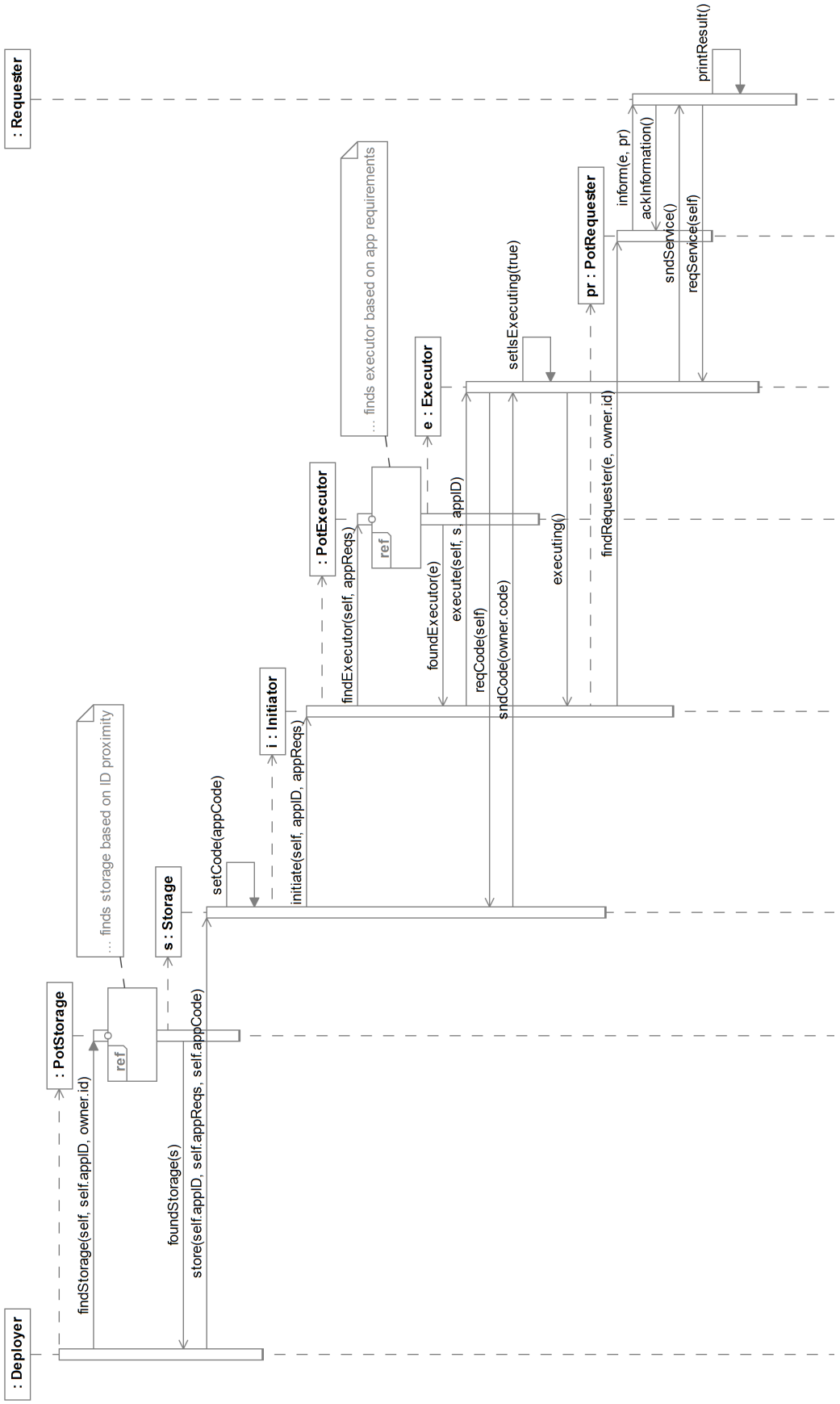


Figure 10.3: Interactions between roles in the SCP case study

10.4.1 Deploying the Application

Fig. 10.4 depicts all role types together with their attributes and messages which are necessary to allow deployment of a software application in the SCP (attributes, messages, and operations necessary for other tasks are grayed out). As already described in the domain model, an instance of the role type `Deployer` takes care to deploy the application in the network and an instance of the role type `Storage` stores the application's byte code. However, we need a third role type `PotStorage` whose instances are responsible for finding the node whose ID is closest to the ID of the application.

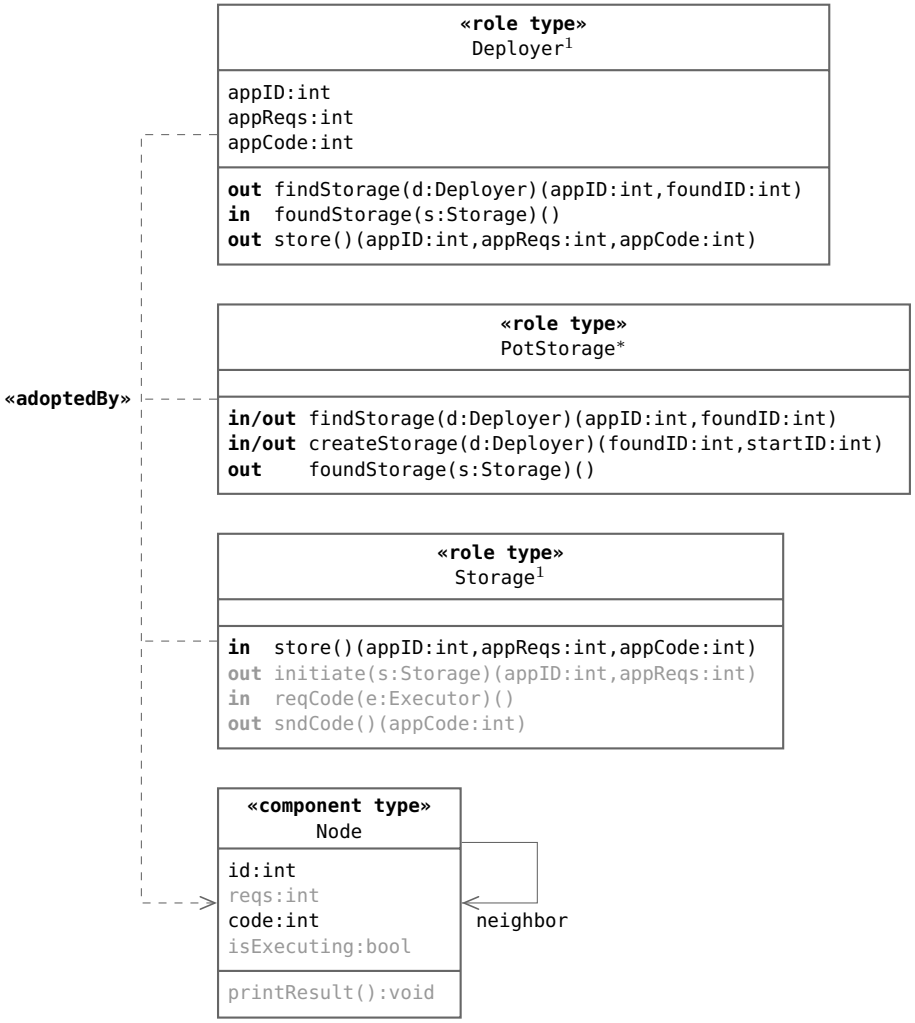


Figure 10.4: Ensemble structure for the SCP case study, part application deployment (gray attributes, messages, and operations are not necessary for this task)

Ensemble Structure: As already described in Sec. 10.2, the deployer stores the ID, the requirements and the byte code of the application to be deployed in its attributes. It is able to send and receive three types of messages: By sending the message `findStorage`, it triggers the search for an appropriate storage in the network, i.e., the node with the ID closest to the application's ID. With the message `foundStorage`, it receives the reference to the found storage. By sending the message `store`, it issues the actual storage of the application in the network. The task of the potential storages is to determine the appropriate node to store the application. To this end, a potential storage can receive

the message `findStorage` indicating that a storage for a specific application should be found. By sending the same message, it forwards the request for a storage through the network. The second message `createStorage` indicates that an appropriate node for storing the application was found and the storage role can now be created on that node. By sending the message `foundStorage`, a potential storage can inform other roles that an appropriate node was found and a new instance for the storage was created. Lastly, the storage itself is able to receive the message `store`. With this message, it receives the application's byte code and can then store it on its owning node. All other messages of the storage are not relevant for the task of application deployment.

Role Behavior of the `Deployer`: Fig. 10.5 shows the role behavior of the deployer in HelenaText. The first three lines (line 2–4) initialize the role attributes to store the application's ID, computation requirements, and byte code. In the actual realization of the SCP, these values are not fixed, but are rather given by the user who wants to deploy the app. However, so far, the Helena methodology does not allow open ensembles and therefore user input is not possible. After initialization of its attributes, the deployer creates a potential storage on its neighbor (line 5) and issues the search for an appropriate storage by sending the message `findStorage` to the newly created potential storage (line 6). The parameters of this request are essential: the first parameter denotes the ID of the application, the second parameter denotes the ID of the node which is so far closest to the application's ID (which is currently the ID of the owning node of the deployer). During searching the appropriate storage, the potential storages will try to minimize the distance between these two parameters. After starting the search for an appropriate storage on the potential storage, the deployer waits for the message `foundStorage` (line 7) which gives it a reference to the created storage in the parameter `s`. It finally instructs the storage `s` to store the byte code of the application with the message `store` (line 8) and quits its execution (line 9).

```
1  roleBehavior Deployer =
2    self.appID = 1234 .
3    self.appReqs = 1234 .
4    self.appCode = 1234 .
5    ps <- create(PotStorage,owner.neighbor) .
6    ps ! findStorage(self)(self.appID,owner.id) .
7    ? foundStorage(Storage s)() .
8    s ! store()(self.appID,self.appReqs,self.appCode) .
9    quit
```

Figure 10.5: Role behavior of a `Deployer` in the SCP case study in HelenaText

Role Behavior of a `PotStorage`: Fig. 10.6 shows the role behavior of a potential storage in HelenaText. It is the most complex one, thus, we first informally describe the idea of finding the appropriate storage. We assume that all nodes in the network are arranged in a ring structure, i.e., each node is connected to its neighbor (as indicated by the association `neighbor` in Fig. 10.4) and the last node in the chain is connected to the first one again. The potential storages traverse this ring structure two times: During the first round trip, they determine the node whose ID is closest to the application's ID by the message `findStorage`. During the second round trip, they inform each other with the message `createStorage` which node is the appropriate storage, the potential

storage on this node creates the actual storage and informs the deployer about the creation with the message `foundStorage`.

```
1  roleBehavior PotStorage = PotStorageProcess {
2    process PotStorageProcess =
3      ? findStorage(Deployer depl)(int appID,int foundID) .
4      if ( (owner.id-appID < 0 && foundID-appID < 0 && appID-owner.id < appID-foundID)
5        ||
6         (owner.id-appID >= 0 && foundID-appID >= 0 && owner.id-appID < foundID-appID)
7        ||
8         (owner.id-appID < 0 && foundID-appID >= 0 && appID-owner.id < foundID-appID)
9        ||
10        (owner.id-appID >= 0 && foundID-appID < 0 && owner.id-appID < appID-foundID)
11     ) {
12       if (!plays(PotStorage,owner.neighbor)) {
13         psSmallest1 <- create(PotStorage,owner.neighbor) .
14         psSmallest1 ! findStorage(depl)(appID,owner.id) .
15         SecondRoundTrip
16       }
17       else {
18         psSmallest2 <- get(PotStorage,owner.neighbor) .
19         psSmallest2 ! createStorage(depl)(owner.id,owner.id) .
20         SecondRoundTrip
21       }
22     }
23     else {
24       if (!plays(PotStorage,owner.neighbor)) {
25         psNotSmallest1 <- create(PotStorage,owner.neighbor) .
26         psNotSmallest1 ! findStorage(depl)(appID,foundID) .
27         SecondRoundTrip
28       }
29       else {
30         psNotSmallest2 <- get(PotStorage,owner.neighbor) .
31         psNotSmallest2 ! createStorage(depl)(foundID,owner.id) .
32         SecondRoundTrip
33       }
34     }
35   process SecondRoundTrip =
36     ? createStorage(Deployer depl2)(int foundID2,int startID) .
37     if (owner.id == foundID2) {
38       s <- create(Storage,owner) .
39       depl2 ! foundStorage(s)() .
40       Fwd
41     }
42     else {
43       Fwd
44     }
45   process Fwd =
46     if (owner.id != startID) {
47       psFwd <- get(PotStorage,owner.neighbor) .
48       psFwd ! createStorage(depl2)(foundID2,startID) .
49       quit
50     }
51     else {
52       quit
53     }
54 }
```

Figure 10.6: Role behavior of a `PotStorage` in the SCP case study in HelenaText

In more detail, each potential storage executes the behavior given in Fig. 10.6. It first receives the message `findStorage` (line 3). With the second parameter `foundID`, the current potential storage receives the ID of the node which is so far closest to the application's ID. It then decides whether the distance between the ID of its owning node

and the application's ID is smaller than the distance between the received ID and the application's ID (thereby, we assume the node IDs are greater than 0). The **if**-branch of the if-then-else construct (line 4–22) describes the behavior when the distance is smaller, the **else**-branch (line 23–34) when the distance is larger or equal (note that in HELENA we do not offer $|x|$ as the absolute value of x). If the distance is smaller (line 4–11), the potential storage forwards the ID `owner.id` of its owning node to the potential storage on its neighbor (line 12–21); if not (line 23), it forwards the received ID `foundID` (line 24-33). Before forwarding, the current potential storage has to decide whether the first round trip through the network was already finished (it does so by checking whether its neighbor already plays the role of a potential storage (line 12 and 24)). If the first round trip is not yet finished, the current potential storage uses the message `findStorage` for forwarding (line 14 and 26); if it was finished, it uses the message `createStorage` (line 19 and 31). In both cases, the potential storage continues its behavior with the process `SecondRoundTrip` where it waits for the message `createStorage` indicating that the second round trip is on-going. During the second round trip, the current potential storage determines whether its owning node was the closest to the application based on ID proximity. If so, the parameter `foundID2` of the received message `createStorage` is the same as the ID of its owning node. Therefore, it compares the parameter `foundID2` with the ID of its owning node `owner.id`. If they are the same (line 37), i.e., the owning node is the closest, the potential storage creates a new role instance of type `Storage` on its owning node (line 38) and informs the deployer about the creation of the storage (line 39). Afterwards, the current potential storage forwards the creation message `createStorage` to its neighboring potential storage (line 40) if the second round trip has not already been finished (line 46). This forwarding is necessary to allow all potential storages to stop executing their behaviors even if they are not owned by the node with the ID closest to the application's ID. If the owning node of the current potential storage is not the closest node to the application (line 42), the current potential storage just forwards the creation message (line 43). In this case, forwarding is even more essential since the potential storage which is owned by the node with the closest ID might reside after the current potential storage in the node ring structure.

Role Behavior of the `Storage`: Fig. 10.7 shows the role behavior of the storage in HELENATEXT. Only the first two lines are relevant for deploying the application. In line 2, the storage receives the message `store` indicating that it should store the application. With the parameter `appCode` of this message, it receives the application's byte code. Afterwards, it stores the byte code in the attribute `code` of its owning node. The next actions in the role behavior do not contribute to the deployment, but trigger the execution of the application which is described in the next subsection.

```
1  roleBehavior Storage =
2    ? store()(int appID, int appReqs, int appCode) .
3    owner.code = appCode .
4
5    i <- create(Initiator,owner) .
6    i ! initiate(self)(appID,appReqs) .
7
8    ? reqCode(Executor e)() .
9    e ! sndCode()(owner.code) .
10   quit
```

Figure 10.7: Role behavior of a `Storage` in the SCP case study in HELENATEXT

10.4.2 Finding an Executor

Fig. 10.8 depicts all role types with their attributes and messages necessary to trigger the execution of an application in the SCP (attributes, messages, and operations necessary for other tasks are grayed out). Finding an executor is issued by the `Storage` which triggers the `Initiator`. As already described in the domain model, the `Initiator` takes care to determine the actual executor based on the computation requirements of the application. The `Executor` retrieves the application's code from the `Storage`, executes the application, and provides access to it. However, we need a fourth role type `PotExecutor` whose instances are responsible for finding a node in the network satisfying the requirements of the application and is thus able to execute the application.

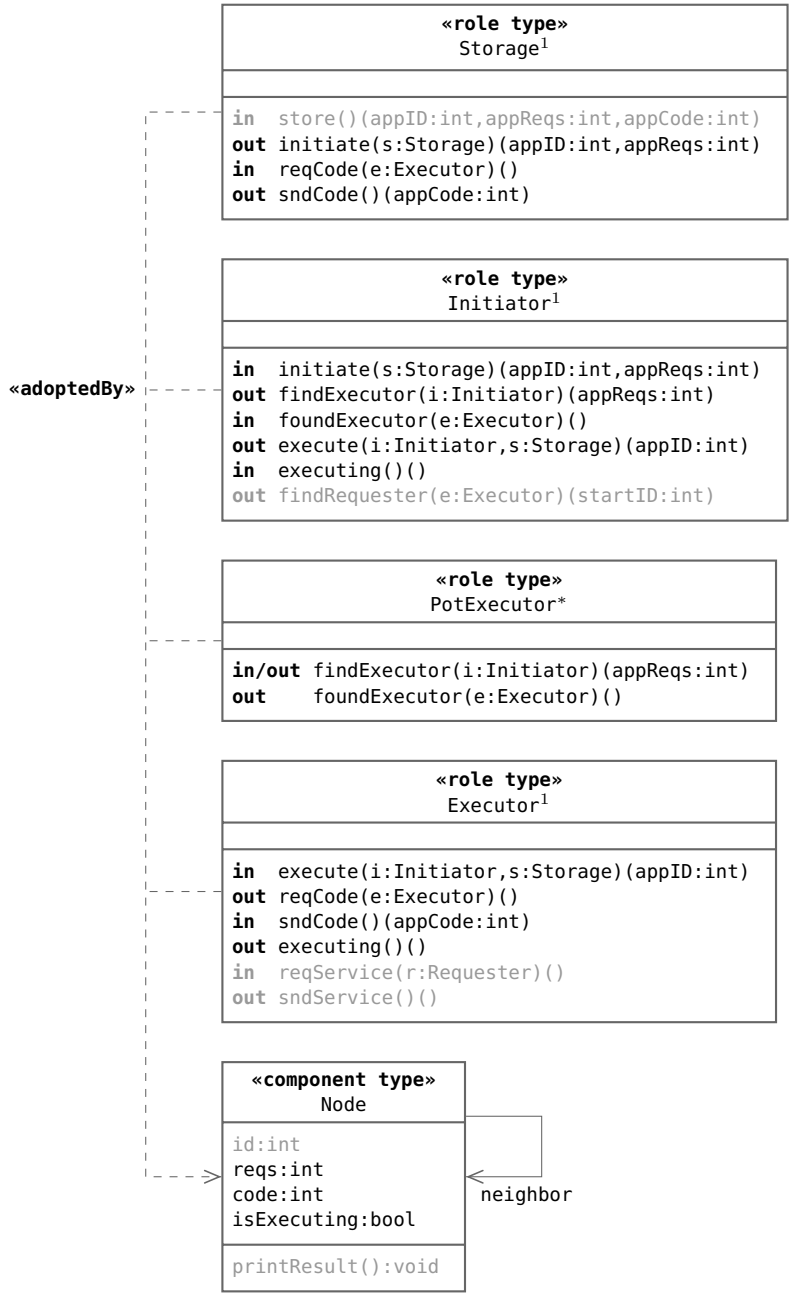


Figure 10.8: Ensemble structure for the SCP case study, part for finding an executor (gray attributes, messages, and operations are not necessary for this task)

Ensemble Structure: The storage triggers the task of finding an executor by the message `initiate`. It furthermore can receive requests for the application's byte code by the message `reqCode` and can respond with the message `sndCode`. The initiator's execution is triggered by receiving the message `initiate`. By sending the message `findExecutor`, it triggers the search for an appropriate executor in the network, i.e., a node which satisfies the application's computation requirements. With the message `foundExecutor`, it receives the reference to the found executor. By sending the message `execute`, it issues the actual execution of the application in the network and receives the acknowledgment of the execution by the message `executing`. The task of the potential executors is to determine the appropriate node to execute the application. To this end, a potential executor can receive the message `findExecutor` indicating that an executor for a specific application should be found. By sending the same message, it forwards the request for an executor through the network. By sending the message `foundExecutor`, a potential executor can inform other roles that an appropriate node was finally found and a new role instance for the executor was created. Lastly, the executor itself is able to receive the message `execute`. It furthermore requests the byte code of the application to be executed with the message `reqCode` and receives the byte code with the message `sndCode`. With the message `executing`, it informs other roles that it finally executes the application. All other messages of the (storage,) initiator and executor are not relevant for the task of executing the application.

Role Behavior of the `Storage` (continued): Fig. 10.7 shows the role behavior of the storage in HelenaText from before. After having stored the application's byte code as required in the previous task, the storage now triggers the search for an appropriate executor by first creating an initiator on its owning node (line 5) and then sending the message `initiate` to it (line 6). The storage is not part of the actual search process, but if an executor was found, it provides the byte code to the executor. Therefore, the storage waits with the message `reqCode` for a request of the byte code (line 8) and sends it back via the message `sndCode` (line 9). In our design, the storage then quits its execution since we assume only one executor in the SCP ensemble. However, if we assumed several executors, the storage should be continuously able to answer requests for the application's byte code via recursive process invocation.

Role Behavior of the `Initiator`: Fig. 10.9 shows the role behavior of the initiator in HelenaText. In line 2, the initiator receives the message `initiate` indicating that it should initiate the search for the executor. Most importantly, it thereby receives the computation requirements of the application with the parameter `appReqs`. It then initiates the search by creating a potential executor on its owning node (line 4) and sending the message `findExecutor` containing the application's requirements to the newly created potential executor. After having received the reference to the found executor by the message `foundExecutor`, it triggers execution on that executor by sending the message `execute` and waits for the acknowledgment of the execution with the message `executing`. The next actions in the role behavior do not contribute to the task of deploying the application, but to giving access to the application.

Role Behavior of a `PotExecutor`: Fig. 10.10 shows the role behavior of a potential executor in HelenaText. The idea of finding an appropriate executor resembles the idea of finding an appropriate storage. We again assume that all nodes in the network are arranged in a ring structure. The potential executors traverse this ring structure

```
1  roleBehavior Initiator =
2    ? initiate(Storage s)(int appID,int appReqs) .
3
4    pe <- create(PotExecutor,owner) .
5    pe ! findExecutor(self)(appReqs) .
6    ? foundExecutor(Executor e)() .
7    e ! execute(self,s)(appID) .
8    ? executing()() .
9
10   pr <- create(PotRequester,owner.neighbor) .
11   pr ! findRequester(e)(owner.id) .
12   quit
```

Figure 10.9: Role behavior of a `Initiator` in the SCP case study in HelenaText

by the message `findExecutor` until an appropriate executor is found, i.e., a node which satisfies the computation requirements of the application. If such a node is found, the current potential executor creates the actual executor on this node and informs the initiator about the creation with the message `foundExecutor`.

In more detail, each potential executor executes the behavior given in Fig. 10.10. It first receives the message `findExecutor` (line 3). With the parameter `appReqs` of this message, the current potential executor receives the computation requirements of the application. It then decides whether its owning node satisfies these requirements. The **if**-branch of the guarded choice construct (line 3–7) describes the behavior if the owner satisfies the requirements, the **or**-branch (line 8–12) if not. If the owner satisfies the requirements (line 3), the current potential executor creates a new role instance of type `Executor` on its owning node (line 4), informs the initiator about the creation of this executor (line 5), and quits its behavior without any forwarding of the search request since the executor has already been found. If the owner does not satisfy the requirements (line 8), the potential executor forwards the request to find an appropriate executor to its neighboring potential executor (line 9–10) and quits its execution.

```
1  roleBehavior PotExecutor =
2    ? findExecutor(Initiator i)(int appReqs) .
3    if (owner.reqs == appReqs) {
4      e <- create(Executor,owner) .
5      i ! foundExecutor(e)() .
6      quit
7    }
8    else {
9      pe <- create(PotExecutor,owner.neighbor) .
10     pe ! findExecutor(i)(appReqs) .
11     quit
12   }
```

Figure 10.10: Role behavior of a `PotExecutor` in the SCP case study in HelenaText

Role Behavior of the `Executor`: Fig. 10.11 shows the role behavior of the executor in HelenaText. In line 3, the executor receives the message `execute` indicating that it should execute the application. To be able to execute the application, it first has to retrieve the application's byte code from the storage. Therefore, it requests the code from the storage by sending the message `reqCode` (line 5) and receives it via the message `sndCode` (line 6). Afterwards, we simulate the execution of the application by setting

the attribute `isExecuting` of the owning node to **true** (line 7). Finally, the executor acknowledges the execution to the initiator by sending the message `executing` (line 8). The next actions in the role behavior do not contribute to the task of executing the application, but to providing access to the application.

```
1  roleBehavior Executor = ExecutorProcess {
2    process ExecutorProcess =
3      ? execute(Initiator i, Storage s)(int appID) .
4
5      s ! reqCode(self)() .
6      ? sndCode()(int appCode) .
7      owner.isExecuting = true .
8      i ! executing()() .
9
10     ExecutorRunning
11
12   process ExecutorRunning =
13     ? reqService(Requester r)() .
14     r ! sndService()() .
15     ExecutorRunning
16 }
```

Figure 10.11: Role behavior of a `Executor` in the SCP case study in HelenaText

10.4.3 Keeping the Application Alive and Providing Access

Fig. 10.12 depicts all role types together with their attributes and messages which are necessary to keep the application alive and provide access to it (attributes, messages, and operations which are necessary for other tasks are grayed out). For keeping alive, we do not envision any further role types since we assumed that nodes do not fail or leave the network. To provide access to the application, the `Initiator` as the app-responsible node first informs all `Requester`s that the application is now available for requests. However, we need another role type `PotRequester` whose instances are responsible for finding all nodes in the network which want to request a service from the executed application, i.e., all currently existing `Requester`s, and inform them about the availability of the application. Afterwards, (possibly many) `Requester`s request a service from the executed application by accessing the `Executor`.

Ensemble Structure: For the task of keeping the application alive, no role types are needed since we do not failure of coordinated leaving of nodes. The task of providing access to the application is triggered by the initiator with the message `findRequester`. It initiates that all currently existing requesters in the network are searched and informed that the application is now available in the network. The task of the potential requesters is to inform all currently existing requesters about the availability of the application. To this end, a potential requester can receive the message `findRequester` indicating that all requesters should be informed. By sending the same message, it forwards the information through the network. By sending the message `inform`, a potential requester informs a requester about the availability of the application and by receiving the message `ackInformation`, it receives the acknowledgment that the requester actually received the information. A requester itself is informed about the availability of the application by receiving the message `inform`. It acknowledges the information by sending the message `ackInformation`. It then can request a service from the application (e.g., to edit a

Google Docs document) by sending the message `reqService`. The answer is simulated by receiving the message `sndService`. As described in the domain model, the attribute `hasResult` indicates whether the requester already received an answer or not. Lastly, the executor can be asked to give access to the application by receiving the message `reqService`. It sends back the answer by sending the message `sndService`. All other messages of the initiator and executor are not relevant for the task of executing the application.

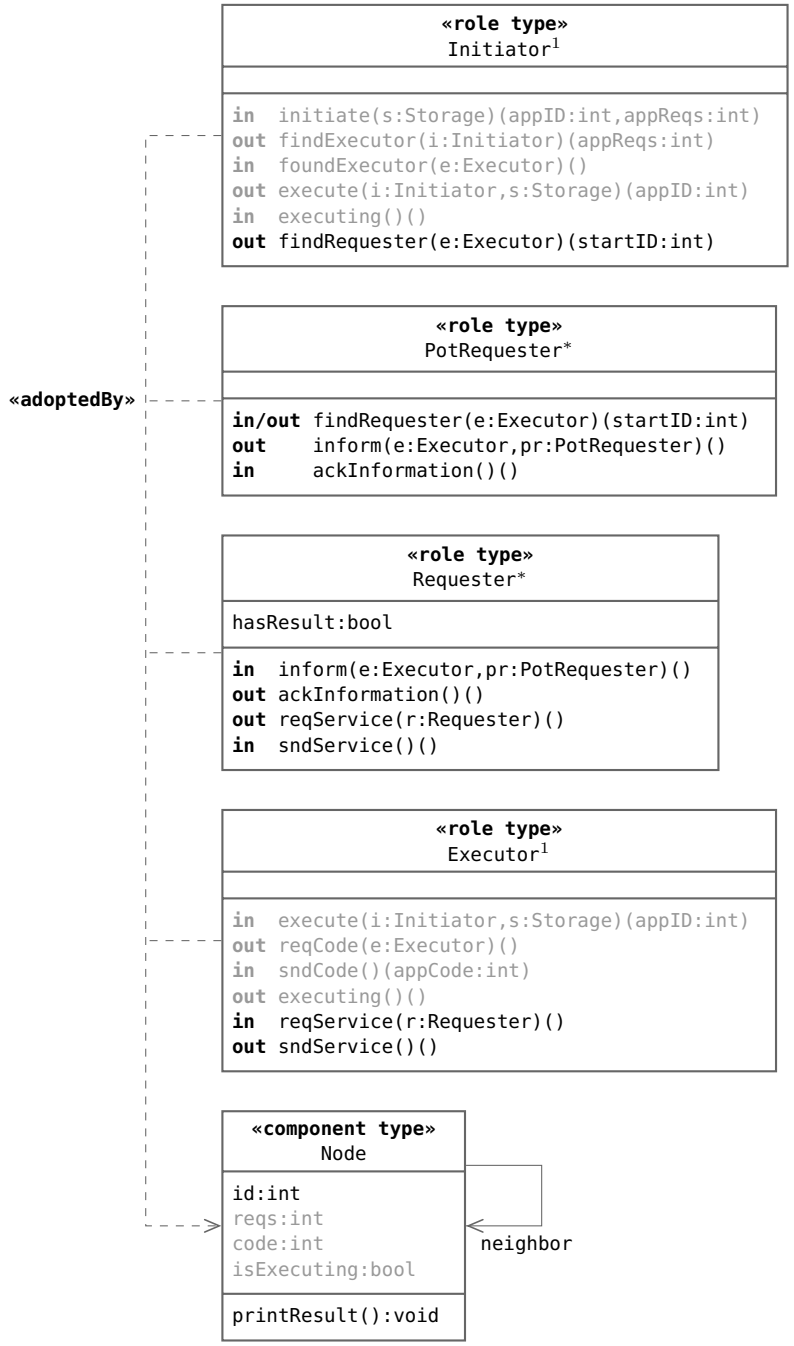


Figure 10.12: Ensemble structure for the SCP case study, part for keeping the application alive and providing access to the application (gray attributes, messages, and operations are not necessary for this task)

Role Behavior of the `Initiator` (continued): Fig. 10.9 shows the role behavior of the initiator in HelenaText from before. After having initialized the execution of the application, the initiator now triggers that all requesters in the ensemble get informed that the application is available. It first creates a potential requester on its neighboring node (line 10) and then sends the message `findRequester` to it (line 11). The data parameter of this message is important: it denotes the ID of the node where the search of all requesters in the ensemble started. The potential requesters will later on decide based on this parameter that they can stop to search when they reach the node with the ID represented by this parameter again. After that, the initiator quits the execution of its behavior since no more tasks remain for the initiator. If we also modeled failure and coordinated leaving of nodes, the initiator would have to take care that the application is kept alive and running. However, under the assumption of no failures and leaving, the initiator has finished its task.

Role Behavior of a `PotRequester`: Fig. 10.13 shows the role behavior of a potential requester in HelenaText. The idea of finding all requesters in the ensemble resembles the idea of finding an appropriate storage or an appropriate executor. We again assume that all nodes in the network are arranged in a ring structure. The potential requesters traverse this ring structure by the message `findRequester` one time. Whenever a requester is found during the round trip, the requester is informed about the availability of the application.

In more detail, each potential requester executes the behavior given in Fig. 10.13. It first receives the message `findRequester` (line 3). If its owning component currently also plays the role of a requester (line 5), it informs the requester about the availability of the application (line 6–8). It first retrieves the reference to the requester on its owning node (line 6) and then informs the requester by sending the message `inform` (line 7). Afterwards, it waits for an acknowledgment that the requester received the information by waiting for the message `ackInformation` (line 8). Finally, the current potential requester forwards the search message `findRequester` to its neighboring potential requester (line 9 resp. line 17–19) if the round trip has not already been finished. This is decided based on whether the ID of the owning node is the same as the parameter `startID` of the received message `findRequester`. If its owning component does currently not play the role of a requester (line 11), the potential requester just forwards the search message `findRequester` (line 12 resp. line 15–23) if necessary.

Role Behavior of the `Requester`: Fig. 10.14 shows the role behavior of the requester in HelenaText. In line 2, the requester receives the message `inform` indicating that the application is now available for requests. The requester acknowledges the information by sending the message `ackInformation` (line 3) to the potential requester `pr` which sent the information. Afterwards, it requests a service (like editing a Google Docs document) from the executed application via the executor `e` by sending the message `reqService` (line 5). Waiting for a response is simulated by waiting for the message `sndService` (line 6). Finally, we simulate that the result is stored by setting the attribute `hasResult` of the requester to **`true`** (line 7) and print it by calling the operation `printResult` of the owning node (line 8). With that, the requester finishes its behavior, i.e., all services which the requester requests from the application are represented by the single request in line 5.

```
1  roleBehavior PotRequester = PotRequesterProcess {
2    process PotRequesterProcess =
3      ? findRequester(Executor e)(int startID) .
4
5      if (plays(Requester,owner)) {
6        r <- get(Requester,owner) .
7        r ! inform(e,self)() .
8        ? ackInformation()() .
9        NextPotRequester
10     }
11     else {
12       NextPotRequester
13     }
14
15   process NextPotRequester =
16     if (owner.id != startID) {
17       pr <- create(PotRequester,owner.neighbor) .
18       pr ! findRequester(e)(startID) .
19       quit
20     }
21     else {
22       quit
23     }
24 }
```

Figure 10.13: Role behavior of a `PotRequester` in the SCP case study in HelenaText

```
1  roleBehavior Requester =
2    ? inform(Executor e, PotRequester pr)() .
3    pr ! ackInformation()() .
4
5    e ! reqService(self)() .
6    ? sndService()() .
7    self.hasResult = true .
8    owner.printResult() .
9    quit
```

Figure 10.14: Role behavior of a `Requester` in the SCP case study in HelenaText

Role Behavior of the `Executor` (continued): Fig. 10.11 shows the role behavior of the executor in HelenaText. After starting to execute the application, the executor is ready to receive infinitely many requests for the application in the process `ExecutorRunning` (line 10 resp. 12–15). It waits for a request by the message `reqService` (line 13) and sends back a response with the message `sndService` (line 14). Note that we did not model that the executor is stopped at some point in time. Therefore, it is always able to receive request via the recursive process invocation in line 15.

10.4.4 Summary and Discussion of the Behavior Specification

After having described each role and its role behavior for the SCP case study in detail, we want to recap the overall interaction behavior by looking again at Fig. 10.3. It makes the interplay between the collaborating roles clear. The diagram shows how the different roles interact to first store the application, then to execute the application, and finally to provide access to the application and request a service from it (we assume a single requester in the sequence diagram). We omitted to represent the search strategies of the potential storages, the potential executors and the potential requesters. They all follow the same pattern to traverse the underlying node ring structure in one

or two round trips such that the most suitable node(s) is found. That means for the potential storages the closest node based on ID proximity is searched, for the potential executors one node satisfying the computation requirements of the application, and for the potential requesters all nodes which currently want to request a service from the application. Apart from chronological order of the messages exchanged between the roles, the sequence diagram makes it also apparent that the deployer and the requester can independently be started. By waiting for the message `inform`, the requester synchronizes with the application deployment and execution by waiting until it got informed that the application is available and then requesting a service from the application.

Let us finally summarize all assumptions, which were made during the design of the SCP case study.

- We only consider the deployment of a single application in the network. This assumption can be alleviated by introducing arrays for storing and executing the application.
- We assume all peers in the network to be homogeneous. This assumption can be alleviated by introducing several underlying component types apart from `Node`, each offering different capabilities to its adopted roles.
- We do not model coordinated leaving of nodes. This assumption can be alleviated by introducing new logic for the initiator to monitor the execution of the application on the executor.

At this point, we also want to mention some restrictions underlying the HELENA approach which especially attracted attention during the design of the SCP case study.

- HELENA relies on binary communication and does not support broadcast yet. Broadcasting would facilitate the design of the case study insofar that instead of sequentially traversing the whole ring structure of nodes, we could just broadcast a message to the network and wait for appropriate answers. For example, when searching for an appropriate executor, the initiator could broadcast the request for an executor with the desired computation requirements to the network. Every node which satisfies the computation requirements could sent back a bidding to indicate that it would be able to execute the application. The initiator then simply selects one of the bidders as executor. Though (if available in HELENA) broadcast sending could be easily integrated in our process expressions, to collect corresponding answers is still an open issue.
- We build ensemble specifications on a given set of components such that we cannot model situations in which components fail uncontrolled. However, we are aware that one of the main characteristics of our case study is that nodes may fail at any time. We wish that such failovers are handled transparently from the role behaviors in HELENA. The idea is that components are monitored such that when failing all adopted roles are transparently transferred to another component and restarted there. This introduces another monitoring layer to the current HELENA methodology which is future work and not yet part of it.
- A further issue concerns robustness since we assume reliable network transmission in our HELENA models. We do not want to include any mechanisms for resending messages in the role behavior specifications. Like failover mechanisms, this should be handled transparently by an appropriate infrastructure underlying the current HELENA methodology.

10.5 Verification

Having specified the SCP case study in HELENA, it remains to show that the ensemble specification is directed towards achieving the goals formulated in Sec. 10.3. As explained in Chap. 5, the HELENA methodology proposes to prove satisfaction of goals by translating the HELENA model to PROMELA and model-checking the generated PROMELA specification with Spin.

The full generated PROMELA specification from the HELENA ensemble specification together with an initial state and the translated goals has 1569 lines of code and is shown in Appendix D.2. It can also be found on the attached CD in the project `eu.ascens.helenaText.scp` in the file `promela-gen/2nodes/scp.pml`.

10.5.1 Translation from HELENA to PROMELA

For the translation from HELENA to PROMELA, we rely on the HELENA workbench in Chap. 8. The workbench allows to specify the SCP case study in HELENATEXT as described in the previous section and automatically generates the corresponding PROMELA specification following the translation rules in Sec. 5.2.2. However, to preserve semantic equivalence between the HELENA ensemble specification and its PROMELA translation the assumptions described in Sec. 5.1 have to be respected. In the case of the SCP case study, we have to take care that process invocation is not used as a branch in any if-then-else construct. To meet these assumptions, we inline the process invocations of the process `Fwd` in the role behavior of a `PotStorage` and the process invocations of the process `NextPotRequester` in the role behavior of a `PotRequester`. The extended HELENA ensemble specification is shown in Appendix C.1.

The full generated PROMELA specification from this adapted ensemble specification has 1569 lines of code and is shown in Appendix D.2. It contains a process type `Node` (excerpt shown in Fig. 10.15) which reflects the corresponding component type of the HELENA model. As explained in Sec. 5.2, its parameters reflect all component attributes (line 2 in Fig. 10.15) and associations (line 3) as well as its own input channel `self` (line 4). The component process for a `Node` waits with a `do`-loop (line 10–23) for requests from its adopted roles on its `self` channel. Depending on the request, it runs some internal computation and sends back a reply, e.g., in line 14, it sends back the value of the component attribute `id` or spawns a new process representing a `Deployer` in line 17–20.

Another set of eight processes represents all role types and their behaviors as specified in HELENATEXT. As an example, the role process for the `Storage` is shown in Fig. 10.16. Its only parameters are the channel `owner` for communication to its owning node and its own input channel `self`. In the process term itself, first, parameters for all created role instance and formal role instance and data parameters are created (line 4–7). The rest of the process reflects the first four actions of the role behavior of the `Storage` of Fig. 10.7. Line 9 reflects receiving the message `store` (thereby, the second and the third item of the content of the message represent only dummy values since the message `store` does not have any role instance parameters). Afterwards, the component attribute `code` is set by issuing a request to the owning node in line 11–16. An `Initiator` is created on the owning node of the `Storage` (cf. line 18–23) and the message `initiate` is sent to it (cf. line 25).

```
proctype Node(
    int id; int reqs; int code; bool isExecuting;
    chan neighbor;
    chan self) {

  chan deployer = [1] of { mtype,chan,chan,int,int,int };
  ...

  NodeOperation op;
  do
  ::atomic {
    self?op ->
      if
      ::op.optype == GET_ID -> op.answer!id
      ::op.optype == SET_ID -> op.parameters?id
      ...
      ::op.optype == CREATE_DEPLOYER ->
        ...
        run Deployer(self, deployer);
        op.answer!deployer
      ...
  }
  od
}
```

Figure 10.15: Excerpt of the component process for the component type `Node` for the SCP case study in Promela

```
proctype Storage(chan owner, self) {
  ...

  int appID;
  int appReqs;
  int appCode;
  chan i;
  ...

  self?store,1,1,appID,appReqs,appCode;

  NodeOperation op;
  op.optype = SET_CODE;
    chan parameters = [0] of { int };
  op.parameters = parameters;
  owner!op;
  parameters!appCode;

  NodeOperation op;
  op.optype = CREATE_INITIATOR;
    chan answer = [0] of { chan };
  op.answer = answer;
  owner!op;
  answer?i;

  i!initiate,self,1,appID,appReqs,1;

  ...
}
```

Figure 10.16: Excerpt of the role process for the role type `Storage` for the SCP case study in Promela

10.5.2 Preparation of the PROMELA Translation

To be able to check the satisfaction of the goals of the SCP case study in the PROMELA specification, we have to manually prepare the generated specification.

Firstly, the initial state starting from which the goals shall be checked must be established in the PROMELA specification. We assume an initial state in which two nodes, one deployer and two requesters (one on each node) exist. We do not specify which node is closest to the deployed application based on ID proximity, but we take care that it is exactly one and that all IDs are greater than 0. We additionally guarantee that at least one node satisfies the application's computation requirements. The deployer is owned by either of the two nodes and each of the two nodes adopts exactly one requester.

This initial state is represented by the `init`-process in Fig. 10.17. In line 10–13, we nondeterministically choose which node gets the ID 100 which is closest to the hard-coded ID 1234 of the application. In line 14–17, we take care that one node satisfies the computation requirements 1234 of the application. In line 19–20, two processes are spawned which represent the two nodes forming the underlying peer network. In line 22–26, it is nondeterministically decided which of the two nodes is the owner of the deployer, and in line 27–32, which node is the owner of which of the two requesters. In line 34–39, three processes representing the deployer and the two requesters are spawned.

Finally, we have to take care that the behaviors of all three initial roles start at the same time. Therefore, we need to adapt the automatically generated process types for each initial role. Their first action is to wait for a dedicated message `setOffInitialRole` which sets of the behavior of an initial role. In the `init`-process, this message is sent to all three initial roles as the last indivisible atomic sequence in line 41–47. If we do not care that all initial roles wait for that message before executing their role behavior, it might happen that one role already proceeds in its behavior before the other initial roles are initialized at all.

As a second step, the goals of the SCP case study need to be translated to PROMELA as described in Sec. 5.3.1. All operators in the goals are translated to their ASCII representatives (e.g., $\wedge$ is translated to `&&`). Expressions referring to role attributes are prefixed with `roleAttr_`. To reflect satisfaction of LTL formulae relatively to an initial state, each translated goal ϕ has to be adapted to $\Box(\mathit{init} \Rightarrow \phi)$. *init* thereby is a property which only holds when the initialization in the `init`-process was finished and thus the initial HELENA state was established. We use `Deployer@startDeployer` for this *init* property.

With these adaptations, the goals from Table 10.1 are given as the inline specifications in Fig. 10.18. All goals are directly translated except the first goal "code gets stored". In PROMELA, the absolute value function is not built-in. Therefore, the comparison $|Node[1]{:}id - Deployer{:}appID| < |Node[2]{:}id - Deployer{:}appID|$ must be fully expanded for all possible values of the difference between $Node[1]{:}id$ (or $Node[2]{:}id$) and $Deployer{:}appID$. However, with this full expansion, Spin can no longer handle the LTL property due to its syntactic size and fails with memory buffer overflow. Thus, we removed the this comparison from the first goal.

```
1  init {
2    chan n1 = [0] of { NodeOperation };
3    chan n2 = [0] of { NodeOperation };
4
5    int id1 = 1;
6    int id2 = 2;
7    int reqs1 = 1;
8    int reqs2 = 2;
9
10   if
11   ::id1 = 100;
12   ::id2 = 100;
13   fi;
14   if
15   ::reqs1 = 1234;
16   ::reqs2 = 1234;
17   fi;
18
19   run Node(id1,reqs1,0,0,n2,n1);
20   run Node(id2,reqs2,0,0,n1,n2);
21
22   chan ownerDeploy;
23   if
24   ::ownerDeploy = n1;
25   ::ownerDeploy = n2;
26   fi;
27   chan ownerReq;
28   chan ownerReq2;
29   if
30   ::ownerReq = n1;ownerReq2 = n2;
31   ::ownerReq = n2;ownerReq2 = n1;
32   fi;
33
34   chan deploy;
35   node_retrieveRole(CREATE_DEPLOYER,ownerDeploy,deploy);
36   chan req;
37   node_retrieveRole(CREATE_REQUESTER,ownerReq,req);
38   chan req2;
39   node_retrieveRole(CREATE_REQUESTER,ownerReq2,req2);
40
41   atomic {
42     Msg msg;
43     msg.msgtype = setOffInitialRole;
44     deploy!msg;
45     req!msg;
46     req2!msg;
47   }
48 }
```

Figure 10.17: The `init`-process for the SCP case study in PROMELA

```
ltl Achieve_CodeStored { // code gets stored
  [] ( Deployer@startDeployer ->
    <> (
         (Node[1]:code > 0 && Node[1]:code == Deployer:roleAttr_appCode)
      || (Node[2]:code > 0 && Node[2]:code == Deployer:roleAttr_appCode)
    )
  )
}
ltl Maintain_Storage { // code stays stored
  [] ( Deployer@startDeployer ->
    [] (
      (  (Node[1]:code > 0 && Node[1]:code == Deployer:roleAttr_appCode)
      || (Node[2]:code > 0 && Node[2]:code == Deployer:roleAttr_appCode)
      )
      -> [] (  (Node[1]:code > 0 && Node[1]:code == Deployer:roleAttr_appCode)
            || (Node[2]:code > 0 && Node[2]:code == Deployer:roleAttr_appCode)
            )
    )
  )
}
ltl Achieve_AppExecuted { // app gets executed
  [] ( Deployer@startDeployer ->
    (Node[1]:reqs == Deployer:roleAttr_appReqs
    || Node[2]:reqs == Deployer:roleAttr_appReqs)
    ->
    <> (  (Node[1]:reqs == Deployer:roleAttr_appReqs && Node[1]:isExecuting)
        || (Node[2]:reqs == Deployer:roleAttr_appReqs && Node[2]:isExecuting)
    )
  )
}
ltl Maintain_Execution { // app stays executed
  [] ( Deployer@startDeployer ->
      [] (
        (Node[1]:isExecuting || Node[2]:isExecuting)
        -> [] (Node[1]:isExecuting || Node[2]:isExecuting)
      )
  )
}
ltl Achieve_Usage4 { // requester 4 gets served
  [] ( Deployer@startDeployer ->
    <> Requester[4]:roleAttr_hasResult
  )
}
ltl Achieve_Usage5 { // requester 5 gets served
  [] ( Deployer@startDeployer ->
    <> Requester[5]:roleAttr_hasResult
  )
}
```

Figure 10.18: The goals for the SCP case study in PROMELA

10.5.3 Model-Checking the PROMELA Translation

Model-checking of the PROMELA translation against the translated SCP goals is done in several steps. We start from a fast and lossy search algorithm and move to slower, but lossless optimization techniques as described in Sec. 5.3.2. In the following, we report on experiences when model-checking the HELENA design model with Spin. We expose some pitfalls which we experienced with the original HELENA model and how we improved it to the model described in the previous section. Afterwards, we discuss the statistics for model-checking the SCP case study with lossless optimization techniques. We omit the statistics for the other search algorithms since they were only used for improving the original HELENA model and only give a restricted impression about the size of the HELENA model in PROMELA.

10.5.3.1 Pitfalls and Improvements of the Original Helena Model:

To get a first impression about the satisfaction of goals, we start with bit state space search. With this lossy optimization technique, we get a first and most importantly fast impression about the satisfaction of goals and can get rid of some obvious errors in the Helena specification. If the specification passes all goals in the bit state space search, we move to the slower optimized search algorithm hash-compact. This is also a lossy optimization technique, but it most likely reaches a coverage of 100% in contrast to bit state space search. After having eliminated errors with hash-compact search, we finally use only lossless optimization techniques for model-checking. We employ Partial Order Reduction, Collapse compression and the Minimized Automaton representation at the same time gaining a full coverage of the model with a high compression rate at the price of a highly increased runtime.

Verification of the Helena model helped us to get rid of all design errors and to improve the model until it finally satisfies all goals of the SCP case study. Errors normally lie in two categories:

- First, we find *behavioral errors* which impede the executability of the overall system. Messages are sent to roles which are generally able to receive them, but do not expect them at the moment, a new instance of a certain role type is created on a component which already plays it, or an instance of a certain role type is retrieved from a component which does not currently play it. These errors occur if the role behaviors are not designed carefully and the semantic restrictions of action execution are not respected.
- If all those errors are eliminated from the specification, we find *logical errors* where the overall system can be executed, but the goals are not achieved. These errors concern the application logic represented in the role behaviors and require a deep understanding of the application domain to be resolved.

The different errors are essentially indicated by two error messages printed with the Spin output: `too many processes/queues created` or `acceptance cycle found`. In both cases, Spin provides a counterexample given as a trace through the Promela specification (cf. Sec. 5.3.2). To map the counterexample back to the original Helena model, it is necessary to fully understand the translation from Helena to Promela. It is still an open problem how to generate a trace in the original Helena specification instead of the trace in the Promela translation. However, these counterexamples helped us to understand design errors in the original Helena model of the SCP case study (in the previous subsection, we only showed the revised Helena model where all errors were already eliminated):

- The error message `too many processes/queues created` indicated that we specified an ensemble which created infinitely often new role instances. This happened for example during first attempts to find all requesters in the network. We cycled infinitely often through the complete peer network, every time creating a new potential requester on each node.
- The error message `acceptance cycle found` had two different causes. The first one was that the execution of the Promela specification got stuck and could not execute any further steps. Therefore, the finite trace was implicitly elongated with τ actions by Spin such that the generated counterexample cycled through τ actions. This problem is easily recognized by the phrase `START OF CYCLE` at the end of the trace. Our specification of the SCP case study got only stuck because

of behavioral errors as described in the previous paragraph.

- The error message `acceptance cycle found` was furthermore caused by real cycles in the specification, i.e., logical errors in the original HELENA model. For example, in a second attempt to find all requesters in the network, we cycled infinitely often through the complete peer network, but this time we did not create new potential requesters on every node, but reused the previously created one. Nevertheless, cycling through the network did not stop in this design model and therefore an acceptance cycle was found.

Unfortunately, we experienced problems with using the hash-compact search method on our SCP model. It checked the goals on the same model nondeterministically satisfied or unsatisfied. If the goals were unsatisfied, the model-checking output of Spin stated that an acceptance cycle was found, but the generated counterexample did not contain any cycles. Therefore, we decided to explore and improve our models with bit state space search for a first approximation of satisfaction of goals and then moved directly to full state space search with all compression techniques.

10.5.3.2 Statistics of Model-Checking the Final HELENA Model:

We performed model-checking of the SCP case study with Spin version 6.4.4 64-bit and GCC version 6.9.2 64-bit on a 64-bit Debian 8.1 desktop computer with 32GB RAM and eight Intel(R) Xenon(R) cores each running on 3.40GHz.

Model with Two Nodes: In the first round, the PROMELA specification was initialized with the `init`-process in Fig. 10.17. That means that two nodes formed the underlying peer network where one is closest to the application based on ID proximity and one node satisfies the application's computation requirements. On top of that, one deployer and two requesters (two on each node) were started while nondeterministically choosing one of the nodes as owner. To repeat from the description of the initial state in Sec. 10.5.2, by initializing the PROMELA specification with this `init`-process, we made three important assumptions explicit: all IDs of nodes are greater than 0, at least one node satisfies the computation requirements, and all requesters are already created in the initial state.

We started with bit state space search for model-checking and improving the HELENA model. Each run lasted around 30 minutes and needed around 3GB of memory depending on the current goal to be checked and error-freeness of the model. We do not report an detailed statistics about these verification runs since they were only used for removing obvious errors in the first attempts of the specification of the SCP case study.

Table 10.2 reports the detailed results of model-checking the final HELENA model of the SCP case study against all its goals with a full state space search with Partial Order Reduction (POR) optimization, Collapse compression, and Minimized Automaton (MA) representation (100 is used as the estimate of the maximal depth of the graph).

All goals from Table 10.1 are satisfied by the final HELENA ensemble specification described in Sec. 10.4. The corresponding PROMELA model (initialized as explained above) to be checked is large: The state vector has a size of 2723 bytes (according to [HB07], state vectors around 4000 bytes are large). The state space grows from 215 million states for the goal "code gets stored" to 645 million states for the goal "code stays stored" with all other goals lying in-between (according to [HB07], state spaces with 10–600 million states are large). Without any compression techniques, the memory consumption for a full state space search would exceed today's memory capacities by

Table 10.2: Statistics of model-checking the SCP case study with two nodes against its goals with POR optimization, Collapse compression and Minimized Automaton (DMA=100) representation

	code gets stored	code stays stored	app gets executed	app stays executed	requester 4 gets served	requester 5 gets served
search depth	849	849	849	849	849	849
state vector size (B)	2723	2723	2723	2723	2723	2723
stored states (million)	215	645	219	643	483	483
transitions (million)	692	2767	708	2757	1632	1632
theoretic memory (GB)	566	1697	577	1691	1272	1272
actual memory (GB)	1.018	1.023	1.022	1.025	1.758	1.713
elapsed time (h)	3.4	12.6	3.5	12.4	7.3	7.4

far. 692 GB to 2767 GB would be necessary to check each of the goals. Fortunately, with the three lossless optimization techniques POR, Collapse and MA, the memory consumption can be reduced to 0.05–0.16% of the overall memory consumption such that only 1.0–1.8 GB of memory are needed. The memory consumption reaches its peek for the last two goals "requester 4/5 gets served" since compared to the other four goals, model-checking here has to especially focus on the interplay between application deployment and execution and requests to the application. The impressive compression rate comes with the downside of a highly increased runtime. A verification run of the SCP model takes between 3.4 hours and 12.6 hours. The runtime reaches its peek for the two maintain goals "code stays stored" and "app stays executed" since they require to maintain a certain property throughout the lifetime of the system while for satisfaction of achieve goals it is enough if the goal is achieved at some point in the lifetime of the system.

Model with Three Nodes: For the second round, the PROMELA specification was initialized with three nodes forming the underlying peer network. As before, one node is closest to the application based on ID proximity and one node satisfies the application's computation requirements. On top of that, one deployer and two requesters (two on each node) were started while nondeterministically choosing one of the nodes as owner. By just adding this one additional node, the state space immediately exploded. Full state space search for one goal only (e.g., "code gets stored") ran out of memory with a vector size of 3827 bytes and 18291 millions states stored in 31 GB of memory after 425 hours (approx. 17.7 days). It was only possible to check the model with bit state space search (cf. Table 10.3). However, these verification runs only produce a hash factor of around 3 such that we cannot expect full coverage as we discussed in Sec. 5.3.2.

10.6 Implementation

In the last section, we showed that the ensemble specification of the SCP case study from Sec. 10.4 satisfies all goals from Sec. 10.3. In this section, we explain how we realize the HELENA model on top of the already existing technology stack of the SCP (cf. the SCP Node layer in Fig. 10.1).

Table 10.3: Statistics of model-checking the SCP case study with three nodes against its goals with bit state space search and a memory limit of 30 GB

	code gets stored	code stays stored	app gets executed	app stays executed	requester 5 gets served	requester 6 gets served
search depth	994	1114	1117	988	988	986
state vector size (B)	3571	3827	3827	3571	3571	3571
stored states (million)	48	41	12	36	15	21
transitions (million)	175	192	126	197	139	139
theoretic memory (GB)	165	149	45	123	52	71
actual memory (GB)	0.0024	0.0024	0.0029	0.0018	0.0018	0.0018
elapsed time (min)	13.32	14.1	10.68	14.1	11.85	11.45
hash factor	2.79	3.30	3.32	3.75	3.50	3.48

Our code generator in Sec. 8.4 already generates a prototypic implementation in Java from the HELENA model. The generated implementation relies on the jHELENA framework from Chap. 7 which transfers and realizes the main HELENA concepts of roles and ensembles in Java. Both the generated code and the underlying jHELENA framework are used as a basis for the experimental realization of the SCP and are adapted to built on the existing technology stack of the SCP.

The HELENA methodology separates between base components and roles running on top of them. The technology stack of the SCP already provides the implementation of the components, that is the *SCP node layer* in Fig. 10.1. Thus, we transfer the ideas of the jHELENA framework to a new HELENA *SCP middleware* (cf. Fig. 10.1) which connects the SCP node layer with role-related functionality. The offered role-related functionality is the ability to create and retrieve roles via the network as well as routing messages between roles by using Pastry. Furthermore, the HELENA SCP middleware also provides access to the storage capabilities of the underlying SCP nodes using PAST. In a second step, we translate the behavioral specifications of all roles to Java code following the ideas of the code generator for jHELENA, but taking into account the newly created HELENA SCP middleware.

In the following two subsections, we discuss the HELENA SCP middleware and the adapted role behavior implementations, respectively, stressing where the concepts of the jHELENA framework could directly be used, where the HELENA model could directly be translated to Java and where special care had to be taken to make the realization robust.

The final implementation of the SCP can be retrieved from `http://svn.pst.ifi.lmu.de/trac/scp`, version v3 of the node core implementation with gossip strategy.

10.6.1 HELENA SCP Middleware

The HELENA SCP middleware needs to be able to manage the role layer on top of the SCP node as well as to provide all functionalities used by roles like creating and retrieving other roles and exchanging messages between roles as well as accessing the storage capabilities of the underlying components, the SCP nodes.

Role Management: We first focus on how roles are realized in Java and how they are administrated on top of the SCP nodes, independently from the actual behavior each role executes. The HELENA SCP middleware must support three types of management operations: the ability to create role types, to instantiate and execute them as well as to directly address them for message exchange.

We reuse the idea of jHELENA and naturally map a role type to a Java class and a role instance to an instantiation of this class. To enable concurrent execution of several role instances on one SCP node, each role instance is realized as a Java thread, running locally in the OSGi container of the current node. A registry on each node stores all instances currently adopted by the node such that we can easily retrieve already existing role instances of an SCP node. To retrieve and exchange messages with roles on other nodes, the HELENA SCP middleware requires a way of addressing roles. In Pastry, each node is already identified by a unique 160-bit identifier. It is relatively straightforward to add a similar unique identifier for roles. However, there is also another kind of structuring element which is not directly visible in the behavioral specifications: the ensemble which constitutes the environment for the roles. This can clearly be seen when looking at the functions the SCP middleware needs to offer for role handling – these are the `create` and the `get` actions. Both require knowledge about which ensemble is addressed for creating a new role or where to look for an existing role. A role instance is therefore characterized by three identifiers in the HELENA SCP middleware: the node identifier (for addressing nodes using Pastry), the ensemble identifier and the role identifier.

Infrastructure for Executing Actions: Besides managing roles, the HELENA SCP middleware must realize all actions a role can execute in its behavior. The first two are role creation and role retrieval. Similarly to the jHELENA framework, they are implemented as Java methods (`createRoleInstance` and `getRoleInstance`) in the HELENA SCP middleware. They require a node and an ensemble ID as well as the class of the required role as input. In contrast to the jHELENA framework, the owning SCP node of the role instance which wants to create (or retrieve) the role instance communicates via Pastry with the target SCP node which should own the newly created (or already existing) role instance. It instructs the target SCP node to create and start the new role (or retrieve it). The target SCP node returns the identifier of the newly created (or already existing) role which can then be used for role-to-role message routing.

Furthermore, the behavior specifications of roles make heavy use of role-to-role communication. A role must be able to send a message and to expect to receive a certain message in its behavior. For this purpose, the SCP middleware again provides two methods `sendMessage` and `waitForMessage` for communication between roles.

The method `sendMessage` takes a message and a target role. In contrast to the jHELENA framework again, the message is routed via Pastry and is put into the input buffer of the target role. The method only returns when this has been successfully completed (i.e., an internal acknowledge is sent back upon which the `sendMessage` function returns normally). Otherwise, an exception is raised. Of course, correct collaboration requires that any message is finally consumed from the buffer. Moreover, any consumed message should also be expected by the target role as an input message in accordance with its role behavior specification. Since we already checked the HELENA model with the model-checker Spin for goal satisfaction (cf. Sec. 10.5), this is trivially satisfied if all goals are achieved.

The second method is `waitForMessage` which instructs the SCP middleware to wait for a message of a certain type, The method takes a timeout value such that an exception is raised if a message does not arrive in the given time (though specifying `INFINITY` is an option).

For access to the owning node of a role, we do so far not provide any generic methods. Setting attributes of the owning node and calling operations on them is realized by calls to application-specific methods in the SCP node, e.g., the method `storeInPAST` on the SCP node for the realization of the operation `store` in the HELENA model.

10.6.2 SCP Role Behavior Implementations

Given the HELENA SCP middleware as a basic infrastructure for role management, communication between roles and to the owning SCP nodes, we can now proceed to derive the implementation of the HELENA model for the SCP case study.

Direct Translation: As discussed above, role types are implemented in Java using classes. Thus, for each of the eight role types from the previous section, a class is created, inheriting from an abstract role template for easier access to the SCP middleware methods. Each role is instantiated within a certain ensemble and SCP node. Upon startup, a dedicated method implementing the role behavior is called.

The actions in the role behavior specifications are translated to message exchanges and operations calls on the owning SCP node. For each message type, a message class with an appropriate name is created, and equipped with the required parameters as indicated in the role types. Instances of these classes are later transmitted through the network (appropriately wrapped as Pastry messages). For example, the `store` message shared between `Deployer` and `Storage` is implemented by an instance of the `StoreMessage` class which carries the application name as a field. As described before, the HELENA SCP middleware does not provide a generic way of setting attributes or calling operations on the owning SCP node. They are realized as explicit calls to application-specific methods of the SCP node. For example, setting the attribute `code` of the owning node in line 3 of the role behavior of the `Storage` in HELENA (cf. Fig. 10.7) is realized by calling the method `storeInPAST` on the SCP node.

Relying on these realizations of actions, a role behavior is translated into Java as follows:

- Transitions referring to the two middleware functions `create` and `get` are directly translated to calls to the corresponding framework methods `createRoleInstance` and `getRoleInstance`. They return role identifiers which can then be used for communication.
- Transitions with incoming messages, e.g., `?store()(int appID, int appReqs, int appCode)` in line 2 of the role behavior of the `Storage` (cf. Fig. 10.7), are translated into a call of the middleware method `waitForMessage` for the corresponding message class, e.g., the `StoreMessage`. The `waitForMessage` method returns an instance of the message once received, which can be queried for the actual ID, computation requirements and the byte code of the application.
- Transitions with an outgoing message, e.g., `i!initiate(self)(appID,appReqs)`, are translated into a call to the middleware method `sendMessage`. The message to be sent must be given as a parameter.

- Transitions which set the attributes of the owning node or call operations of the owning node are translated to the call of their application-specific counterpart on the SCP node.
- All other transitions, like guarded choice or recursive process invocation, are translated into their appropriate Java counterparts similarly to the code generation to jHelena in Sec. 8.4.

With this basic description, the role behaviors are directly translatable into Java code. As an example, Fig. 10.19 shows (in condensed form) the `run`-method of the `Storage` role which is directly derived from its behavior specification in Fig. 10.7. Line 2–5 correspond to receiving the message `store` in line 2 in Fig. 10.7. As in jHelena, the parameters of the received message are stored in attributes of the role (line 3–5). To store the byte code of the application (line 3 in Fig. 10.7), we use the dedicated method `storeInPAST` of the underlying SCP node (line 7) which stores the byte code in the distributed hashtable provided by PAST (cf. Sec. 10.1). To create an initiator on the owning node (line 5 in Fig. 10.7), the method `createRoleInstance` is called (line 9) where the first parameter is the identifier of the owning SCP node of the current role. The message `initiate` from line 6 in Fig. 10.7 is sent to the newly created initiator by calling the method `sendMessage` (line 11–12). Thereby, the first parameter represents the sending role instance and the second parameter the receiving role instance. The following three parameters denote the actual parameters of the `initiate` message. Similarly to receiving and sending before, also the message `reqCode` (line 8 in Fig. 10.7) is received by calling the method `waitForIncomingMessage` of the Helena SCP middleware and the message `sndCode` (line 9 in Fig. 10.7 is sent by calling the method `sendMessage`. Termination and especially quitting to play a role is not represented in the shown `run`-method. The registry which manages the current role instances per SCP node and ensemble takes care to quit the role.

```
1  public void run() {
2    StoreMessage storeMsg = waitForIncomingMessage(INFINITY, StoreMessage.class);
3    this.appID = storeMsg.getAppID()
4    this.appReqs = storeMsg.getAppReqs();
5    this.appCode = storeMsg.getAppCode();
6
7    this.getSCPNode().storeInPAST(this.appID, this.appCode);
8
9    RoleId iRole = this.createRoleInstance(this.getNodeId(), InitiatorRole.class);
10
11   sendMessage(new InitiateMessage(this.getRoleId(), iRole,
12     this.getRoleId(), this.appID, this.appReqs));
13
14   ReqCodeMessage reqCodeMsg = waitForIncomingMessages(INFINITY, ReqCodeMessage.class);
15   RoleId eRole = reqCodeMsg.getE();
16
17   sendMessage(new SndCodeMessage(this.getRoleId(), eRole, this.appCode));
18 }
```

Figure 10.19: Role behavior implementation for the `Storage`

Efficiency Improvements of the Translated Implementation: In principle, all role behaviors from Sec. 10.4 can be straightforwardly translated to Java building on the Helena SCP middleware. However, this direct translation does not exploit all benefits the SCP technology stack offers. Firstly, in the Helena model, we modeled the search

for the node with the ID closest to the application's ID by two round trips through the network via potential storages. In the implementation of the role behaviors in Java, this can be improved by using Pastry. With Pastry, the deployer can directly determine the closest node and can immediately create the storage on this node. Thus, the potential storages are no longer needed and replaced by more sophisticated functionalities of Pastry which cannot be modeled directly in HELENA. Secondly, in the HELENA model, we also modeled the search for an executor satisfying the computation requirements of the application by a round trip through the network via potential executors. In the real-life implementation, however, a gossip protocol spreads information about all SCP nodes through the network. With this information, each SCP node builds its own picture of the network, e.g., which nodes has which computation abilities. Therefore, the initiator can reduce the number of potential executors to be asked for execution to the SCP nodes which most probably satisfy the computation requirements of the application according to the gossiped information. Lastly, in the HELENA model, the initiator informs all requesters about the availability of the application via a round trip through the network with potential requesters. In the real-life implementation, the Java API offers possibilities to register observers. Thus, the initiator can directly inform only those nodes which registered as requesters for this application without a full round trip through the network. The implementation on `http://svn.pst.ifi.lmu.de/trac/scp`, version v3 of the node core implementation with gossip strategy, implements all these improvements of the original HELENA model.

10.6.3 Discussion of the Implementation

The formal HELENA model easily guides the implementation in Java. We first provide all role-related functionality by a HELENA SCP middleware. This middleware amounts to around 1000 lines of code (LOC). It reuses the idea of the jHELENA framework from Chap. 7 and connects the HELENA concepts of roles and ensembles with the underlying SCP technology stack. On top of that, the roles from the HELENA model are implemented. Each corresponding class stays below 150 LOC with another 400 LOC in message classes. The implementation is systematically derived from the role behaviors and therefore does not require much implementation effort. The encapsulation of responsibilities in separate roles helps to make the SCP code clean and easy to understand. On the other hand, the implementation of the HELENA SCP middleware with 1000 LOC might seem a significant amount of implementation effort just for providing role-related functionalities without any application logic. However, one has to keep in mind that whenever the application logic is changed, this middleware remains unchanged and only the role behaviors need to be changed which is much less effort. The SCP middleware can be used for any type of application based on a network of components which are voluntarily provided, store data, and interact in a peer-to-peer manner. Furthermore, by deriving our implementation of the SCP from its formal HELENA model, we can rely on the results from model-checking the formal model (cf. Sec. 10.5). Careful analysis guarantees that the formal HELENA model satisfies all specified goals from Sec. 10.3. A direct translation to Java as we discussed it preserves satisfaction of those goals in the implementation.

Nevertheless, some effort has to be taken to make the system robust against communication failures on the network layer and to communicate with the outside world. Additionally, special bootstrapping is needed to start the initial roles of the ensemble.

Each of the framework methods may fail for various reasons, and the resulting exceptions must be handled. Firstly, in all operations, timeouts may occur if a message could not be delivered. Secondly, role-to-role messages may fail if the target node does not (yet) participate in the expected ensemble or does not (yet) play an expected role; this also applies to the `getRoleInstance` method. The `createRoleInstance` may fail if the role class could not be instantiated or started. These errors are not captured in the role behaviors, but may occur in practice (in particular, they may occur during development if the implementation is not yet fully complete and stable). However, message could also be lost due to network errors. The HELENA SCP middleware has to take care to reliably deliver message, for example by resending messages until the reception is acknowledged.

Furthermore, there are also some points where the roles need to receive from and to return information to an outside party. For example, the `Requester` role is invoked when a UI request is made for an application; the response from the application must be presented to the user. This requires open ensembles and therefore explicit invocation of a role from an outside party and vice versa. One could think of specialized actions for these communication with the outside world or introduce answers a role in general gives to users.

A further issue is bootstrapping of HELENA ensembles. At each ensemble startup, at least one role needs to be instantiated by an outside party before messages can be received. In this case study, there are at least two initial roles which must be instantiated before the ensemble can be executed: the `Deployer` and at least one `Requester`. This bootstrapping point cannot be deduced from the local behavior specifications, but is given by the initial state of an ensemble. Therefore, it must be treated separately. In the case of the SCP, this part is played by the SCP UI (top right in Fig. 10.1).

10.7 Related Work

The science cloud platform is a peer-to-peer, voluntary-computing cloud platform-as-a-service. As one of the case studies in the ASCENS project [WHKM15], the SCP was thoroughly studied with techniques from this project. Mayer et al. [MKH+13, MVK+15] give an overview what type of design patterns help to capture adaptation of the SCP and how the SCP can be modeled in HELENA, specified in SCEL and implemented based on Pastry and the ContractNET protocol. In the context of the ASCENS project [MVK+15], the SCP case study was investigated for distributed Denial-of-Service attacks and defense patterns have been proposed, reliable routing between nodes was verified on top of Pastry, performance was monitored and predicted with SPL, and mobile nodes which can join and leave the SCP were explicitly supported by jDEECo.

As furthermore stated in [MKH+13], the related work in the three underlying computing paradigms has traditionally focused on a) routing and distributed storage of data (p2p), b) distributing workload from a central server (voluntary computing), and c) provisioning resources inside centralized data centers (cloud computing). Combining these areas has started to attract attention in recent years; we believe however that this research is far from concluded.

Voluntary clouds have been identified as a recent research trend in a state-of-the-art survey [ZCB10] in 2010. Another survey-type paper [PBF+11] by Panzieri et al. from 2011 also lists implementing cloud implementation on top of P2P networks as an open problem, and observes the usually centralized nature of voluntary computing. Panzieri et al. list the work by Babaoglu et al. from 2006 [BJK+06] as the first proposal for a

"fully decentralized P2P cloud". The group has since followed up with additional works, of which a very interesting recent one is [BMT12] from 2012, which implements a similar system to the one presented here on the infrastructure-as-a-service level.

An approach to bridge volunteer and cloud computing, but without going for a fully decentralized organization, is the work by Cunsolo et el. [CDPS09] in 2009. There, the idea is for users to contribute additional resources to certain centralized components. Also in 2009, Chandra and Weissman [CW09] have come up with the term *Nebulas* instead of clouds for distributed voluntary resource use. They list three requirements for such systems, which we have addressed partially in this paper.

10.8 Publication History

The idea and the underlying concepts of the science cloud platform have already been presented in [MKH+13]. The concept of roles have been applied to the science cloud platform and its implementation in [KMH14]. This chapter goes beyond these two publications by exercising the whole HELENA development methodology on the case study. In particular, we improved the original HELENA model of the science cloud platform upon verification results. The model is extensively verified and verification results and limitations are discussed in full detail. The HELENA SCP middleware has already been proposed in [KMH14] for implementing the science cloud platform with HELENA concepts on its real distributed platform, but we emphasize its strong resemblance to the jHELENA framework.

10.9 Present Achievements and Future Perspectives

This chapter showed that the HELENA development methodology can be applied to a larger software system and that it enables a holistic development of the system in mind. The Science Cloud Platform (SCP) served as our case study. The SCP is able to deploy and execute a software application in a voluntary peer-to-peer network as well as to provide access to the deployed application for users.

Domain Model: Starting from an informal description of this case study, we first derived a domain model with one component type and five roles. Splitting the task of application execution in several independent roles thereby most naturally represented the domain and helped to understand the individual subtasks.

Goal Specification: Based on this domain model, the informal (achieve and maintain) goals of the case study were formalized as LTL formulae. Goals which required persistent storage or persistent execution of the application were expressed by referring to attributes of the underlying component-based platform of the SCP. Goals which expressed individual objectives like being served exploited attributes of roles running on top of the component-based platform. LTL proved to be an appropriate logic to formulate the goals of the SCP case study. Nevertheless, an interesting extension would be to use first-order LTL to be able to include quantifiers in the formalization of goals.

Design Model: Having the goals in mind, we enhanced the domain model with behaviors for each participating entity of the HELENA model as well as further roles to allow appropriate collaboration in the ensemble. Thereby, the benefits of the role-based

modeling approach of HELENA became apparent: The SCP is comprised of four subtasks: deploying the application, finding an executor, keeping the application alive, and providing access to the application. In a standard component-based design, we would have to come up with a single component type for a computing node which is able to combine the functionalities for each responsibility in one complex behavior (this is the case in the previous "all-in-one" implementation of the SCP [MKH+13]). By relying on HELENA, we benefited from the possibility to model the system in terms of collaborating roles: On the one hand, roles allowed to separate the definition of the capabilities and behavior required for a specific responsibility from the underlying component. On the other hand, adopting different roles allowed components to change their behavior on demand. Nevertheless, we missed several features in the HELENA modeling approach: Broadcast messages would improve the performance of the overall model since e.g., the search for an appropriate storage of the application could be realized by a bidding process instead of a full round trip through the underlying peer network. Waiting for several messages at the same time and proceeding according to the received message would allow to design a more flexible and externally controlled system. Lastly, a concept for waiting and being notified (e.g., about certain component attribute values) would increase performance since values would not have to be continuously polled.

Verification: The HELENA design model was translated to PROMELA and checked against its goals with the model-checker Spin. Therefore, we could examine the modeled system before implementation for any behavioral or logical error impeding the satisfaction of goals. The PROMELA model turned out to be quite large even for only two peers in the underlying peer network of the SCP since the HELENA model prescribes eight different role types and even more instances on top of that peer network. However, Spin was able to fully model-check the PROMELA model with two nodes. We could continuously improve the HELENA design model with the help of counterexamples generated from Spin until finally satisfaction of all goals was proven. Nevertheless, model-checking with Spin would benefit from a reduction of the model size. This could either be achieved by a new design model in HELENA, e.g., with broadcasting possibilities, or by a more space-efficient translation to PROMELA where e.g., the data stored on components or the currently adopted roles of components are expressed by global variables instead of by local variables of independently running component processes.

Implementation: During implementation of the model, translating the role behaviors to Java code has proven to be straightforward. To gain this complexity reduction, first a (reusable) HELENA SCP middleware layer was needed to provide HELENA-specific functionalities. The HELENA SCP middleware increased the development effort compared to the previous "all-in-one" implementation of the SCP [MKH+13], but once developed it does not have to be changed if the logic of the SCP needs adaptation. Furthermore, the encapsulation of responsibilities and subtasks in separate roles helped to make the SCP code clean and easy to understand. Special care had to be taken in four areas which are implicit in HELENA specifications: handling faults during communication (HELENA assumes reliable communication), node identification for role creation and retrieval, handling node failures, and communication between ensembles and the outside world. For future work, it would thus be interesting to include an infrastructure in HELENA which copes with unreliable systems and failing components. One idea is to snapshot the system regularly such that roles can be transferred to other components when a component fails.

The HELENA Development Methodology: Generally speaking, the HELENA development methodology allowed a rigorous development of the SCP case study. The HELENA concepts of roles and ensembles provided reasonable abstractions which served as a clear documentation, analysis model, and guideline for the implementation. With HELENA, goals could be formally captured and checked in the HELENA design model of the SCP case study due to the formal foundation of the HELENA approach. The implementation of the SCP case study could systematically be derived from the HELENA model and achieves all its goals with high confidence due to the systematic derivation from the formally verified HELENA model.

Chapter 11

Role-Based Adaptation
Being Adaptive with HELENA

The HELENA development methodology especially aims at the development of ensemble-based systems. However, the ideas and concepts can be tailored to other application domains as well. In this chapter, we propose a holistic model-driven engineering process on top of the HELENA development methodology to develop self-adaptive systems. It is a comprehensive and coherent methodology for engineering self-adaptive systems where adaptation logic and application logic is well separated through all development phases and artifacts are easily traceable through the whole development process.

Key concept of the HELENA approach to self-adaptive systems is to achieve adaptation by changing the behavioral mode of a component in response to perceptions and to realize behavioral modes by roles which a component can dynamically adopt. Furthermore, the new model-driven engineering process provides systematic transitions between all phases: specification, design, verification, and implementation.

For specification, we introduce adaptation automata which allow to specify complex adaptation behavior by hierarchical structure and history of states similar to UML state charts. Furthermore, we propose the HELENA Adaptation Manager pattern to derive a role-based HELENA model from a specification. The pattern exploits the concept of roles as proposed in HELENA to represent modes and their behaviors and therefore allows to adapt a component's behavior by changing the currently active role. Due to its formal foundation, the HELENA model can then be analyzed with Spin and executed with the Java framework jHELENA.

In the following, we first give an overview about the current state-of-the-art of engineering self-adaptive systems to motivate the need for a holistic development process in Sec. 11.1. Table 11.2 gives an overview of the whole HELENA development process for self-adaptive systems. Table 11.3 introduces a robotic search-and-rescue scenario which we use as a running example. Our specification technique is discussed in Sec. 11.4. The HAM pattern to refine an adaptation specification to a design is presented in Sec. 11.5. The systematic transition from a specification to a formal HELENA model following the HAM pattern is divided in two parts. The derivation of the role-based architecture is described in Sec. 11.6, the derivation of the dynamic behaviors in Sec. 11.7. The further systematic transitions to a verification model and to jHELENA code are not described here since they rely on the ideas introduced in Chap. 4, Chap. 5, Chap. 7, and Chap. 8. In Sec. 11.8, we compare our methodology to prominent reference models for self-adaptation. We conclude in Sec. 11.10 with a discussion and some hints on future work.

The complete HELENA ensemble specification of the running example proposed later on has 723 lines of code and is listed in Appendix E. It can also be found on the attached CD in the file `search-and-rescue.helena`.

11.1 Introduction

To cope with changing conditions at runtime, the autonomic computing paradigm [KC03] advocates to equip components with the capability of self-adaptation. A self-adaptive component keeps track of its individual and shared goals, perceives its internal state as well as its environment, and adapts its behavior accordingly [C+09, ST09]. We say, a self-adaptive component changes its *behavioral mode* in response to perceptions.

Engineering Self-Adaptive Systems, State-of-the-Art: It is commonly agreed that adaptation logic of self-adaptive systems (SAS) should be developed independently from application logic [C+09, BMSG+09, IBM06]. Cheng et al. [C+09] even argue that the feedback loop implementing the adaptation logic must become a first-class entity throughout the whole development process. By adhering to this principle of separation of concerns, the application logic of behavioral modes can be developed without considering changes in the environment. Conversely, adaptation logic is designed to switch between behavioral modes without taking care how the modes perform their task.

In the literature of SAS, this separation of concerns is addressed at different development phases: Automata-based approaches [LE13, ZMLL11, MPT12, BCG+13] offer formal specification and verification techniques. Architectural patterns [PCZ13] introduce design guidelines for realization. Role models [SWHB05, MT11] propose concepts for switching between behavioral modes. Architecture-based self-adaptation[OGT+99, GCH+04, KM07] is presented as a framework for designing and implementing SAS.

Consequences: None of the approaches addresses separation of concerns consistently from specification over verification to design and implementation. We are missing a holistic development process considering adaptation logic independently from application logic in all main phases and supporting systematic transitions between all of them. Therefore, artifacts cannot be easily traced through the whole process and we cannot guarantee correct realization of requirements since we lack systematic transitions between each of the phases.

HELENA Development Methodology for Self-Adaptive Systems: Integrating and extending existing approaches, we propose a holistic model-driven engineering process in this chapter to develop self-adaptive systems on top of the HELENA development methodology. It consistently separates adaptation logic from application logic in all main development phases. Key concept of the approach is to achieve adaptation by changing the behavioral mode of a component in response to perceptions and to realize behavioral modes by HELENA roles which a component can dynamically adopt.

- We introduce *hierarchical adaptation automata* to specify a self-adaptive component with its (possibly) complex adaptation behavior. They offer history states and hierarchical composition of states as expressive tools to specify complex adaptation behavior and they provide placeholders to plug application logic in.

- For the design, we propose a role-based architecture following the HELENA *Adaptation Manager (HAM) pattern*. Roles as proposed in HELENA intuitively express the different tasks of self-adaptive components, like being aware of the environment, managing adaptation, and performing particular behavioral modes.
- We describe a *systematic transition* (which can be fully automated) from specification to role-based design. Especially, representing an adaptation automaton as a standard labeled transition automaton is particularly involved since hierarchical structure and history states have to be resolved.
- By realizing the role-based design with HELENA, we benefit from its *verification and implementation tools* through reusing its automatic transformations to PROMELA (cf. Chap. 5) and jHELENA (cf. Chap. 7).

Thus, we introduce the missing *traceability of artifacts* throughout the whole engineering process while keeping adaptation logic and application logic separated.

11.2 HELENA Development Methodology for Self-Adaptive Systems

The HELENA development methodology for self-adaptive systems (cf. Fig. 11.1) is a holistic methodology to develop self-adaptive systems. It imposes the principle of separation of concerns on all of its four main phases – specification, design, implementation and verification – and supports systematic transitions between the phases.

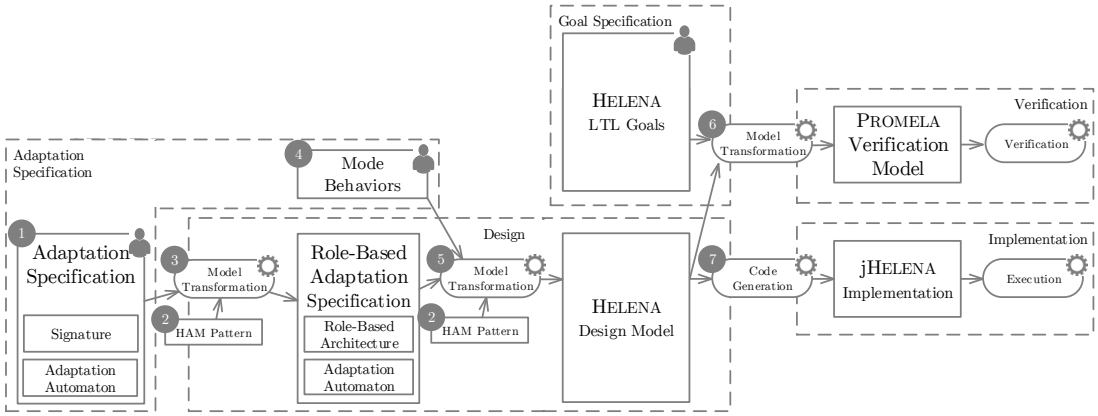


Figure 11.1: The HELENA development methodology for self-adaptive systems (repeated from Sec. 1.5). Rectangular boxes denote artifacts, boxes with rounded edges are activities. The boxes marked with a user icon are artifacts which have to be created by the specifier; all other artifacts can be systematically derived by the shown activities.

(1) Adaptation Specification (Sec. 11.4): We first specify the self-adaptive components which contribute to the system. The *signature* captures the static properties of the component, i.e., its attributes and its types of behavioral modes. *Adaptation automata* are used to describe when the component switches between behavioral modes and provides placeholders to later on plug the application logic in. Adaptation automata provide a rich specification technique by allowing to hierarchically compose states and to keep track of the history of previously executed behaviors like in UML state charts.

(2) Design Pattern (Sec. 11.5): We propose the *HAM pattern* to create a role-based HELENA model for self-adaptive components. We exploit the concept of roles to

encapsulate modes and their behaviors. A component's behavior is adapted by changing the currently active role of the component. An adaptation manager, also represented by a role of the component, takes care to realize switching between mode roles according to the logic specified in the adaptation automaton.

(3) Model Transformation Part 1 (Sec. 11.6): We systematically derive a *role-based architecture* following the HAM pattern for the initially specified self-adaptive component and its adaptation automaton.

(4) Specification of Application Logic: After the first model transformation, we can now rely on the role-based architecture to specify the application logic of the behavioral modes in the form of *mode role behaviors*.

(5) Model Transformation Part 2 (Sec. 11.7): We integrate the mode role behaviors and the adaptation automaton as a role behavior into the role-based architecture to gain a fully specified HELENA *design model*.

(6) Verification: As already explained in Chap. 5 and exercised at our Science Cloud Platform case study in Chap. 10, HELENA models can be formally analyzed for goal satisfaction. We systematically transform HELENA models to PROMELA which is fully automated as described in Sec. 8.3. The generated PROMELA verification model is then verified against goals specified as temporal properties using Spin. This approach can be reused to check the satisfaction of goals of a self-adaptive component.

(7) Implementation: Finally, the HELENA methodology provides the execution framework jHELENA which transfers the concept of roles to Java (cf. Chap. 7). We can rely on this framework to implement and execute our HELENA model. An automatic code generator from HELENA specifications to jHELENA code was already presented in Chap. 8.

11.3 Search-and-Rescue Scenario

One of the case studies of the EU-project ASCENS [WHKM15] is a robotic search-and-rescue scenario [DLPT14]. Robots are distributed over an unknown area where recently some kind of disaster happened. The robots have to find victims and to transport them to a rescue area. Since we do not want any human to enter the dangerous area, the robots have to self-adaptively manage their behaviors. For example, during searching for victims, a robot searches randomly and informs other robots about the location of found victims. If it was informed about the victim's location by another robot, it switches to a directed walk towards the victim. A robot also changes its behavior whenever it reaches a victim during search and starts to help rescuing the victim. Orthogonally, the robot may run out of battery at any time. Then, the robot has to switch to a low-power behavior and waits for another robot to get recharged.[1].

11.4 Adaptation Specification

The first step in the development process in Fig. 11.1 is the adaptation specification. A self-adaptive system is composed of a set of self-adaptive component types whose instances form the system at runtime. For each self-adaptive component type, we specify its *signature*, i.e., its attributes and behavioral modes without any application logic. Additionally, we describe the rules how each component (of a certain type) changes

[1] We do not claim that our specification represents the robotic scenario in full detail, but rather want to highlight the benefits of our methodology at a simple application.

its current behavioral mode depending on perceptions by an *adaptation automaton*. The automaton only specifies conditions under which the component switches between behavioral modes such that it leaves placeholders where we can later on plug the application logic in.

In the remainder of this section, we formally define the signature of self-adaptive component types and adaptation automata. We illustrate the specification with our search-and-rescue scenario and highlight the benefits of the proposed specification technique.

Notation: Whenever we consider tuples $t = (t_1, \ldots, t_n)$ in the following, we may use the notation $t_i(t)$ to refer to t_i.

11.4.1 Signature of a Self-Adaptive Component Type

The *signature of a self-adaptive component type* describes the static properties of the component. It has a name and stores normal data in attributes. Additionally, it stores perceptions about the state of the environment and its own state as awareness data. While component data may also be perceptions, it does not directly trigger adaptation. However, it may transitively influence awareness data which is then the ultimate source to decide whether to adapt. Finally, each component type supports a set of behavioral modes between which the component can switch in response to changes of awareness data. At this stage, we do not take care of the actual behavior in each mode since we concentrate on the adaptation logic only.

Def. 11.1: Signature of a Self-Adaptive Component Type

The signature of a self-adaptive component type *is of the form* $ct = (nm, attrs, attrs_{aware}, modes)$ *such that*

- nm *declares the name of the self-adaptive component type,*
- $attrs$ *is a set of attributes representing component data,*
- $attrs_{aware}$ *is a set of awareness attributes representing awareness data,*
- $modes$ *is a set of behavioral modes supported by the component type.*

Example: In our search-and-rescue scenario, we just need one self-adaptive component type to represent the system. Its signature is shown in Fig. 11.2 in graphical notation.

The type `Robot` stores its own position and is self-aware of its battery level and whether it is at a victim's position (the position is not awareness data; although it is perceived, it does only transitively trigger self-adaptation when reaching a victim). It is aware of the environment by storing the position of a victim (`null` if unknown), and whether it was requested as recharger by another robot. A robot can switch between five behavioral modes as explained in Sec. 11.3. Although we do not specify the actual behavior of each mode at this stage, we give a short intuition: in `RandomWalk` the robot randomly searches for victims, in `DirectedWalk` it directly goes to a specific location of a victim, in `Rescue` it transports a victim to the rescue area, in `LowBattery` it switches into a low-power behavior and waits for recharging, and in `Recharge` it recharges another robot.

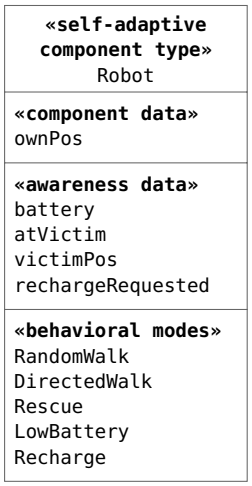


Figure 11.2: Signature of the self-adaptive component type `Robot`

11.4.2 Adaptation Automaton

To describe the rules for switching between behavioral modes, we propose *adaptation automata*. They build on the new notion of hierarchical labeled transition systems (H-LTS) which extend standard LTS by hierarchy and history. Like UML state charts, an H-LTS allows complex states, which are hierarchically composed, and deep history states, which allow to return to the last visited basic sub-state of a complex state.

Auxiliary Definitions: First, we define a *set* Q which represents hierarchically composed complex states. A state from Q is either a basic state which is not hierarchically composed or it is a complex state which contains a set of states from Q and marks one of its states as initial state. More formally, the set Q is defined as follows.

Def. 11.2: Complex States

The set Q *defines a set of complex states over a set* Q_{basic} *of basic states. A state* $q \in Q$ *is either*

- *a basic state* $q \in Q_{basic}$ *or*
- *a complex state* $q = (cset, init)$ *such that* $cset \subseteq Q$ *with* $q \notin cset$ *is a finite, non-empty set of states and* $init \in cset$ *is the initial state of the set.*

We denote the set of all basic states transitively included in a state q by $\textit{basic-states}(q)$ and the set of all sub-states transitively included in a state q by $\textit{sub-states}^*(q)$ such that

$$
\begin{aligned}
&\textit{basic-states}(q) && = q, && \textit{if } q \in Q_{basic}\\
&\textit{basic-states}(q) && = \bigcup_{q' \in cset(q)} \textit{basic-states}(q') && \textit{otherwise}
\end{aligned}
$$

and

$$
\begin{aligned}
&\textit{sub-states}^*(q) && = q, && \textit{if } q \in Q_{basic}\\
&\textit{sub-states}^*(q) && = cset(q) \cup \bigcup_{q' \in cset(q)} \textit{sub-states}^*(q') && \textit{otherwise.}
\end{aligned}
$$

A state $q \in Q$ is *well-formed* if

- it is a basic state $q \in Q_{basic}$ or
- it is a complex state $q = (cset, init)$ such that all $q' \in cset$ are well-formed and all sets $basic\text{-}states(q')$ are disjoint.

Hierarchical Labeled Transition System: Based on these auxiliary definitions, we can define *hierarchical labeled transition systems (H-LTS)*. As standard labeled transition systems (LTS), an H-LTS has states and allows labeled transitions between states. An H-LTS extends an LTS in two ways: An H-LTS allows to hierarchically compose states to complex states as introduced in Def. 11.2 and to define transitions between basic states and/or complex states. Furthermore, an H-LTS introduces history states which represent that we return to the last visited basic sub-state of a complex state.

Def. 11.3: Hierarchical Labeled Transition System

A hierarchical labeled transition system over a set Q_{basic} of basic states *is a tuple* (q, L, δ, δ^*) *such that*

- q *is a state from the set of (well-formed) states over* Q_{basic}*,*
- L *is a set of labels,*
- $\delta \subseteq sub\text{-}states^*(q) \times L \times sub\text{-}states^*(q)$ *is a transition relation, and*
- $\delta^* \subseteq sub\text{-}states^*(q) \times L \times sub\text{-}states^*(q)$ *is a history transition relation.*

The H-LTS is composed of only one state from the complex states over Q_{basic}. We call it the *core state* of the H-LTS. Labeled transitions connect two sub-states (of any depth) of this state. Semantically, an H-LTS can be represented as a standard LTS. Transitions originating from a complex state are an abbreviation for adding a transition with the same target to every basic sub-state of the origin. Transitions leading to a complex state actually target the initial state of the complex state. A separate set of labeled history transitions reflects the idea of deep history states from UML state charts. A history transition leading to a complex state q means returning to the last visited basic state $q' \in basic\text{-}states(q)$.

Adaptation Automaton: An *adaptation automaton for a self-adaptive component type* is a special instance of an H-LTS. The basic states range over all different behavioral modes of the component type. Transitions between states are initiated by predicates over awareness data of the component type. Thus, the adaptation automaton describes the rules when the corresponding component switches between behavioral modes.

Def. 11.4: Adaptation Automaton

An adaptation automaton over the signature $ct = (nm, attrs, attrs_{aware}, modes)$ of a self-adaptive component type *is an H-LTS* $AA_{ct} = (q, L, \delta, \delta^*)$ *over* $modes(ct)$ *such that* L *is a set of predicates over* $attrs_{aware}(ct)$.

Example: In Fig. 11.3, the adaptation automaton of a robot in the search-and-rescue scenario is shown in a graphical notation similar to UML state charts. The initial state of each complex state is indicated by an initial pseudo-state, transitions are labeled with the triggering predicate as event, and history transitions are denoted by leading

to a pseudo-state marked H*. We just highlight complex states and history transitions. The two search strategies `RandomWalk` or `DirectedWalk` are integrated into one complex state `Search` of searching for victims. Therefore, we are able to express that independently from the executed search strategy, the robot switches to rescuing as soon as it is at a victim's position (cf. transition from `Search` to `Rescue`). Similarly, the robot interrupts its current behavior if itself goes out of battery (cf. transition from `FullBattery` to `LowBattery`). However, it resumes the previously executed behavior upon recharge expressed by the history transitions from `LowBattery` back to the history state of `FullBattery`.

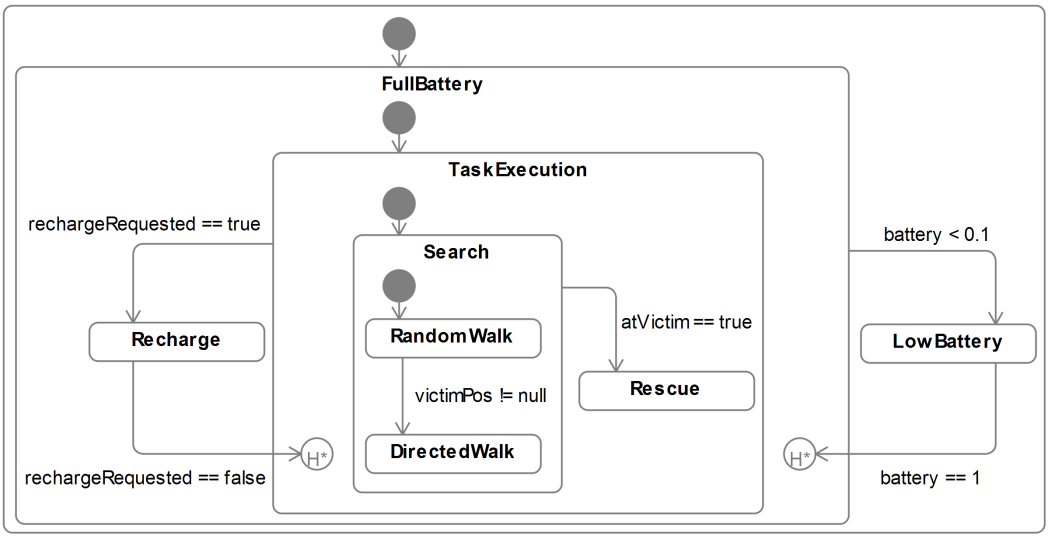


Figure 11.3: Adaptation automaton for self-adaptive component type `Robot`

Adaptation Specification: An *adaptation specification* consists of a set of self-adaptive component types and their adaptation automata. The components may interact when executing their behavioral modes, but this interaction is not specified here since the interaction is part of application logic and not of adaptation logic.

Def. 11.5: Adaptation Specification

An adaptation specification *is a pair* $AdapSpec = (sigs, auts)$ *such that* $sigs$ *is a set of self-adaptive component types and* $auts$ *is a set of adaptation automata such that for every self-adaptive component type* $ct \in sigs$ *there exists exactly one adaptation automaton* $AA_{ct} \in auts$.

Example: For our search-and-rescue scenario, the specification of the system consists only of the self-adaptive component type `Robot` and the corresponding adaptation automaton in Fig. 11.3. We might employ several robots in our final system, but do not envision any other types of components here.

11.4.3 Benefits

Conceptually, the adaptation specification realizes the idea of separation of concerns. The transitions of an adaptation automaton are triggered by changes of awareness data

and therefore capture only the adaptation logic of the component. The states of an adaptation automaton are the different behavioral modes of a component and serve as placeholders where concrete application logic can be plugged in later on. Thus, we do not take care of the actual behavior in each mode since we concentrate on the adaptation logic only.

Methodologically, the adaptation specification benefits from the introduction of hierarchical states and history transitions which has not been proposed in the literature of SAS so far. Hierarchical states allow subsuming transitions with the same trigger from all sub-states of a complex state into one single transition. History transitions represent returning to the last visited basic sub-state of a complex state and therefore prevent unfolding the adaptation automaton for every possible last visited basic sub-state. Due to these features, adaptation rules are specified more compactly than in a standard LTS.

11.5 HAM Pattern

In the HELENA development methodology for self-adaptive systems in Fig. 11.1, we rely on the *HAM pattern* to derive a HELENA design model from an adaptation specification. The HAM pattern is an architectural design pattern which proposes to realize self-adaptive systems with a role-based architecture based on the HELENA methodology according to the autonomic manager pattern [PCZ13]. The proposed pattern describes how the system adapts and how the application logic is specified and plugged in which was not part of the initial adaptation specification.

In the following, we first introduce the foundations for our architectural design pattern, the *autonomic manager pattern*. Afterwards, we present the HAM pattern and elaborate on the benefits of realizing an adaptation specification with this pattern.

11.5.1 Autonomic Manager Pattern

To develop an architectural design for self-adaptive systems, we take inspiration from the *autonomic manager pattern* [PCZ13, CDP+13]. In this pattern (cf. Fig. 11.4), an adaptable component is managed by an adaptation manager (in the original pattern autonomic manager). The manager implements an external feedback loop for the component monitoring the environment and the component itself, analyzing and planning appropriate reactions, and executing adaptations on the managed component as proposed in IBM's blueprint for the MAPE-K loop [IBM06]. Thereby, the component takes perceptions about the environment with its sensor. It forwards these perceptions together with observations about its own state via its emitter to the manager. Therefore, the adaptation manager observes the state of the component and transitively of the environment via its sensor. Depending on these perceptions, it internally decides about appropriate reactions. It imposes the decisions via its effector on the controller port of the component. The component realizes the instructed adaptations affecting the environment through its effectors.

The autonomic manager pattern advocates separation of concerns by externalizing the adaptation manager which imposes its decisions on the component. Additionally, the pattern suggests how the system adapts. Sensors perceive the environment and the component itself, decisions are made based on the perceptions, and the component just performs the desired adaptations without any influence on the adaptation decision.

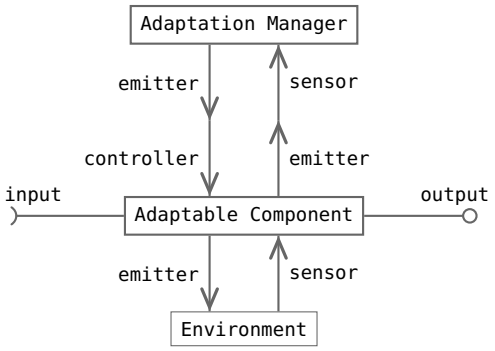


Figure 11.4: Autonomic manager pattern [CDP+13]

However, the pattern lacks a concept how the component actually changes its behavior. We think that roles which can be played by components are an intuitive representation of context-specific behavior. They naturally resemble behavioral modes and can therefore provide the necessary concept for adapting a component's behavior. Behavioral adaptation in the context of roles is easily performed by switching the currently active role of a component since each role encapsulates application logic representing a behavioral mode.

11.5.2 Role-Based Extension of the Autonomic Manager Pattern

To provide a guideline to engineers how self-adaptive systems can be realized, we propose the HELENA Adaptation Manager (HAM) pattern (cf. Fig. 11.5). It reuses the ideas from the autonomic manager pattern in Fig. 11.4 to separate monitoring and adaptation logic from the adaptable component, but it augments it by the concept of roles between which the component switches to realize the instructed adaptations. Also, the adaptation manager and sensors to perceive the environment are realized as roles.

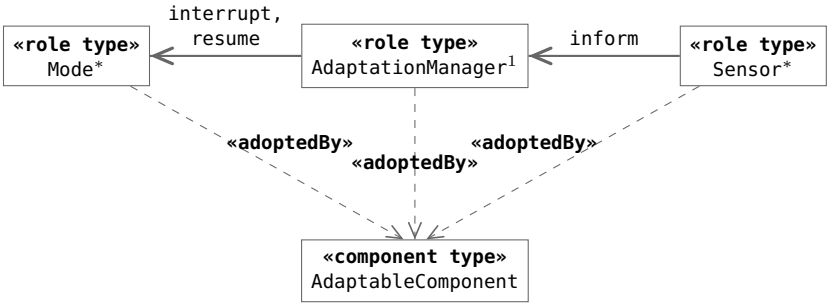


Figure 11.5: HELENA Adaptation Manager (HAM) pattern

Adaptable Component: The central entity of the HAM pattern is the adaptable component. This is the entity in the system which needs to be adapted and should perform the desired adaptive behavior. However, since we apply the idea of roles, the component itself is not active. It is just a data container and serves as the execution platform for roles which are the active entities providing context-specific behavior.

Behavioral Modes: Opposed to the original autonomic manager pattern, we provide a concept for realizing instructed adaptations. A component adapts its behavior by switching between roles representing behavioral modes. The abstract role type `Mode` in

the pattern in Fig. 11.5 is instantiated by concrete mode role types for every behavioral mode the component can execute when the pattern is applied. Although there can be many different mode role types (indicated by *) which can be played by the adaptable component (indicated by the dependency between `Mode` and `AdaptableComponent`), the component should only adopt *one mode role* at a time[2]. The adaptation manager has to take care of deactivating the mode roles which are not adequate anymore by sending the message `interrupt` to them and activating an appropriate mode role by sending the message `resume` to it (indicated by the arrows from `AdaptationManager` to `Mode`). When a mode role is activated, it is responsible for executing its associated role behavior. Like that, the behavior in each mode is encapsulated in a separate role behavior specification and gets activated as soon as the corresponding mode role is adopted by the adaptable component.

Sensors: To decide which mode role should be activated, the component has to monitor its awareness data. We externalize monitoring to sensors represented by roles again. The abstract role type `Sensor` in Fig. 11.5 is instantiated by concrete sensor types for every item of awareness data[3]. There are many different sensor roles (indicated by *) similarly to mode roles, but in contrast to mode roles the component needs to adopt *all sensor roles* in parallel to monitor all items of awareness data. Each sensor is responsible to continuously inform the adaptation manager about the value of its monitored awareness data item by sending the message `inform`.

Adaptation Manager: The `AdaptationManager` is yet another role executed by the adaptable component. It is responsible for realizing the actual adaptation logic. That means, it continuously receives the sensor data via `inform` messages, internally decides how to react to these perceptions, and switches the currently active mode role of the underlying adaptable component by sending `interrupt` and `resume` messages.

Example: Let us illustrate at our search-and-rescue example how we can apply the HAM pattern to derive a role-based HELENA model (cf. Fig. 11.6) for the self-adaptive component type `Robot` in Fig. 11.2.:

- The central entity in the design model is the robot itself. It is a component type, i.e., it is just the resource for executing the different behavioral modes.
- Each behavioral mode in the specification is represented by a new role. For example, the robot is able to execute a behavioral mode like randomly searching for a victim by adopting the corresponding role `RandomWalk`.
- For every item of awareness data in the specification, the corresponding sensor is represented by a new role, e.g., the sensor role `BatterySensor` for the awareness attribute `battery`. Thus, the robot is able to monitor each item of awareness data by adopting all sensor roles at the same time.
- Finally, the adaptation manager is installed as a role on top of the robot[4].

[2]Note that it is not part of the pattern how the behaviors of all introduced roles can be derived. However, in Sec. 11.7 we explain how the specifier adds behaviors for mode roles and how all other role behaviors can systematically be derived from an adaptation specification (cf. Sec. 11.4).

[3]See footnote 2.

[4]Note again that it is not part of the pattern how the behaviors of all introduced roles can be derived from an adaptation specification. However, in Sec. 11.7 we will explain how the specifier adds behaviors for mode roles and how all other role behaviors can systemically be derived from an adaptation specification.

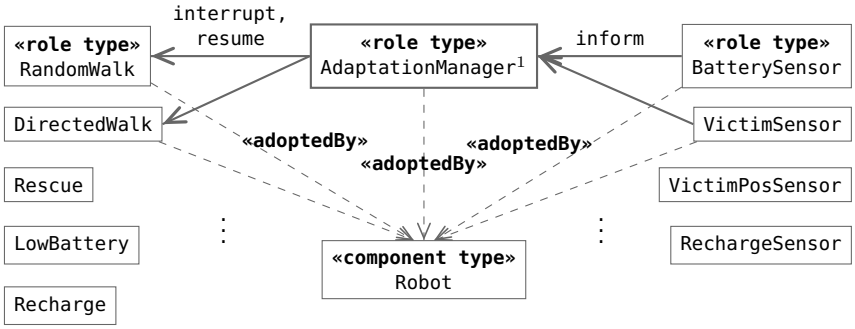


Figure 11.6: Applying the HAM pattern to the search-and-rescue scenario

11.5.3 Benefits

A design model following the HAM pattern benefits from the underlying concepts. The proposed architecture respects separation of concerns as advocated in the autonomic manager pattern. Roles intuitively express the different tasks of self-adaptive components, like being aware of the environment, managing adaptation, and performing particular behavioral modes. The original autonomic manager pattern is furthermore augmented with a concept for adapting behavior by switching between different modes represented by roles. We can automatically derive a formal HELENA model from an adaptation specification following the HAM pattern and equip it with application logic for each behavioral mode role. Table 11.6 and Sec. 11.7 will focus on the formal derivation. Relying on the formal foundation of HELENA allows to analyze the derived model for communication errors and goal satisfaction as explained in Chap. 4, Chap. 5 and Sec. 8.3 and showcased in Chap. 10. With HELENATEXT (cf. Sec. 8.4) and jHELENA (cf. Chap. 7), it is also possible to automatically generate a Java implementation for the HELENA model and execute it.

11.6 Model Transformation Part 1 – Derivation of a Role-Based Architecture

This section describes the first model transformation in the HELENA development methodology for self-adaptive systems in Fig. 11.1. It systematically derives the role-based architecture proposed in the HAM pattern from a given adaptation specification. Afterwards, a second model transformation (cf. Sec. 11.7) will derive the dynamic behavior for all entities in the architecture.

11.6.1 Input Artifacts

The first model transformation starts from the adaptation specification of self-adaptive component types described in Sec. 11.4. It consists of the signature of each self-adaptive component type – describing the component's (normal) data, awareness data, and behavioral modes – and the corresponding adaptation automaton – describing the adaptation logic of the component. The application logic which the component executes is not part of the adaptation specification, but the adaptation automaton provides placeholders for it.

11.6.2 Output Artifacts

From the adaptation specification, a role-based architecture in HELENA (cf. Sec. 11.5) is derived which follows the HAM pattern. It consists of the component with attributes and operations as well as roles for behavioral modes, sensors, and the adaptation manager. However, to gain a full HELENA model, the role-based architecture has to be extended by behaviors for all roles which is pursued by a second model transformation (cf. Sec. 11.7).

11.6.3 Model Transformation

We assume given an adaptation specification $AdapSpec = (sigs, auts)$. In the following, we summarize the systematic derivation of a role-based architecture in HELENA for the signature $ct = (nm, attrs, attrs_{aware}, modes)$ of a single self-adaptive component type from the set $sigs$ and its corresponding adaptation automaton $AA_{ct} \in auts$. To derive the architecture for the whole adaptation specification, each signature and adaptation automaton in the specification needs to be handled as described.

For each $ct \in sigs$ together with its adaptation automaton $AA_{ct} \in auts$, four parts need to be derived: the component's representation in HELENA, its behavioral modes, sensors, and the adaptation manager. We will describe each part separately and illustrate them with our search-and-rescue scenario. Therefore, the following steps show how we derive the architecture depicted in Fig. 11.6 from the signature of the self-adaptive component type `Robot` in Fig. 11.2.

11.6.3.1 Adaptable Component

The signature $ct = (nm, attrs, attrs_{aware}, modes)$ of a self-adaptive component type is represented in HELENA as a component type

$$ct' = (nm(ct), attrs(ct) \cup attrs_{aware}(ct), \emptyset, update\text{-}ops).$$

Apparently, in the HELENA model, we no longer distinguish between (normal) data and awareness data attributes. However for each item of awareness data, we add a special update-operation to the set *update-ops* of component operations. They will later on be responsible for updating the values of the awareness data attributes with the current perceptions about the state of the component or the environment (note that we exploit the fact that the execution of an operation in HELENA can have side-effects on the owning component instance). Their implementation must be provided by an implementor. For the design model, we only declare a set of update operations which is given by $update\text{-}ops = \{updateAttr | Attr \in attrs_{aware}(ct)\}$. The third part of the created HELENA component type represents component associations and is therefore here empty.

Example: For the adaptation specification of our search-and-rescue scenario in Fig. 11.2, we create only one component type

$$\begin{aligned} Robot' = (&Robot,\\ &\{ownPos, battery, atVictim, victimPos, rechargeReq\},\\ &\emptyset,\\ &\{updateB, updateAV, updateVP, updateRR\}). \end{aligned}$$

It represents the self-adaptive component type `Robot` in Fig. 11.2. All denoted operations are the update-operations of the awareness data attributes, e.g., *updateB* is

the update-operation for the awareness data attribute *battery*. The specification of the component type in HELENATEXT can be found in Appendix E.

11.6.3.2 Behavioral Modes

For each behavioral mode $mode \in modes(ct)$, a new role type

$$mode = (mode, \{ct'\}, \emptyset, \emptyset, \{interrupt, resume\})$$

is created. It can only be adopted by the HELENA representation ct' of the self-adaptive component type ct offering the behavioral mode *mode*. The new mode role type does not have any attributes or outgoing messages. It can only receive the messages *interrupt* and *resume* to allow the adaptation manager to deactivate and activate the role later on[5].

Example: For our search-and-rescue scenario, we create the five mode role types depicted in Fig. 11.6. They are all equipped with the messages *interrupt* and *resume* as incoming messages. For example, for the behavioral mode `RandomWalk`, the new mode role type

$$RandomWalk = (RandomWalk, \{Robot'\}, \emptyset, \emptyset, \{interrupt, resume\})$$

is created. It can only be adopted by a component of type *Robot'* which is the HELENA representation of the self-adaptive component type `Robot` offering the behavioral mode `RandomWalk`. It does not have any attributes or outgoing messages and is only able to receive the messages *interrupt* and *resume*. The specification of all mode role types in HELENATEXT can be found in Appendix E.

11.6.3.3 Sensors

For each awareness data attribute $attr \in attrs_{aware}(ct)$, we create a sensor role

$$AttrSensor = (AttrSensor, \{ct'\}, \emptyset, \{inform()(attr{:}String, val{:}Object)\}, \emptyset)$$

that can only be adopted by the HELENA representation ct' of the self-adaptive component type ct monitoring the awareness data attribute *attr*. The new sensor role does not have any attributes or incoming message. However, it can send the message *inform()(attr:String,val:Object)* to inform the manager about the value of its monitored awareness data attribute later on. The name of the monitored attribute will be sent in the parameter *attr* and the current value will be sent in the parameter *val*.

Example: For our search-and-rescue scenario, we create the four sensor role types depicted in Fig. 11.6. They are all equipped with the message *inform* as outgoing message. For example, for the awareness data attribute `battery` in our search-and-rescue scenario, a new sensor role

$$BatterySensor = (BatterySensor, \{ct'\}, \emptyset, \{inform()(attr{:}String, val{:}Object)\}, \emptyset)$$

is created. It can only be adopted by a component of type *Robot'* which is the HELENA representation of the self-adaptive component type `Robot` offering the awareness data attribute `battery`. It does not have any attributes or incoming messages and is only able to send the message *inform*. The specification of all sensor roles types in HELENATEXT can be found in Appendix E.

[5] Later on, the application logic of each mode role type will be specified in form of a role behavior. All messages used in this role behavior have to be added to the set of outgoing and incoming message resp. when needed.

11.6.3.4 Adaptation Manager

The adaptation manager itself is yet another role which later on takes care to adapt the component. Formally, the adaptation manager is represented by a role type

$$AM = (AM, \{ct'\}, \emptyset, \{inform()(attr{:}String, val{:}Object)\}, \{interrupt, resume\})$$

which can only be adopted by an instance of the managed component type ct'. The adaptation manager does not provide any attributes, but can send the messages *interrupt* and *resume* to deactivate and activate roles and can receive the message *inform()(attr:String,val:Object)* to receive the perceptions of sensors later on.

Example: In our search-and-rescue scenario, the adaptation manager is given by

$$AM = (AM, \{Robot'\}, \emptyset, \{inform()(attr{:}String, val{:}Object)\}, \{interrupt, resume\}).$$

It can only be adopted by a component of type *Robot'* which is the HELENA representation of the self-adaptive component type `Robot`. It does not have any attributes, but it is able to send the messages *interrupt* and *resume* and to receive the message *inform()(attr:String,val:Object)*. The specification of role type for the adaptation manager in HELENATEXT can be found in Appendix E.

11.7 Model Transformation Part 2 – Derivation of Dynamic Behaviors

In the second model transformation of the HELENA development methodology for self-adaptive systems in Fig. 11.1, each entity of the role-based architecture is equipped with a dynamic behavior.

11.7.1 Input Artifacts

The input for the second model transformation consists of three parts: The first artifact is the role-based architecture which was created during the first model transformation. It results from applying the HAM pattern to the signature of a self-adaptive component type and provides role types for all behavioral modes of the component, sensor role types for all awareness data attributes, and an adaptation manager. However, neither of the role types is equipped with a role behavior so far.

The second artifact is the adaptation automaton for the self-adaptive component type which was already part of the initial adaptation specification. It describes the adaptation logic and must be incorporated into the HELENA model as the role behavior of the adaptation manager.

The third artifact are mode behaviors. Since role types were created for each behavioral mode in the first model transformation, we can now define their application logic in the form of HELENA role behaviors which were not yet part of the initial adaptation specification. By keeping the mode behaviors separated from the adaptation automaton, we respect separation of concerns. As far as the mode behaviors require any operations of the component or messages being exchanged between roles, the specifier has to define them for the corresponding components and roles.

11.7.2 Output Artifacts

The second model transformation completes the role-based architecture which was created from the signature of a self-adaptive component type in the first model transformation with a behavior for each new role type. Therefore, a fully specified HELENA design model is gained which consists of a component type and associated role types for behavioral modes, sensors, and the adaptation manager as well as corresponding role behaviors. The role behaviors are derived such that the component exhibits the desired adaptive behavior as envisioned by the adaptation automaton: The adaptation manager is informed by the sensors about perceptions of the environment and the component itself. It internally decides about appropriate adaptations based on the adaptation automaton and deactivates and activates the corresponding mode roles. The activated mode role takes care to execute its associated application logic. The component itself does not have any behavior, it rather executes its associated roles.

11.7.3 Model Transformation

For each signature $ct = (nm, attrs, attrs_{aware}, modes)$ of a single self-adaptive component type in an adaptation specification $AdapSpec = (sigs, auts)$, we assume given

- the corresponding role-based architecture consisting of
  - a component type ct',
  - a role type $mode$ for each $mode \in modes(ct)$,
  - a role type $AttrSensor$ for each awareness data attribute $attr \in attrs_{aware}(ct)$,
  - a role type AM for the adaptation manager,
- the adaptation automaton AA_{ct} for the self-adaptive component type from the initial adaptation specification $AdapSpec$, and
- a HELENA role behavior for each mode role.

For each $ct \in sigs$ together with its corresponding role-based architecture, its adaptation automaton $AA_{ct} \in auts$, and role behaviors for each mode role, three types of role behaviors need to be derived: the role behavior for each mode role needs to be adapted to allow the adaptation manager to activate and deactivate its execution, the role behavior for each sensor role needs to be created such that it continuously informs the adaptation manager about the value of the monitored awareness data attribute, and the role behavior of the adaptation manager needs to be derived from the adaptation automaton.

11.7.3.1 Behavioral modes

The specified role behavior for each mode needs to be extended such that the adaptation manager is able to control the execution of the behavior.

Firstly, the behavior must initially be paused. It is only started upon request with the message *resume* from the adaptation manager. Therefore, we modify each role behavior **roleBehavior** *mode* to initially wait for an incoming message *resume* sent by the adaptation manager and then execute the original role behavior.

Secondly, the role behavior always needs to be interruptible and resumable to allow the adaptation manager to switch the currently executed behavioral mode represented by a role and its behavior at any time. We simulate interruption by adding the option $?interrupt.?resume$ to every action prefix $a.P$ in the role behavior (the messages

interrupt and *resume* will later on be sent by the adaptation manager). The option ?*interrupt*.?*resume* is prioritized over the other option of the introduced nondeterministic choice. This prioritization is not part of core HELENA and, thus, the semantics of HELENA needs to be adapted for nondeterministic choice with prioritization. If the prioritized branch is currently executable, this branch will be taken independently of whether the other branch is executable. If the prioritized branch is currently not executable, the execution of the nondeterministic choice construct is decided based on the executability of the other branch. Furthermore, in nondeterministic choice with prioritization we now also allow mixing incoming and outgoing messages as first actions of the branches since we prioritize one of the branches anyway. For verification, the prioritization does not have any impact since to satisfy an LTL goal, all traces of the induced Kripke structure of a HELENA ensemble specification have to satisfy the LTL goal. Some of the traces represent the traces where one option of nondeterministic choice is prioritized, i.e., model-checking will check these particular traces amongst other traces. For implementation, we added a class `PrioritizedNondeterministicChoice` to the jHELENA framework. It allows to prioritize one branch over the other such that jHELENA will first try to execute the prioritized branch and only if it is not executable, it will execute the other branch.

With these two steps together, the role behavior **roleBehavior** $mode = P$ is extended to an interruptible and resumable role behavior by

$$\textbf{roleBehavior}\ mode_{interrupt} = ?resume\ .\ mode'\ \text{with}\ \textbf{process}\ mode'\ =\ \sigma(P)$$

and

$$\begin{aligned}
&\sigma(\textbf{quit}) && = \textbf{quit}\\
&\sigma(a.P) && = Q\\
&\quad \textit{such that } Q \textit{ is a fresh name with} && \\
&\quad \textbf{process}\ Q = ?interrupt.?resume.Q + a.\sigma(P) && \\
&\sigma(P_1 + P_2) && = \sigma(P_1) + \sigma(P_2)\\
&\sigma(\textbf{if}\ (guard)\ \{P_1\}\ \textbf{else}\ \{P_2\}) && = \textbf{if}\ (guard)\ \{\sigma(P_1)\}\ \textbf{else}\ \{\sigma(P_2)\}\\
&\sigma(N) && = N'\\
&\quad \textit{for}\ \textbf{process}\ N = P && \\
&\quad \textit{and}\ \textbf{process}\ N' = \sigma(P). &&
\end{aligned}$$

Example: Let us exemplify the modification of a behavior at the mode `RandomWalk` in our search-and-rescue scenario. We assume given the role behavior for that mode with **roleBehavior** $RandomWalk = \textbf{owner}.randomStep\ .\ RandomWalk.$ In this role behavior, the operation *randomStep* takes care to let the robot randomly take a step, i.e., we exploit the fact that operations can have side-effects in HELENA. The role behavior is modified to

$$\begin{aligned}
&\textbf{roleBehavior}\ RandomWalk_{interrupt} && = ?resume.RandomWalk'\\
&\textbf{process}\ RandomWalk' && = Q\\
&\textbf{process}\ Q && = ?interrupt.?resume\ .\ Q\\
& && \quad + \textbf{owner}.randomStep\ .\ RandomWalk'
\end{aligned}$$

The specification of the interruptible role behaviors of all mode role types in HELENATEXT can be found in Appendix E.

11.7.3.2 Sensors

Intuitively, the role behavior of a sensor takes care to continuously advise the owning component to update the current value of the monitored awareness data attribute by calling the corresponding update-operation on the owning component and to transmit the new value to the adaptation manager by sending the message *inform* with appropriate values to it. Formally, the role behavior of a sensor *AttrSensor* is therefore given by

$$
\begin{aligned}
&\textbf{roleBehavior } \mathit{AttrSensor} && = am{\leftarrow}\textbf{create}(AM, \textbf{owner}) \ .\ \mathit{AttrMonitor} \\
&\textbf{process } \mathit{AttrMonitor} && = \textbf{owner}.\mathit{updateAttr} \ . \\
& && \quad am!\mathit{inform}(\text{“}\mathit{attr}\text{”}, \textbf{owner}.\mathit{attr}) \ . \\
& && \quad \mathit{AttrMonitor}
\end{aligned}
$$

Example: Let us consider the awareness data attribute *battery* in our search-and-rescue scenario. The role behavior of the corresponding sensor role *BatterySensor* is given by

$$
\begin{aligned}
&\textbf{roleBehavior } \mathit{BatterySensor} && = am{\leftarrow}\textbf{create}(AM, \textbf{owner}) \ .\ \mathit{BatteryMonitor} \\
&\textbf{process } \mathit{BatteryMonitor} && = \textbf{owner}.\mathit{updateB} \ . \\
& && \quad am!\mathit{inform}(\text{“}\mathit{battery}\text{”}, \textbf{owner}.\mathit{battery}) \ . \\
& && \quad \mathit{BatteryMonitor}
\end{aligned}
$$

The specification of the role behaviors of all sensor role types in HELENATEXT can be found in Appendix E.

11.7.3.3 Adaptation Manager

The adaptation manager is equipped with a role behavior that is responsible for changing the currently active role of the managed component according to the adaptation automaton AA_{ct}. In summary, we translate the adaptation automaton in three steps into a role behavior: (1) flattening the adaptation automaton to a standard labeled transition system, (2) deriving a process term from the labeled transition system, (3) adding initialization of mode roles.

(1) Flattening the Adaptation Automaton: The first step of flattening the adaptation automaton to a standard LTS is rather involved since hierarchy and history needs to be resolved. However, the procedure resembles flattening of UML state charts which is already tackled in the literature. For details, we refer the interested reader to the work of Wasowski [Was04] who is the only author tackling history pseudo-states during flattening according to Devroey et al. [DPC+14]. The basic idea is to reflect all transitions between complex states by their counterparts on basic states. Furthermore, history transitions are resolved by propagating the information about the last visited state to all other states and therefore possibly duplicating transitions according to that information. Thus, several states of the resulting LTS represent the same basic state from the adaptation automaton, each reflecting a different previously visited state.

(2) Derivation of a Process Term: In the next step, we derive a process term from the LTS. The procedure follows the idea of deriving a right linear grammar from a nondeterministic finite automaton [HMU06]. The LTS created during flattening can be considered as a nondeterministic finite automaton (NFA) whose final states are the deadlock states of the LTS. The NFA can be translated, with the classical procedure, into a right linear grammar. Then, the production rules of this grammar can be considered as process declarations such that the rule for the starting symbol provides the process declaration of the role behavior of the adaptation manager.

During deriving the right linear grammar, we consider all labeled transitions of the adaptation automaton. Each such transition represents self-adaptation of the underlying component. For example, the transition $(mode_1, predicate(attr), mode_2)$ means that the component is currently executing the behavioral mode $mode_1$. Whenever the transition is triggered, i.e., the predicate $predicate(attr)$ becomes **true**, the component will change its behavioral mode to $mode_2$.

In HELENA, we represent this change of the executed behavioral mode by a process declaration. The process first waits for an *inform* message from any sensor. If a value was sent making the predicate **true**, self-adaptation is triggered and the adaptation manager interrupts the behavioral mode $mode_1$ and resumes the behavioral mode $mode_2$. If the value did not trigger adaptation, the adaptation manager waits for the next sensor input. Formally, the resulting sequence of HELENA actions is given by

$$\begin{aligned}\textbf{process } s_1 = {} & ?inform(attr, val)\ . \\ & \textbf{if } (predicate(attr))\ \{mode_1!interrupt\ .\ mode_2!resume\ .\ s_2\} \\ & \textbf{else } \{s_1\}\end{aligned}$$

(3) Initialization of Mode Roles: In the last step, we add some initialization actions at the beginning of the behavior of the adaptation manager to initialize all possible mode roles and start the initial role of the self-adaptive component type. To this end, we create every possible mode role *mode* with the statement $modeInst \leftarrow \textbf{create}(mode, \textbf{owner})$. Afterwards, we start the initial role *initInst* from the adaptation automaton – which is the basic mode reached by following the *init* references from the core state of the adaptation automaton – by sending a *resume* message to it with $initInst!resume$ (this role has to be created with one of the actions before). Lastly, the role behavior of the adaptation manager continues by invoking the process term which was generated during the first two steps of flattening and deriving a role behavior for the adaptation manager.

Example: Let us exemplify the result of these three derivation steps by an excerpt of the behavior of the adaptation manager in our search-and-rescue scenario[6].

$$\begin{aligned}\textbf{roleBehavior } AM \quad = {} & rwInst \leftarrow \textbf{create}(RandomWalk, \textbf{owner})\ . \\ & \ldots \\ & lbInst \leftarrow \textbf{create}(LowBattery, \textbf{owner})\ . \\ & rwInst!resume\ . \\ & RandomWalkProcess\end{aligned}$$

[6] We use the notation **else** in the usual meaning as an abbreviation.

```
process RandomWalkProcess      = ?inform(attr,val) .
                                 if (attr=="battery" & val<0.1) {
                                   rwInst!interrupt .
                                   lbInst!resume .
                                   LBFromRWProcess
                                 }
                                 ...
                                 else {RandomWalkProcess}
process LBFromRWProcess        = ?inform(attr,val) .
                                 if (attr=="battery" & val==1) }
                                   lbInst!interrupt .
                                   rwInst!resume .
                                   RandomWalkProcess
                                 }
                                 elsif ...
```

First, new instances for each of the five behavioral modes are created, e.g., the role instance *rwInst* for the behavioral mode `RandomWalk`. Then, the role behavior of the *RandomWalk* role is started by sending the message *resume* to it. This is the initial behavioral mode in the adaptation automaton in Fig. 11.3 and therefore has initially to be executed.

Afterwards, According to the adaptation automaton, the adaptation manager has to change the executed behavioral mode from `RandomWalk` to another behavioral mode if one of four transitions is triggered: Either it changes its behavioral mode to `DirectedWalk` if the underlying component got informed about a victim's position (trigger predicate `victimPos != null`). Or it changes its behavioral mode to `Rescue` if the underlying component found a victim by itself (trigger predicate `atVictim == true`). Or it changes its behavioral mode to `Recharge` if the underlying component was requested as a recharger by another robot (trigger predicate `rechargeRequested == true`). Or it changes its behavioral mode to `LowBattery` if the underlying component runs out of battery (trigger predicate `battery < 0.1`). In the role behavior of the adaptation manager depicted above, we only show the transition to the low battery mode. The behavior shows that when the battery is low, the adaptation manager interrupts the role instance *rwInst* representing randomly searching for a victim and resumes the role instance *lbInst* representing the low battery mode. Afterwards, the manager continues its role behavior in a state where it knows that the currently active role is *lbInst* and the previously active role was *rwInst* represented by *LBFromRWProcess*. This information is needed since if the robot was recharged, it has to resume the previously active role (see history transition from `LowBattery` to the complex state `TaskExecution` in Fig. 11.3). The complete specification of the role behavior of the adaptation manager in HelenaText can be found in Appendix E.

11.8 Related Work

In this section, we present related work from different areas. We first discuss how our approach relates to prominent adaptation reference models. We then compare our adaptation specification to other automata-based specification techniques in the field of

self-adaptive systems. Lastly, we elaborate on how the notion of roles is used to provide an architecture for self-adaptive systems. For all approaches, we evaluate the support of a systematic development process as we propose it.

11.8.1 Adaptation Reference Models

FORMS [WMA12] is the most prominent reference model for self-adaptive systems. According to it, the distinguishing characteristics of SAS is the capability of reflection about itself. In FORMS, the base layer of a system executes the basic behavior as simple reactions to the environment. One level higher in the reflective layer, the system reasons about whether the basic behavior is adequate in the current environment and (possibly) adapts it accordingly. For this purpose, the reflective layer retains a model of the base layer. While the kind of reflection model is not fixed in FORMS, in architecture-based self-adaptation approaches like [Gat98, OGT+99, GCH+04, KM07, KM09] the reflection model corresponds to a representation of the system's architecture. Reflection then reasons about architectural reconfiguration to meet the adaptation requirements.

Our methodology is aligned with the idea of reflection in FORMS since the adaptation manager decides on the adequacy of the currently executed behavioral mode separately from the application logic. Furthermore, we consider our methodology as an architecture-based self-adaptation approach like [Gat98, OGT+99, GCH+04, KM07, KM09] since the adaptation manager reflects on the architecture of the self-adaptive component in the sense of currently executed behavioral mode. In contrast, we propose a role-based architecture for switching between behavioral modes to adapt a self-adaptive component. Basing our methodology on HELENA also provides a basis for formal reasoning about adaptation as presented in FORMS.

11.8.2 Adaptation Specification Techniques

An interesting requirements specification technique for adaptive systems is presented by Luckey and Engels [LE13]. They specify adaptation logic in adapt cases (similarly to use cases) on top of a system's architecture. The operationalization is based on UML activity diagrams and can be checked against quality properties formulated as temporal logic formulae. Opposed to them, we specify adaptation logic similarly to UML state charts since they provide hierarchically composed states and history states. Furthermore, we propose a specific concept how to adapt a system by changing mode role while adapt cases have to call user-defined operations to reconfigure the system. For implementation, the authors do not introduce any first-class concepts to realize adapt-cases while we propose a specific role-based design to transfer the separation of adaptation logic and application logic to implementation.

Automata-based approaches [ZC06, ZMLL11, MPT12, BCG+13] specify adaptation by evolution of finite state machines representing behavioral modes. Zhang et al. [ZC06, ZGC09] specify behavioral modes as finite state machines called steady-state programs. An n-plex adaptive program represents adaptation by transitions between steady-state programs. Zhao et al. [ZMLL11] also represent adaptive programs as finite state machines, but use mode automata to describe evolutions between them. In S[B]-systems [MPT12], stable regions of a state machine represent behavioral modes and special transitions between these regions characterize adaptation. Adaptable interface automata [BCG+13] reuse the idea of special transitions to change some kind of control data resulting in adaptation.

We augment these approaches by H-LTS with history and hierarchy which allow to specify adaptation rules more compactly. Additionally, we can systematically derive verification and implementation models due to relying on HELENA.

11.8.3 Role-Based Adaptation

As Haesevoets et al. [HWH14] state, "roles are [...] recognized as an important modeling concept [...]. Nevertheless, [they have] not received the attention [they] deserve." However, we consider roles as a convenient concept to describe different behaviors dynamically assigned to components. Therefore, self-adaptation can easily be expressed by switching roles.

Steegmans et al. [SWHB05] propose a design process for adaptive behaviors of agents based on roles. They realize the role model by free-flow trees and a corresponding framework. Opposed to our automatic model-driven realization, they do not transfer the concept of roles to the implementation which we consider beneficial to preserve a clean architecture.

The framework Self-Epsilon [MT11] proposes to employ a controller on each object in a self-adaptive system. The controller takes care to change the current role of the object depending on the current context. While conceptually similar to our approach, Self-Epsilon does not aim at providing a formal role-based model for reasoning or at explicitly describing the architecture of such systems. The focus of Self-Epsilon is to support programming role-based models by a domain-specific extension of Java.

11.9 Publication History

The content of this chapter is based on [Kla15a]. Compared to this publication, both model transformations are described in more detail and the robotic search-and-rescue scenario is completely presented in the appendix Appendix E.

11.10 Present Achievements and Future Perspectives

Present Achievements: This chapter explained how the HELENA development methodology can be extended to a holistic development process to engineer self-adaptive systems separating adaptation logic from application logic. The key concept is to model behavioral modes and switching between modes separately. We introduced adaptation automata as a rich specification technique of adaptation logic providing placeholders for the application logic. Furthermore, we proposed to realize a self-adaptive system by a role-based architecture in HELENA and provided the HELENA Adaptation Manager (HAM) pattern as a guideline for the design. By relying on HELENA, the resulting HELENA model can then be analyzed with Spin for goal satisfaction and executed with the Java framework jHELENA.

Discussion: The HELENA development methodology for self-adaptive systems is a holistic methodology providing systematic transitions and therefore easy traceability of artifacts between all development phases.

During specification, adaptation logic is compactly specified by adaptation automata which provide hierarchy and history of states. Though powerful, an H-LTS is more complex than a standard LTS. Thus, a graphical representation (cf. Fig. 11.3) similarly to UML state charts would be helpful for visualization.

We initially specify the adaptation logic by adaptation automata and provide placeholders for plugging in the application logic of modes later on. Currently, mode behaviors can only be specified after deriving a role-based architecture. Therefore, our specification technique could be extended such that application logic can be defined independently from this architecture. For example, one could imagine that the application logic of all behavioral modes can be defined as basic building blocks for the adaptive component in parallel to the adaptation logic defined in the adaptation automaton switching between those behavioral modes.

Furthermore, roles are an intuitive concept for encapsulating context-specific behavior. Behavioral change is easily achieved by changing the currently active role. However, roles cannot share behavior, e.g., obstacle avoidance in our robotic search-and-rescue scenario. Therefore, it would be useful to introduce hierarchy of roles similar to the subsumption architecture [Bro86] or even to allow concurrent execution of several mode roles.

Relying on HELENA allows to use its verification and implementation tools (cf. Chap. 4, Chap. 5, and Sec. 8.3 as well as Chap. 7 and Sec. 8.4). However, although the transition to a verification model for the model-checker Spin is fully automated, the specification of goals for individual modes or the whole self-adaptive component is not yet an integral part of our proposed process. Likewise, the provided jHELENA framework is so far only a proof of concept of how to realize roles in Java. However, we already showed with our case study of the Science Cloud Platform in Chap. 10 that its implementation can be ported to real life platforms and the HELENA abstractions facilitate the implementation of larger software systems.

In summary, the HELENA development methodology for self-adaptive systems serves as a comprehensive and coherent methodology to develop self-adaptive systems and provides a systematic guideline to consistently engineer such systems.

Future Perspectives: Apart from the extensions for the individual phases of the development methodology, we see different topics of interest for future work:

Adaptation of Adaptation Strategy: In our proposed methodology, the adaptation logic of the self-adaptive component is explicitly defined in the adaptation automaton. However, it might be necessary to adapt the strategy of adaptation at runtime. Therefore, one could think about hierarchically composing adaptation managers on top of each other. A similar approach was presented in ActivFORMS [IW14] where a high-level goal management layer takes care to adapt the active model representing the current adaptation manager and its behavior. Likewise, in PSCEL [MPT13], a policy automaton allows to change the currently active policy, and therefore the adaptation strategy, based on conditions about the system's state.

Generation of Adaptation Logic: Beyond the specification of hierarchically composed adaptation managers even techniques from artificial intelligence could be used to generate adaptation logic like given by an adaptation automaton from a domain specification or observations.

Collective Adaptation: We also consider collective adaptation as a very interesting extension of our work. Cabri et al. [PCL14] propose adaptation patterns to capture possible collaboration forms in cooperative scenarios. In the field of service-oriented computing, Foster et al. [FUKM07, Fos09] apply the idea of architectural

modes to a set of services which have to interact to complete a given task. They use service mode behaviors to specify the adaptation logic when to switch the modes of the whole set of collaborating services. So far, our approach only allows to adapt single component types. Different self-adaptive component types might interact when executing their application logic, but they cannot synchronize their adaptation logic. We consider it very valuable to integrate them into our HELENA approach for adaptation.

Chapter 12

Conclusion

This thesis considered the whole development lifecycle of ensemble-based systems. These systems consist of a large number of independent components which dynamically form goal-oriented communication groups called ensembles. Not all components of the system join such a temporary collaboration; only a subset of the overall system contributes the required functionalities to the ensemble. We addressed the particular characteristics of ensemble-based systems with the HELENA development methodology shown in Fig. 12.1.

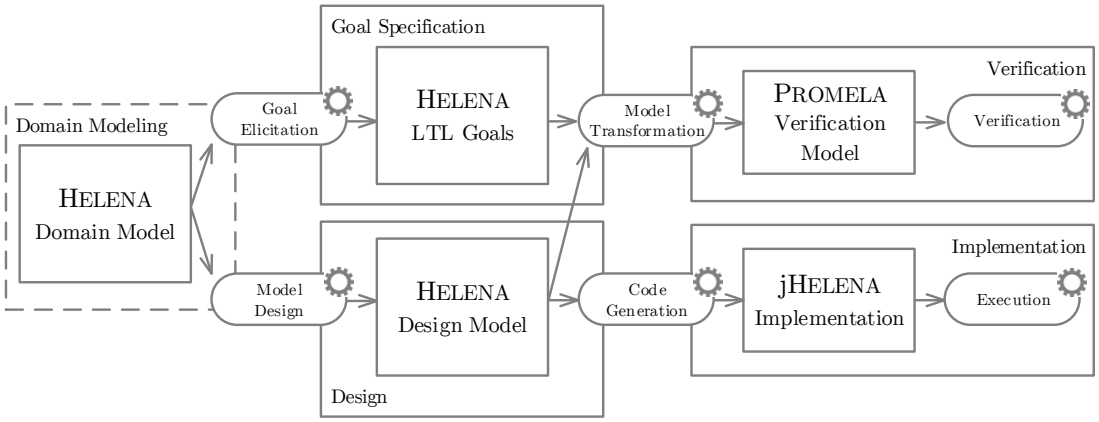


Figure 12.1: The HELENA development methodology for ensemble-based

Its key idea has been to split an ensemble-based system into two dimensions: On the one hand, each ensemble in the system was considered independently from other ensembles, but on top of the same component-based platform. Each ensemble focused on the description and execution of its participants only and did not consider the participation of its members in other ensembles or the concurrent execution of independent ensembles. On the other hand, each participant of an ensemble was described by a role, separately from the particular component executing the role. Roles, featuring the goal-directed behavior of participants, were therefore acting as the active entities in an ensemble. Components were only passive containers which provided the technical functionalities to adopt and execute roles.

The HELENA development methodology was proposed as a holistic model-driven engineering process for ensemble-based systems based on these two principles. It introduced techniques and tools for all main development phases: domain modeling, goal specification, design, verification, and implementation. All techniques consistently employed ensembles and roles as first-class concepts on top of a component-based platform.

With the HELENA workbench, we provided tool support for all activities of the development methodology and allowed automated transitions between the phases. We showed the applicability of the HELENA development methodology and its tools to a larger software system, the Science Cloud Platform. Finally, and in order to make further progress towards adaptive ensembles, we extended our methodology to allow adaptation of components by changing their currently active role based on perceptions of the environment.

12.1 Contributions

In more detail, this thesis contributed the following scientific results:

The Modeling Approach of HELENA*: Syntax and Semantics:* The HELENA development methodology was centered around a formal modeling approach for ensemble-based systems. The structural relationships between participants of an ensemble was described by an ensemble structure of collaborating roles. The interaction of goal-directed behaviors of such roles determined the dynamic behavior of the complete ensemble. For the description of role behaviors, a dedicated process algebra was proposed which introduced particular actions for role creation and retrieval, (synchronous or asynchronous) message exchange between roles, and component access. The evolution of ensembles according to the contributing roles was formalized through a two-level transition semantics. The first level defined the evolution of a single role instance while the second level built upon the first to evolve several role instances in collaboration.

Goal Specification and Verification for HELENA *Models:* HELENA LTL was introduced as a logic to describe temporal goals for ensembles. Dedicated atomic propositions reasoned about the current state of roles and components and were composed to complex HELENA LTL formulae with propositional and temporal operators. To verify that a HELENA ensemble specification actually achieved its goals expressed as HELENA LTL formulae, we proposed to translate HELENA to PROMELA and to check the resulting PROMELA verification model with the model-checker Spin. The translation was proven semantically correct for a kernel part of HELENA such that model-checking results with Spin could be transferred back to HELENA.

Implementing HELENA *Models with Object-Orientation:* To execute HELENA ensembles, we provided the Java framework jHELENA. It realized all HELENA modeling concepts and their semantics in Java by introducing two layers: The `metadata` layer allowed to define the meta model of ensemble specifications in terms of component types and ensemble structures. The `developer interface` provided the basic functionality to realize an actual ensemble-based application and implements the execution semantics of HELENA.

Tool Support for HELENA*:* All proposed techniques were united into a single development environment, the HELENA workbench. At its core, the HELENA workbench supported the domain-specific language HELENATEXT to specify ensemble-based applications according to the formal modeling approach of HELENA. HELENATEXT was integrated into a fully-fledged Eclipse editor with syntax highlighting, content assistance and code validation. Furthermore, two automated code generators instantly translated ensemble specifications to PROMELA for model-checking with Spin and to Java for execution with jHELENA.

Case Study: The Science Cloud Platform served as a case study of a larger software system completely developed with the HELENA development methodology. It showed HELENA's general applicability to rigorously describe collaborations on top of a component-based platform. Verification ensured the elimination of collaboration mismatches in early stages while the introduction of ensembles and roles improved the clarity of the implementation. However, it became apparent that the state space soon exploded during verification and that special effort had to be taken to make the system robust against failure of components and message loss.

Modeling Awareness and Adaptation with HELENA*:* Finally, we demonstrated the potential of the role-based modeling approach of HELENA towards the development of adaptive (collective) ensembles. We introduced adaptation automata to specify adaptation logic independently from application logic. Furthermore, we proposed the architectural HELENA Adaptation Manager pattern which realizes a self-adaptive system by a role-based architecture and provides a guideline for the design of such systems. The pattern relied on the HELENA concepts of roles to describe behavioral modes of a self-adaptive component between which it could change to react to perceptions in the environment. Apart from the intuitive usage of roles to represent behavioral modes, relying on HELENA also allowed to benefit from its verification and implementation techniques.

12.2 Challenges of Ensemble-Based Systems Revisited

With the proposed HELENA development methodology, we addressed the challenges of ensemble-based systems discussed in Sec. 1.1. The following discussion outlines which challenges were completely covered; open issues will be presented together with future work in the next section.

Concurrency: Ensembles had to manage concurrency on two levels, for the different participants of a single ensemble and for a single component contributing to different ensembles. HELENA proposed to distribute work within a single ensemble between several collaborating roles which were executed simultaneously on the same or different components. In principle, it also allowed a single component to participate in different ensembles by concurrently adopting several roles.

Heterogeneity: Ensembles were formed on top of a heterogeneous set of components. Their heterogeneity should guide the assignment of work in an ensemble according to their individual properties. HELENA proposed to define for each role type contributing to an ensemble which component types can adopt it. The component types could thereby be heterogeneous in the capabilities which they offer, but despite their heterogeneity they all provided the necessary capabilities to adopt the particular role type. Consequently, in the running ensemble, the owner of a role could be chosen from a heterogeneous set of components typed by these heterogeneous component types.

Extensibility: Ensembles had to be extensible on two levels, for joining components to the overall system and for joining members to an ensemble. HELENA cannot cope with new component instances in the status presented in this thesis; the number of component instances cannot change during the execution of ensembles. However, on the level of ensembles, HELENA allows to dynamically change the number

of participants. We introduced a dedicated action to create new role instances participating in the ensemble and allowed roles to terminate their execution if they finished their behavior.

Dynamism: Ensembles should dynamically establish new communication links between participants of the collaboration and dynamically assign additional tasks to members of the group. HELENA allowed for the establishment of new communication links by exchanging references of role instances via message exchange. Those references could then be used for communication with the referenced role instance. The possibility to create new role instances on demand provided the basis for assigning an additional task to a participant of the ensemble by letting the participant adopt the newly created role instance.

Transparency: Ensembles should distribute work among their participants transparently from the concrete components taking care of the different subtasks. HELENA provided this kind of transparency by distinguishing between roles and components. At design time, ensembles were specified in terms of roles on top of the components which later on adopted the roles.

Goal-Orientation: Ensembles collaborated towards a global goal. HELENA guaranteed that an ensemble always worked towards its intended goal by verification. Goals were specified with HELENA LTL and checked with the model-checker Spin for the specified ensembles.

Autonomy: Ensembles should organize themselves without any global supervision. HELENA allowed ensembles to autonomously guide their composition. The initial state of an ensemble defined the roles which initially formed the collaboration. By using the special ensemble-based actions for role creation and retrieval, these initial roles could then dynamically and autonomously extend the ensemble by new members. Thus, HELENA did not have to employ any kind of global coordinator which supervised the composition of the ensemble.

12.3 Future Work

Some of the challenges of ensemble-based systems were not fully addressed in the current status of the HELENA development methodology. Furthermore, working with HELENA gave rise to new ideas about how to further improve it. We already discussed future work in detail for each of the phases of the HELENA development methodology throughout this thesis. Thus, this final section only summarizes the most important aspects for future work for each of our contributions and gives some outlooks for increasing the expressibility and suitability of the complete HELENA development methodology.

12.3.1 The Modeling Approach of HELENA

The main area of improvement is to increase the expressibility of the HELENA modeling approach and thereby to particularly address the still remaining challenges of ensemble-based systems. We gather our ideas around three advancement fields.

Autonomy: Ensembles are dynamically and autonomously formed communication groups on top of a large component-based platform. In HELENA, we especially

emphasize the distinction between the underlying passive components of the system and the active goal-oriented ensembles on top. However, so far, role creation and retrieval in ensembles has to directly name the component which should own a certain role. If it was possible to define the desired owner via predicates over the properties of components, the owner of the role to be created could freely be chosen from a set of components satisfying the desired properties, either by a centralized ensemble manager or by decentralized negotiations between the components satisfying the desired properties.

Behavioral Guarantees and Safety: According to HELENA, we guarantee goal satisfaction for ensembles by verification. Thereby, we assume reliable communication, non-failing components and deterministic effects of actions. However, today's software systems are situated in uncertain and possibly faulty environments and have to cope with unexpected conditions and changes at runtime. To still guarantee goal-directed behavior, ensembles should apply particular mechanisms for quality assurance:

- Robust message exchange is the foundation for collaboration in ensembles. Thus, techniques for detecting message loss and resending messages have to be incorporated into the messaging facilities of roles.
- A component might become inappropriate or unavailable to adopt a certain role. The component might be overloaded by concurrently contributing to too many ensembles, it might come (mostly physically) out of range for a certain ensemble, or another component might be better suited to play the role under consideration. In such cases, role transferal from one component to another would improve the overall performance of the system and should be supported by a dedicated action.
- Besides quitting and transferring a role in a controlled way, a component might also fail and therefore unexpectedly abandon a role. Monitoring mechanisms should allow the affected ensembles to detect these failures and to restart the role on a different owner based on previous snapshots of the progress of the role.
- In the context of uncertain and unreliable environments, we should even take one step further and wonder whether we really can foresee any possible events and predefine an appropriate behavior coping with these changes. Techniques from artificial intelligence and learning could help to generate behaviors on-demand which take nondeterministic effects and unexpected events into account.

Extensibility and Openness: Another distinguishing property of today's software systems is extensibility to new members of the system and openness towards the interaction with the outside world. HELENA currently only permits a fixed set of components forming the underlying platform of an ensemble-based system and only allows closed ensembles which interact solely with their own members. We envisage several improvements for opening ensemble-based systems modeled with HELENA:

- Joining and leaving components might change the available resources in the overall system. Joining components should be immediately included into the component-based platform and made available to the overall system by

informing all currently running ensembles about these new resources. If components leave the system, the information about their loss should be spread through the system and other components should be found to replace the missing participants of ensembles.

- In a system of dynamically changing participants, directed message exchange as proposed between roles might often fail. Therefore, other forms of communication should be supported additionally. Broadcasting could inform all participants of an ensemble. Group communication rendered in terms of attributed-based communication [ADL16] could select a set of receivers via predicates and thus only inform members of an ensemble which are interested or relevant for this communication.
- Closed ensembles as we considered them throughout this thesis cannot interact with other ensembles, with components or users outside the system or the environment. To open ensembles to the outside, open communication is needed which is not bound to a particular receiver in the ensemble. This open communication could then be used to compose ensemble specifications. Ensembles could collaborate on larger tasks, each ensemble fulfilling one particular subtask.

12.3.2 Goal Specification and Verification

During goal specification and verification, we experienced two main pitfalls. On the one hand, the formulation of goals with pure linear temporal logic did not scale to large systems. Linear temporal logic does not allow the use of quantifiers such that in the worst case all components or roles of the system have to be enumerated to express a goal. A different logic like first-order LTL and an appropriate model-checker would facilitate the compact formulation of goals.

On the other hand, model-checking soon reached its boundaries due to state space explosion, especially for the case study of a larger software system, the Science Cloud Platform. Additionally, we expect the state space to grow even larger if we do not only check a single ensemble, but multiple concurrent ensembles. So far, the translation from HELENA to PROMELA directly represents all HELENA concepts in PROMELA. A first step to reduce the size of the state space could be to optimize this translation in terms of a more compact representation, e.g., by translating components to a user-defined data structure instead of processes. Another idea is to find inductive arguments such that not the whole state space of a large model has to be explored during model-checking. Rather, verification could only prove an initial case and an inductive step which allows to transfer the proof to an arbitrary model size.

12.3.3 Implementing HELENA Models with Object-Orientation

The jHELENA framework already provided a first prototype how an ensemble-based system can be implemented and executed with Java following the formal HELENA syntax and semantics. However, an ensemble-based system is only executed on a single Java virtual machine. To represent realistic ensemble-based systems which are often situated in a physical environment and are distributed on different machines, the framework should allow real distribution of components. With real distribution, new issues arise which have to be taken care of in the formal HELENA approach as well as in its implementation, e.g., messages can be lost or components can get out of reach.

12.3.4 Tool Support for Helena

Although the Helena workbench provided tool support for all phases of the development process and is well integrated into Eclipse, it still leaves room for improvements.

- During specification, we missed a graphical notation especially for the description of ensemble structures. On the basis of the Graphical Modeling Framework of Eclipse, a similar representation as the one used throughout this thesis could easily be added to the textual notation of HelenaText.
- Furthermore, we experienced the benefit of representing all role behaviors of an ensemble together in (possibly hierarchical) interaction diagrams similar to a UML sequence diagram. Such a combined visualization would help to better understand the interactions between collaborating roles.
- The two code generators to Promela and Java do already translate the most relevant parts of a Helena ensemble specification. However, the domain-specific language HelenaText does not yet allow to specify goals and initial states; thus, their representation in Promela and Java cannot be automatically generated. To relieve the developer of an ensemble-based application from any hand-coded implementation in Promela and Java, HelenaText should provide a dedicated notation for goals and initial states and the code generators should systematically translate them.
- During model-checking, Spin produced counterexamples if goals were not satisfied by the Promela translation of the original Helena specification. It required in-depth knowledge about the translation to map these counterexamples back to the original Helena specification. Thus, a developer would benefit from a direct representation of the counterexample in Helena to understand model-checking results more easily.

12.3.5 Case Study

With the case study of the Science Cloud platform, we could show the general applicability of the Helena development methodology to a larger software system. However, the case study did not yet exploit the full potential of Helena. One of the main characteristics of Helena is that it allows to build ensembles from a heterogeneous set of component types such that the ensembles can benefit from the different properties of the underlying component types. Thus, an appropriate case study should be exercised which builds upon a heterogeneous component-based platform. The employed ensembles should distribute work unaffected by the different properties of the underlying components, maybe even benefiting from the different capabilities.

12.3.6 Modeling Awareness and Adaptation with Helena

Our approach to model awareness and adaptation with Helena was just a first step towards adaptive ensembles. It showed that the idea of roles fits well to changing behavioral modes of components. However, it does not yet consider the adaptation of a whole ensemble. Simultaneous role change of the participating components needs to be planned and coordinated such that special coordination support of collective adaptation should be added to Helena. Inspiration could be taken from Puviani et al. [PCZ13] who propose a taxonomy of self-adaptation patterns for collective adaptation based on

the autonomic manager pattern on which we relied for the adaptation of a single self-adaptive component. Furthermore, our approach prescribed a fixed adaptation strategy. If we allowed to either generate an adaptation strategy based on the current situation or to choose between a set of strategies at runtime, we would be more flexible to deal with the different situations the system can be in.

12.4 Final Thoughts

Role-based modeling first made appearance as early as the late 70s. Roles proved to be an intuitive concept to describe context-sensitive responsibilities and behavior of components in a system. They allowed to encapsulate these constituent, but self-contained parts of a component into dedicated entities. The recent and constant growth of software systems in terms of participating components which cannot be handled and grasped at once calls for concepts to focus modeling only on specific parts of a component instead of the whole component. This might usher in a new era of role-based modeling. The HELENA development methodology is a step towards a holistic development methodology for large systems viewed from different perspectives exploiting the notion of roles. Hopefully, it will help to bring role-based modeling to flourish once again.

Appendix A

Correctness Proof in Full Detail

This appendix proves the correctness of the translation from HelenaLight to PromelaLight, i.e., that a HelenaLight specification and its PromelaLight translation satisfy the same set of $LTL_{\setminus \mathbf{X}}$ formulae. We thereby rely on the definition of silent actions in Def. 6.30 and the two relations $\approx$ and $\sim$ defined in Def. 6.31. Table 6.2 gives an overview about the structure of the proof which is shown in full detailed in the following.

A.1 Satisfaction of $LTL_{\setminus \mathbf{X}}$ Formulae in $\approx$-Equivalent States

To be able to apply Thm. 6.5, we first have to show that the relation $\approx$ preserves satisfaction of atomic propositions. Thereby, we have to make the restriction that a role behavior in the underlying HelenaLight ensemble specification must not start with a state label. The reason is the introduction of start labels in PromelaLight. To make it clear, we consider a HelenaLight ensemble specification just consisting of the following role behavior (and the corresponding role type)

$$\textbf{roleBehavior } rt = \mathit{label}.rt.$$

In PromelaLight, this is translated to

$$\textbf{proctype } rt = \mathrm{start}_{rt} : \textbf{true}; \mathit{label} : \textbf{true}; \textbf{goto } \mathrm{start}_{rt}.$$

Let's consider a global HelenaLight ensemble state σ with $\sigma(n) = (rt, v, q, rt)$ and a global PromelaLight state γ with

$$\gamma(n) = (rt, \beta, \mathit{label} : \textbf{true}; \textbf{goto } \mathrm{start}_{rt})$$

such that $\sigma \approx \gamma$. With our previous definition of satisfaction of LTL formulae, the state σ would not satisfy the formula $rt[n]@\mathit{label}$ while the state γ would satisfy it. However, even if we changed the definition of satisfaction in HelenaLight such that the recursive call would also satisfy the formula $rt[n]@\mathit{label}$, we could not overcome the problem. If we consider the same global HelenaLight ensemble state σ with $\sigma(i) = (rt, v, q, rt)$ and a global PromelaLight state γ with

$$\gamma(i) = (rt, \beta, \mathrm{start}_{rt} : \textbf{true}; \mathit{label} : \textbf{true}; \textbf{goto } \mathrm{start}_{rt}),$$

these two states are again $\approx$-equivalent, but the state σ would satisfy the formula $rt[n]@\mathit{label}$ with the changed definition of satisfaction while the state γ would not.

Therefore, we have to make the restriction that a role behavior in the underlying HelenaLight ensemble specification must not start with a state label.

To finally show that $\approx$-equivalent states satisfy the same set of atomic propositions, we use the following auxiliary lemma.

Lemma A.1: State Labels in Process Expressions

Let

- P, P' *be well-formed* HelenaLight *process expressions such that*
  - *they contain recursive process invocations at most for the role behavior* rt *and*
  - *the role behavior declaration for* rt *does not start with a state label,*
- π, π' *be* PromelaLight *process expressions, and*
- $label \neq start_{rt}$ *be a state label in* HelenaLight.

$$\textit{If } trans_{proc}(P) \overset{\tau^*}{\hookrightarrow} \pi, \textit{ then } P = label.P' \textit{ iff } \pi = label : \textbf{\textit{true}}; \pi'.$$

Proof of Lemma A.1

The proof proceeds by induction over the structure of P.

We assume $trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi$. Furthermore, we assume that P only contains process invocations at most for rt, the role behavior for rt does not start with a state label, and $label \neq \text{start}_{rt}$.

We consider all four forms which the process expression P can have according to Def. 6.10. Thus, we have to show that if $P = label.P$, then $\pi = label : \textbf{true}; \pi'$ or if $P \neq label.P$, then $\pi \neq label : \textbf{true}; \pi'$ resp.

Case 1: $P = \textbf{quit}$, i.e., $P \neq label.P'$.

With the definition of $trans_{\text{proc}}$ in Fig. 6.17, we have

$$trans_{\text{proc}}(P) = \textbf{false}.$$

The process expression **false** cannot evolve according to the rules in PromelaLight in Fig. 6.13. Therefore, we know that

$$\pi = \textbf{false} \neq label : \textbf{true}; \pi'.$$

Case 2: $P = a.Q$.

With the definition of $trans_{\text{proc}}$ in Fig. 6.17, we have

$$trans_{\text{proc}}(P) = trans_{\text{act}}(a); trans_{\text{proc}}(P').$$

To elicit all forms the PromelaLight expression π can take in $trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi$, we have to distinguish four types of actions.

Case 2a: $a = X \leftarrow \textbf{create}(rt_j)$, i.e., $P \neq label.P'$.

- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$trans_{\text{proc}}(P) = \ \mathbf{run}\ rt_j(X); trans_{\text{proc}}(Q).$$

- This process expression cannot evolve by τ actions according to the semantic rules of PROMELALIGHT in Fig. 6.13. Therefore, the process expression π must have the form

$$\pi = \mathbf{run}\ rt_j(X); trans_{\text{proc}}(Q).$$

- Thus, this process expression does not start with a state label, i.e.,

$$\boxed{\pi \neq label : \mathbf{true}; \pi'.}$$

Case 2b: $a = Y!msgnm(X)$, i.e., $P \neq label.P'$.

- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$trans_{\text{proc}}(P) = Y!msgnm, X; trans_{\text{proc}}(Q).$$

- This process expression cannot evolve by τ actions according to the semantic rules of PROMELALIGHT in Fig. 6.13. Therefore, the process expression π must have the form

$$\pi = Y!msgnm, X; trans_{\text{proc}}(Q).$$

- Thus, this process expression does not start with a state label, i.e.,

$$\boxed{\pi \neq label : \mathbf{true}; \pi'.}$$

Case 2c: $a = ?msgnm(X{:}rt_j)$, i.e., $P \neq label.P'$.

- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$trans_{\text{proc}}(P) = \mathbf{self}?msgnm, X; trans_{\text{proc}}(Q).$$

- This process expression cannot evolve by τ actions according to the semantic rules of PROMELALIGHT in Fig. 6.13. Therefore, the process expression π must have the form

$$\pi = \mathbf{self}?msgnm, X; trans_{\text{proc}}(Q).$$

- Thus, this process expression does not start with a state label, i.e.,

$$\boxed{\pi \neq label : \mathbf{true}; \pi'.}$$

Case 2d: $a = label$, i.e., $P = label.Q = label.P'$.

- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$trans_{\text{proc}}(P) = label : \mathbf{true}; trans_{\text{proc}}(Q).$$

- Since we assumed that $label \neq \text{start}_{rt}$, this process expression cannot evolve by τ actions according to the semantic rules of PromelaLight in Fig. 6.13. Therefore, the process expression π must have the form

$$\pi = label : \mathbf{true}; trans_{\text{proc}}(Q).$$

- Thus, this process expression starts with a state label, i.e.,

$$\boxed{\pi = label : \mathbf{true}; \pi'.}$$

Case 3: $P = P_1 + P_2$, i.e., $P \neq label.P'$.
With the definition of $trans_{\text{proc}}$ in Fig. 6.17, we have

$$trans_{\text{proc}}(P) = \mathbf{if} :: trans_{\text{proc}}(P_1) :: trans_{\text{proc}}(P_2)\, \mathbf{fi}.$$

We assumed that $\boxed{trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi.}$

- According to the rules *nondet. choice-1* and *nondet. choice-2* in PromelaLight in Fig. 6.13, $trans_{\text{proc}}(P)$ can evolve by τ actions only if either

$$trans_{\text{proc}}(P_1) \overset{\tau^*}{\hookrightarrow} \pi$$

or

$$trans_{\text{proc}}(P_2) \overset{\tau^*}{\hookrightarrow} \pi.$$

- Additionally, both process expressions $trans_{\text{proc}}(P_1)$ and $trans_{\text{proc}}(P_2)$ do not start with a state label because they are translations of well-formed HelenaLight process expressions which do not allow a state label as the first action of each branch of a nondeterministic choice (cf. Def. 6.11).
- Thus, since both process expressions P_1 and P_2 are structurally smaller than P and P_1 and P_2 do not start with a state label (i.e., $P_1 \neq label.P'$ and $P_2 \neq label.P'$), we can conclude by induction that it holds

$$\boxed{\pi \neq label : \mathbf{true}; \pi'.}$$

Case 4: $P = rt$ with **roleBehavior** $rt = Q$, i.e., $P \neq label.P'$.
(Since we assumed that the process expression P contains recursive process invocation at most for rt, only the role behavior declaration of role type rt can be invoked here).

With the definition of $trans_{\text{proc}}$ in Fig. 6.17, we have

$$trans_{\text{proc}}(P) = \mathbf{goto}\ \text{start}_{rt}.$$

We assumed that $\boxed{trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi.}$

- According to the rules *goto* and *sequential composition* in PROMELaLIGHT in Fig. 6.13, $trans_{\text{proc}}(P)$ can evolve by τ actions to

$$\begin{aligned} &\textbf{goto } \text{start}_{rt} \\ &\quad \overset{\tau}{\hookrightarrow} \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(Q) \\ &\quad \overset{\tau}{\hookrightarrow} trans_{\text{proc}}(Q). \end{aligned}$$

- Therefore, the process expression π can either have the form

$$\pi = \textbf{goto } \text{start}_{rt}$$

or the form

$$\pi = \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(Q)$$

or the form

$$\pi = trans_{\text{proc}}(Q).$$

- Additionally, $trans_{\text{proc}}(Q)$ does not start with a state label since we assumed that the role behavior declaration of rt does not start with a state label.
- Thus, since we Q is structurally smaller than P and $Q \neq label.P'$, we can conclude by induction that all three forms of π do not start with a state label, i.e.,

$$\boxed{\pi \neq label : \textbf{true}; \pi'.}$$

■

Proof of Lemma A.1

With the auxiliary lemma Lemma A.1, we are now able to show that $\approx$-equivalent states satisfy the same set of atomic propositions.

Prop. A.2: Satisfaction of $LTL_{\setminus \mathbf{X}}$ Formulae of $\approx$-Equivalent States

Let

- $K(T_{\text{HEL}}) = (S_{\text{HEL}}, A_{\text{HEL}}, \rightarrow^{\bullet}_{\text{HEL}}, F_{\text{HEL}})$ *be the induced Kripke structure of a* HELENALIGHT *ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *with* $\Sigma = (nm, roletypes, roleconstraints)$ *such that*
  - *no role behavior in behaviors contains a start state label* $start_{rt}$ *and*
  - *no role behavior in behaviors starts with a state label and*
- $K(T_{\text{PRM}}) = (S_{\text{PRM}}, A_{\text{PRM}}, \rightarrow^{\bullet}_{\text{PRM}}, F_{\text{PRM}})$ *be the induced Kripke structure of its* PROMELALIGHT *translation* $trans(EnsSpec)$.

For all $\sigma \in S_{\text{HEL}}, \gamma \in S_{\text{PRM}}$, *if* $\sigma \approx \gamma$, *then* $F_{\text{HEL}}(\sigma) = F_{\text{PRM}}(\gamma)$.

Proof of Prop. A.2

We assume $\sigma \approx \gamma = (\texttt{proc}, \texttt{ch})$ and that no role behavior in the ensemble specification starts with a state label and no role behavior contains a start state label start_{rt}.
To show that $F_{\text{HEL}}(\sigma) = F_{\text{PRM}}(\gamma)$, we have to determine satisfaction of $rt[i]@label$ for any $i \in dom(\sigma)$, $i \in dom(\texttt{proc})$, and any state label $label \neq \text{start}_{rt}$.
The local states of the corresponding role instance and process instance resp. are given by $\sigma(i) = (rt, v, q, P)$ and $\texttt{proc}(n) = (rt, \beta, \pi)$ such that it holds that

either $trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi$

or $chandecls_{\text{proc}}(P)\ \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi$
with **roleBehavior** $rt = P$.

If it holds that $trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi$, we can rely on Lemma A.1 and only have to distinguish two cases:

Case 1: $P \neq label.P'$,

i.e., σ does not satisfy any atomic proposition $rt[i]@label$.

Thus, we have to show that also $\texttt{proc}(n) = (rt, \beta, \pi)$ does not satisfy any atomic proposition $rt[i]@label$.

- We assumed that $trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi$.
- Thus, we can apply Lemma A.1 and conclude $\pi \neq label : \textbf{true}; \pi'$.
- Therefore, γ does not satisfy any atomic proposition $rt[i]@label$.

Case 2: $P = label.P'$,

i.e., σ satisfies the atomic proposition $rt[i]@label$. Note that because of our assumption that no role behavior contains a start state label start_{rt}, it holds that $label \neq \text{start}_{rt}$ for any role type rt.

Thus, we have to show that also $\texttt{proc}(n) = (rt, \beta, \pi)$ satisfies the atomic proposition $rt[i]@label$ with $label \neq \text{start}_{rt}$.

- We assumed that $trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi$.
- With the definition of $trans_{\text{proc}}$ in Fig. 6.17, we know that $trans_{\text{proc}}(P) = trans_{\text{proc}}(label.P') = label : \textbf{true}; trans_{\text{proc}}(P')$.
- This process expression cannot evolve by τ actions according to the semantic rules of PROMELALIGHT in Fig. 6.13. Therefore, we know that
$$\pi = label : \textbf{true}; trans_{\text{proc}}(P')$$
- Thus, since $label \neq \text{start}_{rt}$ holds due to the well-formedness of process expressions in HELENALIGHT, γ satisfies the atomic proposition $rt[i]@label$.

If it holds that $chandecls_{\text{proc}}(P)$ start_{rt} : **true**; $trans_{\text{proc}}(P) \xrightarrow{\tau^*} \pi$ with **roleBehavior** $rt = P$, we know that γ does not satisfy any atomic proposition $rt[i]@label$ since the declarations in $chandecls_{\text{proc}}(P)$ cannot contain any state labels, the start label start_{rt} : **true** is not an atomic proposition in PromelaLight (cf. Def. 6.27), and no role behavior starts with a state label. On the other hand, we also know that σ does not satisfy any atomic proposition since we assumed that no role behavior starts with a state label. ■

Proof of Prop. A.2

A.2 Divergence-Sensitivity of the Relation ≈

The second precondition for applying Thm. 6.5 is that the relation ≈ is divergence-sensitive. In this section, we show that the relation ≈ is divergence-sensitive. On the one hand, neither in a HelenaLight specification nor in a PromelaLight translation, infinitely many stutter steps according to the relation ≈ are possible. On the other hand, whenever a HelenaLight specification cannot evolve anymore, then also its PromelaLight specification cannot evolve and resides in a ≈-equivalent state.

For the formal proof, we start by showing if we evolve a HelenaLight ensemble state σ, which is in the relation ≈ with some global PromelaLight state γ, to a HelenaLight ensemble state σ', then the resulting state σ' is not in the relation ≈ with γ anymore.

Lemma A.3: ≈-Equivalence of Evolving HelenaLight Ensemble States

Let

- $T_{\text{Hel}} = (S_{\text{Hel}}, I_{\text{Hel}}, A_{\text{Hel}}, \rightarrow_{\text{Hel}})$ *be the labeled transition system of a* HelenaLight *ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *with* $\Sigma = (nm, roletypes, roleconstraints)$ *such that*
  - *no role behavior in behaviors contains a start state label* $start_{rt}$ *and*
  - *no role behavior in behaviors starts with a state label and*
- $T_{\text{Prm}} = (S_{\text{Prm}}, I_{\text{Prm}}, A_{\text{Prm}}, \rightarrow_{\text{Prm}})$ *be the labeled transition system of its* PromelaLight *translation* $trans(EnsSpec)$*, and*
- a *be a* HelenaLight *action.*

$$\text{For all } \sigma \in S_{\text{Hel}}, \gamma \in S_{\text{Prm}},$$
$$\text{if } \sigma \approx \gamma \text{ and } \sigma \xrightarrow{a}_{\text{Hel}} \sigma', \text{ then } \sigma' \not\approx \gamma.$$

Proof of Lemma A.3

For the proof, we assume $\sigma \approx \gamma$ with γ and it exists $i \in dom(\sigma)$ with $\sigma(i) = (rt, v, q, P)$ and $\texttt{proc}(i) = (pt, \beta, \pi)$. Furthermore, we assume $\sigma \xrightarrow{a}_{\text{Hel}} \sigma'$. Then, the proof has to consider all types of actions in HelenaLight which can evolve σ. We show that $\sigma' \not\approx \gamma$ holds.

Case 1: $a = i : X \leftarrow \mathbf{create}(rt_j)$,
From the *create* rule of HelenaLight in Fig. 6.10, we know that **roleBehavior** $rt_j = P_j$ and

$$\begin{aligned}\sigma' = \ & \sigma[i \mapsto (rt, v[X \mapsto next(\sigma)], q, P')] \\ & [next(\sigma) \mapsto (rt_j, \emptyset[\mathbf{self} \mapsto next(\sigma)], \varepsilon, P_j)].\end{aligned}$$

Obviously, $\sigma' \not\approx \gamma$ holds since $dom(\sigma') \neq dom(\gamma)$ due to the introduction of the new role instance with identifier $next(\sigma)$.

Case 2: $a = i : Y!msgnm(X)$,
From the *send* rule of HelenaLight in Fig. 6.10, we know $j \in dom(\sigma)$ with $\sigma(j) = (rt_j, v_j, q_j, P_j)$ and

$$\begin{aligned}\sigma' = \ & \sigma[i \mapsto (rt, v, q, P')] \\ & [j \mapsto (rt_j, v_j, q_j \cdot msgnm(k), P_j)].\end{aligned}$$

Obviously, $\sigma' \not\approx \gamma$ holds due to the introduction of the new entry $msgnm(k)$ in the message queue of role instance j while all channels in γ were not changed, in particular the channel associated to process instance j.

Case 3: $a = i : ?msgnm(X{:}rt_j)$,
From the *receive* rule of HelenaLight in Fig. 6.10, we know that $\sigma(i) = (rt, v, q \cdot msgnm(k), P)$ and

$$\sigma' = \ \sigma[i \mapsto (rt, v[X \mapsto j], q, P')].$$

Obviously, $\sigma' \not\approx \gamma$ holds since the value j was assigned to the variable X while the local environment functions for all process instances in γ were not changed, in particular the local environment function of process instance i.

Case 4: $a = i : label$
Note that due to the assumption that no role behavior contains a start state label start_{rt}, it holds that $label \neq \text{start}_{rt}$ for any role type rt.

From the *label* rule of HelenaLight in Fig. 6.10, we know that $P \xrightarrow{label} P'$ and

$$\sigma' = \ \sigma[i \mapsto (rt, v, q, P')].$$

Firstly, we can easily prove by induction over the structure of P that if $P \xrightarrow{label} P'$, then $P = label.P'$ holds, i.e., if $P \neq label.P'$, then $P \not\xrightarrow{label} P'$.

- If $P = \mathbf{quit}$ and $P = a.P'$ with $a \neq label$ holds, then P can obviously not evolve by the transition $P \xrightarrow{label} P'$. However, if $P = label.P'$ holds, P can obviously evolve by the transition $P \xrightarrow{label} P'$.

- Nondeterministic choice $P = P_1 + P_2$ can only evolve by the transition $P \xhookrightarrow{label} P'$ if either P_1 or P_2 can evolve by the action *label* to P'. Since we only consider well-formed HelenaLight process expressions according to Def. 6.11, *label* cannot be the first action of P_1 or P_2. Thus, we can conclude that $P = P_1 + P_2$ cannot evolve by the action *label*.
- Recursive process invocation $P = rt$ can only evolve by the transition $P \xhookrightarrow{label} P'$ if **roleBehavior** $rt = Q$ and Q can evolve by the action *label* to P'. Since we only consider role behavior declarations which do not start with a state label, we can conclude that $P = rt$ cannot evolve by the action *label*.

Since $\sigma \approx \gamma$, we further know that

$$trans_{\text{proc}}(P) \xhookrightarrow{\tau^*} \pi \text{ or}$$
$$chandecls_{\text{proc}}(P)\ \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P) \xhookrightarrow{\tau^*} \pi.$$

Since $label \neq \text{start}_{rt}$ and $P = label.P'$, the process expression $trans_{\text{proc}}(P) = trans_{\text{proc}}(label.P')$ can only evolve by the action $label : \textbf{true}$ and not by any τ action. Thus, we know that

$$\pi = label : \textbf{true}; trans_{\text{proc}}(P') \text{ or}$$
$$chandecls_{\text{proc}}(P)\ \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(label.P') \xhookrightarrow{\tau^*} \pi.$$

Since we assumed that no role behaviors in *behaviors* starts with a state label and we only consider the HelenaLight ensemble states of the ensemble specification *EnsSpec* and the global PromelaLight states of its PromelaLight translation *trans*(*EnsSpec*), the second part will never happen. Thus, we finally know that

$$\boxed{\pi = label : \textbf{true}; trans_{\text{proc}}(P').}$$

To show that $\boxed{\sigma' \not\approx \gamma}$, we prove that

$$trans_{\text{proc}}(P') \not\xhookrightarrow{\tau^*} \pi \text{ or}$$
$$chandecls_{\text{proc}}(P')\ \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P') \not\xhookrightarrow{\tau^*} \pi.$$

The proof proceeds by contradiction. Thus, we assume that

$$trans_{\text{proc}}(P') \xhookrightarrow{\tau^*} \pi \text{ or}$$
$$chandecls_{\text{proc}}(P')\ \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P') \xhookrightarrow{\tau^*} \pi.$$

Case 4a: $trans_{\text{proc}}(P') \xhookrightarrow{\tau^*} \pi$.

With the conclusion from $\sigma \approx \gamma$, it holds that

$$trans_{\text{proc}}(P') \xhookrightarrow{\tau^*} label : \textbf{true}; trans_{\text{proc}}(P').$$

Proof of Lemma A.3

This is not possible which we can prove by induction over the structure of P'. The essential arguments are that we cannot create structurally infinite HelenaLight process expressions by adding the action *label* infinitely often and a role behavior declaration is not allowed to start with a state label (thus, recursive process invocation cannot traverse a *label* action as first action).

Case 4b: $\mathit{chandecls}_{\text{proc}}(P')\ \text{start}_{rt} : \mathbf{true}; \mathit{trans}_{\text{proc}}(P') \xhookrightarrow{\tau^*} \pi$.
With the conclusion from $\sigma \approx \gamma$, it holds that

$$\mathit{chandecls}_{\text{proc}}(P')\ \text{start}_{rt} : \mathbf{true}; \mathit{trans}_{\text{proc}}(P') \xhookrightarrow{\tau^*} \mathit{label} : \mathbf{true}; \mathit{trans}_{\text{proc}}(P').$$

However, this is not possible since even if $\mathit{chandecls}_{\text{proc}}(P')$ is empty, it holds that $\mathit{label} \neq \text{start}_{rt}$ due to the assumption that no role behavior contains a start state label start_{rt} and therefore $\mathit{label} : \mathbf{true} \neq \text{start}_{rt} : \mathbf{true}$.

Thus, all types of actions were covered. ■

Secondly, we show if we evolve a global PromelaLight state γ, which is in the relation $\approx$ with some HelenaLight ensemble state σ, to a global PromelaLight state γ', then the resulting state γ' is only in the relation $\approx$ with σ if the executed action was a silent τ action. This lemma therefore also proves that silent steps in PromelaLight do not change the satisfaction of $LTL_{\setminus \mathbf{X}}$ formulae since we showed in the previous section that the relation $\approx$ is property preserving. For example, all channel declarations in $\mathit{chandecls}_{\text{proc}}(P)$ at the beginning of the translation of a role behavior declaration are considered as silent and their execution results in a state which is $\approx$-equivalent to the original HelenaLight state.

Lemma A.4: $\approx$-Equivalence of Evolving Global PromelaLight States

Let

- $T_{\text{Hel}} = (S_{\text{Hel}}, I_{\text{Hel}}, A_{\text{Hel}}, \rightarrow_{\text{Hel}})$ *be the labeled transition system of a* HelenaLight *ensemble specification* $\mathit{EnsSpec} = (\Sigma, \mathit{behaviors})$ *with* $\Sigma = (\mathit{nm}, \mathit{roletypes}, \mathit{roleconstraints})$,
- $T_{\text{Prm}} = (S_{\text{Prm}}, I_{\text{Prm}}, A_{\text{Prm}}, \rightarrow_{\text{Prm}})$ *be the labeled transition system of its* PromelaLight *translation* $\mathit{trans}(\mathit{EnsSpec})$, *and*
- a *be a* PromelaLight *action.*

For all $\sigma \in S_{\text{Hel}}, \gamma \in S_{\text{Prm}}$,
if $\sigma \approx \gamma$ *and* $\gamma \xrightarrow{a}_{\text{Prm}} \gamma'$, *then* $\sigma \approx \gamma'$ *iff* $a = \tau$.

Proof of Lemma A.4

For the proof, we assume $\sigma \approx \gamma$ with $\gamma = (\mathsf{ch}, \mathsf{proc})$ and it exists $i \in \mathit{dom}(\sigma)$ with $\sigma(i) = (rt, v, q, P)$ and $\mathsf{proc}(i) = (pt, \beta, \pi)$. Furthermore,

we assume $\gamma \xrightarrow{a}_{\text{PRM}} \gamma'$. Then, the proof has to consider all types of actions in PROMELALIGHT according to Def. 6.30 which can evolve γ. We show that $\sigma \approx \gamma'$ holds only if evolving γ by silent actions. That means that for non-silent actions, $\sigma \not\approx \gamma'$ holds.
We start with silent actions:

Case 1: $a = i : \textbf{chan}\ var$.
From the *chan-1* rule of PROMELALIGHT in Fig. 6.15, we get that

$$\gamma' = (\mathsf{ch}, \mathsf{proc}[i \mapsto (pt, \beta[var \mapsto \textbf{null}], \pi')]).$$

Therefore, we only have to consider the changes in β and of π to show that $\sigma \approx \gamma'$.

- The local environment β is only changed for the variable var. Since the global PROMELALIGHT state γ is well-defined according to Def. 6.25, we know that π must be well-formed. To be well-formed, the variable occurring in a channel declaration may not be declared before (cf. Def. 6.19). Thus, we know that var is a fresh variable, i.e., $var \notin dom(\beta)$. That means that var does not have to relate to any variable in v because it holds that $var \notin dom(v)$ since $dom(w) \subseteq dom(\beta)$.
- Since $\sigma \approx \gamma$, we know that

  $$trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi \text{ or}$$
  $$chandecls_{\text{proc}}(P)\ \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi.$$

  Since we assumed that γ evolves by the action $a = i : \textbf{chan}\ var$, it must hold that

  $$chandecls_{\text{proc}}(P)\ \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi$$

  and not $trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi$ because the function $trans_{\text{proc}}$ cannot produce any channel declarations according to its definition in Fig. 6.17.
  From the premise of the *chan-1* rule of PROMELALIGHT in Fig. 6.15, we know that $\pi \xrightarrow{\textbf{chan}\ X} \pi'$. Since $\textbf{chan}\ X$ is a silent action according to Def. 6.30, we have $\pi \overset{\tau}{\hookrightarrow} \pi'$. By appending this to the previous evolution, we get

  $$chandecls_{\text{proc}}(P)\ \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P) \overset{\tau}{\hookrightarrow} \pi \overset{\tau^*}{\hookrightarrow} \pi'.$$

  Therefore, it holds that

  $$chandecls_{\text{proc}}(P)\ \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi'$$

  which fulfills the last condition for $\sigma \approx \gamma'$.

Thus, it holds that $\sigma \approx \gamma'$.

Case 2: $a = i : \textbf{chan}\ var = [const]\ \textbf{of}\ \{typelist\}$.
From the *chan-2* rule of PromelaLight in Fig. 6.15, we have

$$\gamma' = (\mathsf{ch}[next(\mathsf{ch}) \mapsto (typelist, const, \varepsilon)],\\ \mathsf{proc}[i \mapsto (pt, \beta[var \mapsto next(\mathsf{ch})], \pi')]).$$

Again, we only have to consider the changes in β and of π to show that $\sigma \approx \gamma'$. The proof proceeds analog to Case 1.

Case 3: $a = i : \text{start}_{pt} : \textbf{true}$.
From the *label* rule of PromelaLight in Fig. 6.15, we have

$$\gamma' = (\mathsf{ch}, \mathsf{proc}[i \mapsto (pt, \beta, \pi')]).$$

In this case, we only have to consider the change of π to π' to show that $\sigma \approx \gamma'$.

- Since $\sigma \approx \gamma$, we know that

  $$trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi \text{ or}$$
  $$chandecls_{\text{proc}}(P)\ \text{start}_{pt} : \textbf{true}; trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi.$$

- From the premise of the *label* rule of PromelaLight in Fig. 6.15, we know that $\pi \xhookrightarrow{\text{start}_{pt}:\textbf{true}} \pi'$. Since $\text{start}_{pt} : \textbf{true}$ is a silent action according to Def. 6.30, we have $\pi \overset{\tau}{\hookrightarrow} \pi'$.
- By appending this to the previous evolution, we get

  $$trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi \overset{\tau}{\hookrightarrow} \pi' \text{ or}$$
  $$chandecls_{\text{proc}}(P)\ \text{start}_{pt} : \textbf{true}; trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi \overset{\tau}{\hookrightarrow} \pi'.$$

- Therefore, it holds that

  $$trans_{\text{proc}}(P) \overset{\tau^*}{\hookrightarrow} \pi' \text{ or}$$
  $$chandecls_{\text{proc}}(P)\ \text{start}_{pt} : \textbf{true}; trans_{\text{proc}}(P) \overset{\tau}{\hookrightarrow} \pi'.$$

  which fulfills the last condition for $\sigma \approx \gamma'$.

Case 4: $a = i : \textbf{goto}\ \text{start}_{pt}$
From the *goto* rule of PromelaLight in Fig. 6.14, we have

$$\gamma' = (\mathsf{ch}, \mathsf{proc}[i \mapsto (pt, \beta, \pi')]).$$

In this case, we only have to consider the change of π to π' to show that $\sigma \approx \gamma'$.

The proof proceeds analog to Case 3.

Thus, all four types of silent actions were covered.

Secondly, we consider all types of non-silent actions:

Case 5: $a = i : \mathit{label} : \mathbf{true}$ with $\mathit{label} \neq \mathrm{start}_{pt}$ since this was already covered in Case 3.
From the *label* rule of PROMELALIGHT in Fig. 6.15, we have

$$\gamma' = (\mathsf{ch}, \mathsf{proc}[i \mapsto (pt, \beta, \pi')]).$$

In this case, we only have to consider the change of π to π' to show that $\sigma \not\approx \gamma'$, i.e., we only have to show that it must hold that

$$\mathit{trans}_{\mathrm{proc}}(P) \stackrel{\mathcal{T}^*}{\not\longrightarrow} \pi' \text{ and}$$
$$\mathit{chandecls}_{\mathrm{proc}}(P)\ \mathrm{start}_{pt} : \mathbf{true}; \mathit{trans}_{\mathrm{proc}}(P) \stackrel{\mathcal{T}^*}{\not\longrightarrow} \pi'.$$

- Since $\sigma \approx \gamma$, we know that

$$\mathit{trans}_{\mathrm{proc}}(P) \stackrel{\tau^*}{\longrightarrow} \pi \text{ or}$$
$$\mathit{chandecls}_{\mathrm{proc}}(P)\ \mathrm{start}_{pt} : \mathbf{true}; \mathit{trans}_{\mathrm{proc}}(P) \stackrel{\tau^*}{\longrightarrow} \pi.$$

- We know from the premise of the *label* rule of PROMELALIGHT in Fig. 6.15 that $\pi \xrightarrow{\mathit{label}:\mathbf{true}} \pi'$.
- Thus, with the first part of the conclusion from $\sigma \approx \gamma$, we have

$$\mathit{trans}_{\mathrm{proc}}(P) \stackrel{\tau^*}{\longrightarrow} \pi \xrightarrow{\mathit{label}:\mathbf{true}} \pi'.$$

Since $\mathit{label} : \mathbf{true} \neq \mathrm{start}_{pt} : \mathbf{true}$ is a non-silent action according to Def. 6.30, it is not possible that $\mathit{trans}_{\mathrm{proc}}(P)$ or $\mathit{chandecls}_{\mathrm{proc}}(P)\ \mathrm{start}_{pt} : \mathbf{true}; \mathit{trans}_{\mathrm{proc}}(P)$ evolves to π' by τ actions only, thus

$$\mathit{trans}_{\mathrm{proc}}(P) \stackrel{\mathcal{T}^*}{\not\longrightarrow} \pi' \text{ and}$$
$$\mathit{chandecls}_{\mathrm{proc}}(P)\ \mathrm{start}_{pt} : \mathbf{true}; \mathit{trans}_{\mathrm{proc}}(P) \stackrel{\mathcal{T}^*}{\not\longrightarrow} \pi'.$$

- With the second part of the conclusion from $\sigma \approx \gamma$, we have

$$\mathit{chandecls}_{\mathrm{proc}}(P)\ \mathrm{start}_{pt} : \mathbf{true}; \mathit{trans}_{\mathrm{proc}}(P) \stackrel{\tau^*}{\longrightarrow} \pi \xrightarrow{\mathit{label}:\mathbf{true}} \pi'.$$

Again since $\mathit{label} : \mathbf{true} \neq \mathrm{start}_{pt} : \mathbf{true}$ is a non-silent action according to Def. 6.30, it is not possible that $\mathit{trans}_{\mathrm{proc}}(P)$ or $\mathit{chandecls}_{\mathrm{proc}}(P)\ \mathrm{start}_{pt} : \mathbf{true}; \mathit{trans}_{\mathrm{proc}}(P)$ evolves to π' by τ actions only, thus

$$\mathit{trans}_{\mathrm{proc}}(P) \stackrel{\mathcal{T}^*}{\not\longrightarrow} \pi' \text{ and}$$
$$\mathit{chandecls}_{\mathrm{proc}}(P)\ \mathrm{start}_{pt} : \mathbf{true}; \mathit{trans}_{\mathrm{proc}}(P) \stackrel{\mathcal{T}^*}{\not\longrightarrow} \pi'.$$

Thus, it holds that $\sigma \not\approx \gamma'$.

Case 6: $a = i : \textbf{goto}\ label$ with $label \neq \text{start}_{pt}$ since this was already covered in Case 4.
According to the translation function in Fig. 6.17 from HelenaLight to PromelaLight, arbitrary **goto**-jumps cannot happen. Thus, this case does not have to be considered.

Case 7: $a = i : \textbf{run}\ pt_j(var)$
From the *run* rule of PromelaLight in Fig. 6.14, we have $\gamma' = (\mathsf{ch}, \mathsf{proc}')$ with

$$\mathsf{proc}' = \mathsf{proc}[i \mapsto (pt, \beta, \pi')] \\ [next(\mathsf{proc}) \mapsto (pt_j, \emptyset[\textbf{self} \mapsto c], \text{start}_{pt_j} : \textbf{true}; seq)]$$

Obviously, $\sigma \not\approx \gamma'$ holds since $dom(\sigma) \neq dom(\mathsf{proc}')$ due to the introduction of the process instance with identifier $next(\mathsf{proc})$.

Case 8: $a = i : var_1!const, var_2$
From the *send* rule of PromelaLight in Fig. 6.14, we have $\gamma' = (\mathsf{ch}', \mathsf{proc}')$ with

$$\mathsf{ch}' = \mathsf{ch}[\beta(var_1) \mapsto (T, \kappa, \omega \cdot (const, v))].$$

Since we only consider PromelaLight translations of a HelenaLight specification, var_1 must correspond to an existing role instance in σ (cf. well-formedness of process expression in HelenaLight Def. 2.10), i.e.., $\beta(var_1)$ must refer to a channel instance which is referenced by $\beta(\textbf{self})$ of an existing process instance. Therefore, obviously, $\sigma \not\approx \gamma'$ holds due to the introduction of the new entry $(const, v)$ in channel $\beta(var_1)$ while all message queues in σ were not changed, in particular the message queue of role instance var_1.

Case 9: $a = i : var_1?const, var_2$
From the *receive* rule of PromelaLight in Fig. 6.14, we have $\gamma = (\mathsf{ch}, \mathsf{proc})$ with $\mathsf{ch}(\beta(var_1)) = (T, \kappa, (const, v) \cdot \omega)$ and $\gamma' = (\mathsf{ch}', \mathsf{proc}')$ with

$$\mathsf{ch}' = \mathsf{ch}[\beta(var_1) \mapsto (T, \kappa, \omega)].$$

Since we only consider PromelaLight translations of a HelenaLight specification, var_1 can only be the predefined constant **self** (cf. translation function between HelenaLight and PromelaLight in Fig. 6.17).
Then, obviously, $\sigma \not\approx \gamma'$ holds since the entry $(const, v)$ in channel $\beta(\textbf{self}) = i$ was removed while all message queues in σ were not changed, so in particular the message queue of role instance i.

Thus, all four types of non-silent actions were covered. ■

With these two auxiliary lemmata, Lemma A.3 and Lemma A.4, we can show that the relation ≈ is divergence-sensitive.

The first observation is that neither in a HelenaLight specification nor in a PromelaLight translation, infinitely many stutter steps according to the relation ≈ are possible. In HelenaLight, silent actions do not exists at all and therefore stutter steps are not possible. In PromelaLight, silent actions are only channel declarations, **goto**-statements and start state labels. In the PromelaLight translation of a role behavior, channel declarations only occur at the beginning of the translation and initialize a fixed set of local variables for the translated role behavior. Therefore, they cannot produce an infinite sequence of stutter steps. The only possibility to produce an infinite sequence of stutter steps are repeated **goto**-jumps. For example, if the role behavior **roleBehavior** $rt = rt$ was allowed, it would be translated to the PromelaLight process type

$$\textbf{proctype}\ rt(\textbf{chan self}) = \text{start}_{rt} : \textbf{true}; \textbf{goto}\ \text{start}_{rt}.$$

The execution of this PromelaLight process type would result in an infinite sequence of τ actions while the HelenaLight role behavior might not evolve at all. Therefore, we do not allow immediate process invocation (also not as nested in nondeterministic choice)

Secondly, we observe that the evolution of a HelenaLight ensemble specification and also of a PromelaLight specification can get stuck if no action is executable anymore, i.e., the semantic labeled transition systems resides in a terminal state. When inducing a Kripke structure from these specifications, we do not allow finite path fragments through the Kripke structure anymore and elongate those paths by an infinite sequence of τ actions residing in the terminal state forever. Thus, to show divergence-sensitivity of the relation ≈, we have to prove that whenever a HelenaLight ensemble specification gets stuck, its PromelaLight translation can also not evolve anymore (and vice versa). Since all HelenaLight actions are translated to a single semantically equivalent PromelaLight action, the only critical case is recursive process invocation. In PromelaLight, recursive process invocation needs two silent actions (a **goto**-jump and passing the start label) to invoke the role behavior from the beginning. To guarantee divergence-sensitivity of the relation ≈, we have to guarantee that in PromelaLight those two silent actions only result in a terminal state (elongated by τ actions) if HelenaLight also does (and vice versa). The only assumption we have to make is that process invocation is not one of the branches of nondeterministic choice. To illustrate this, let's consider the HelenaLight role behavior

$$\textbf{roleBehavior}\ rt = ?msgnm_1(X : rt).(rt + X!msgnm_2(\textbf{self}).\textbf{quit}).$$

which contradicts the aforementioned assumption. It is translated to the following PromelaLight process type

$$\begin{aligned}
\textbf{proctype}\ rt(\textbf{chan self}) \quad &= \textbf{chan}\ X; \\
&\quad \text{start}_{rt} : \textbf{true}; \\
&\quad \textbf{self}?msgnm_1, X; \\
&\quad \textbf{if} \\
&\quad :: \textbf{goto}\ \text{start}_{rt} \\
&\quad :: X!msgnm_2, \textbf{self}; \textbf{false} \\
&\quad \textbf{fi}
\end{aligned}$$

If the evolution of the role behavior reaches the nondeterministic choice in HelenaLight, it checks executability of both, receiving the message $msgnm_1$ and sending the message $msgnm_2$. If receiving the message is not possible, but sending the message is possible, the first branch will not be selected for execution (but the second branch). In PromelaLight, however, the executability of the **goto**-jump and of sending the message $msgnm_2$ are checked. Therefore, in PromelaLight, the first branch can be selected for execution since we only reach the non-executable action of receiving the message $msgnm_2$ after executing some silent actions. This means, that while HelenaLight would select the second branch, PromelaLight would select the first branch and result in a terminal state which is elongated to an infinite sequence of silent actions. To avoid this problem, we have to assume that process invocation is not one of the branches of nondeterministic choice to show divergence-sensitivity of the relation $\approx$.

Prop. A.5: Divergence-Sensitivity of the Relation $\approx$

Let

- $K(T_{\text{HEL}}) = (S_{\text{HEL}}, A_{\text{HEL}}, \rightarrow^{\bullet}_{\text{HEL}}, F_{\text{HEL}})$ *be the induced Kripke structure of a* HelenaLight *ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *with* $\Sigma = (nm, roletypes, roleconstraints)$ *such that*
  - *no role behavior in behaviors contains a start state label* $start_{rt}$,
  - *no role behavior in behaviors starts with a state label, and*
  - *in any nondeterministic choice construct in a role behavior in behaviors, process invocation is not one of the branches, and*
- $K(T_{\text{PRM}}) = (S_{\text{PRM}}, A_{\text{PRM}}, \rightarrow^{\bullet}_{\text{PRM}}, F_{\text{PRM}})$ *be the induced Kripke structure of its* PromelaLight *translation* $trans(EnsSpec)$.

The relation $\approx$ is divergence-sensitive for $K(T_{\text{HEL}})$ and $K(T_{\text{PRM}})$.

Proof of Prop. A.5

To recap, the property of divergence-sensitivity is defined as follows:
For all $s_1 \in S_{\text{HEL}}, t_1 \in S_{\text{PRM}}$ with $s_1 \approx t_1$ holds: if there exists an (infinite) path fragment $s_1 s_2 s_3 \ldots$ in $K(T_{\text{HEL}})$ with $s_i \approx t_1$ for all $i \geq 1$, then there exists an (infinite) path fragment $t_1 t_2 t_3 \ldots$ in $K(T_{\text{PRM}})$ with $s_1 \approx t_j$ for all $j \geq 1$ and symmetrically for (infinite) path fragments in $K(T_{\text{PRM}})$.

We first show that a HelenaLight ensemble specification as well as its PromelaLight translation cannot evolve by infinitely many τ actions. In the direction from HelenaLight to PromelaLight, we can apply Lemma A.3. The lemma proves that whenever a HelenaLight ensemble state σ with $\sigma \approx \gamma$ (for a global PromelaLight state γ) evolves, its resulting HelenaLight ensemble state σ' is no longer in the relation $\approx$ with γ. Thus, we know that there does not exist an (infinite) path fragment $s_1 s_2 s_3 \ldots$ in T_{HEL} such that all states s_i remain $\approx$-equivalent to one particular state $t_1 \in S_{\text{PRM}}$.

In the other direction from PromelaLight to HelenaLight, we can apply Lemma A.4. The lemma proves that whenever a global PromelaLight

state γ with $\sigma \approx \gamma$ (for a HelenaLight ensemble state σ) evolves, its resulting global PromelaLight state γ' is only in the relation $\approx$ with σ if the executed action was a τ action. That means that we can only construct an (infinite) path fragment $t_1t_2t_3\ldots$ in $K(T_{\text{PRM}})$ with $s_1 \approx t_j$ if we only execute τ actions on the corresponding path in the underlying labeled transition system. However, we consider a PromelaLight translation of a HelenaLight specification. Hence, we cannot deduce an infinite sequence of τ actions as we discuss in the following:

- According to the translation function in Fig. 6.17, both types of channel declarations **chan** X and **chan** X = [*roleconstraints*(rt_j)] **of** {**mtype**, **chan**} only occur at the beginning of the translation in PromelaLight before the start state label. Thus, they cannot be reached again (e.g., by a **goto**-jump) and cannot produce an infinite sequence of τ actions.
- According to the translation function in Fig. 6.17, each **goto**-jump is always the action **goto** start$_{rt}$ and therefore always jumps to the start label start$_{rt}$: **true** of the process. Thus, a **goto** action can only produce an infinite sequence of τ actions if the action start$_{rt}$: **true** can produce it.
- According to the translation function in Fig. 6.17, the action start$_{rt}$: **true** is always followed by the first action of the translated role behavior declaration rt. This first action cannot be a silent action: (a) Channel declarations are only included before the start state label. (b) Since we do not allow recursive process invocation immediately at the beginning of a role behavior declaration (cf. Def. 6.12), the executed first action in PromelaLight cannot be a **goto** action. (c) Due to the assumption that no role behavior contains a start state labels, start state labels cannot be the first action of a role behavior declaration. (d) Every other action cannot produce an infinite sequence of τ action as argued before. Thus, the action start$_{rt}$: **true** cannot produce an infinite sequence of τ actions.

Thus, we know that there does not exists an (infinite) path fragment $t_1t_2t_3\ldots$ in T_{PRM} such that all states t_j remain $\approx$-equivalent to one particular state $s_1 \in S_{\text{HEL}}$.
Apart from that, we have to consider that in the induced Kripke structure of a HelenaLight ensemble specification and of its PromelaLight translation all terminal states are equipped with a self-loop. This means that whenever the evolution of a specification blocks, the path is elongated by infinitely many τ action. To prove divergence-sensitivity of the relation $\approx$, we have to show that whenever the HelenaLight ensemble specification remains in a certain state σ, then also the corresponding PromelaLight translation remains in a certain state γ with $\sigma \approx \gamma$ (and vice versa). Since all HelenaLight actions are represented by exactly one semantically equivalent action in PromelaLight, we only have to consider which process constructs might allow different paths in HelenaLight and PromelaLight. In general, only nondeterministic choice allows two different paths through the specifications. HelenaLight

Proof of Prop. A.5

and PROMELALIGHT can only select differently from these two paths if the first actions of the branches differ between HELENALIGHT and PROMELALIGHT. Since all HELENALIGHT actions are represented by a single PROMELALIGHT action, the only possibility for a differing selection is if one branch is a process invocation. In this case, HELENALIGHT determines the executability of this branch based on the executability of the first action of the invoked process (which might not be executable) while PROMELALIGHT determines the executability based on the executability of the **goto**-statement (which is always executable). However, this case was explicitly excluded from the proposition. Thus, with all other process constructs, a HELENALIGHT ensemble specification only remains in a certain state σ iff its PROMELALIGHT translation remains in a state γ with $\sigma \approx \gamma$. ■

A.3 $\sim$-Equivalence of Initial States

The third precondition for applying Thm. 6.5 is that any admissible initial state of a HELENALIGHT ensemble specification is related to its PROMELALIGHT translation by the relation $\sim$.

Prop. A.6: $\sim$-Equivalence of Initial States

Let σ be an admissible initial state of a HELENALIGHT *ensemble specification, then $\sigma \sim trans_{init}(\sigma)$.*

Proof of Prop. A.6

Consider an admissible initial state σ of a HELENALIGHT ensemble specification and its translation $trans_{\text{init}}(\sigma)$.

- In HELENALIGHT, σ consists of at least one role instance and for all role instance states it holds that $\sigma(i) = (rt, \emptyset[\textbf{self} \mapsto i], \varepsilon, P)$ with **roleBehavior** $rt = P$ (cf. Def. 6.17).
- For the PROMELALIGHT translation holds $trans_{\text{init}}(\sigma) = (\mathsf{ch}, \mathsf{proc})$ with (cf. Def. 6.29):

  (1) $dom(\mathsf{ch}) = dom(\sigma)$,

  (2) $dom(\mathsf{proc}) = dom(\sigma)$, and

  (3) for all $i \in dom(\mathsf{proc})$:
  $\mathsf{proc}(i) = (rt, \emptyset[\textbf{self} \mapsto c_i],$
  $chandecls_{\text{proc}}(P)\ \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P))$
  with $c_i = ((\textbf{mtype}, \textbf{chan}), roleconstraints(rt), \varepsilon) \in dom(\mathsf{ch})$
  and $c_i \neq c_j$ for $i \neq j$.

- Therefore, all conditions for $\sigma \sim trans_{\text{init}}(\sigma)$ are satisfied, in particular item (2d) is satisfied since

$$\pi = chandecls_{\text{proc}}(P)\ \text{start}_{rt} : \textbf{true}; trans_{\text{proc}}(P)$$

  with **roleBehavior** $rt = P$.

■

A.4 ≈-Stutter Simulation of HelenaLight Specifications

Using the definition of the relation $\sim$ and $\approx$ in Def. 6.31, we can show that a HelenaLight ensemble specification is ≈-stutter simulated by its PromelaLight translation according to the relation $\sim$.

To be able to show that, we need Lemma A.4 and an auxiliary lemma. The last auxiliary lemma states that whenever a HelenaLight process expression can evolve by a local HelenaLight action, then its PromelaLight translation can also evolve with a sequence of τ actions and the corresponding local PromelaLight action to the HelenaLight action, possibly preceded by some silent **goto**-jumps and start state label actions. The proof of this lemma is straightforward. Only process invocation introduces some additional silent τ steps in PromelaLight which are not part of the translation of HelenaLight action since invocation is realized by a **goto** statement to the beginning of a process declaration.

Lemma A.7: Co-Evolution of PromelaLight Process Expressions

Let EnsSpec be a HelenaLight *specification such that no role behavior in the specification contains a start state label and PrmSpec its* PromelaLight *translation. In the context of these two specifications, let P, P' be well-formed process expressions in* HelenaLight *and let a be a* HelenaLight *action on the level of roles.*

$$\textit{If } P \overset{a}{\hookrightarrow} P', \textit{ then } trans_{proc}(P) \xhookrightarrow{\tau^* trans_{act}(a)} trans_{proc}(P').$$

Proof of Lemma A.7

The proof proceeds by induction on the structure of P.

We assume $P \overset{a}{\hookrightarrow} P'$ and consider all forms which the process expression P can have according to Def. 6.10.

Case 1: $P = \mathbf{quit}$.

The process expression **quit** cannot evolve according to the rules in HelenaLight in Fig. 6.9. Therefore, the left side of the implication is not satisfied and it remains nothing to show.

Case 2: $P = a.Q$, $a.Q \overset{a}{\hookrightarrow} Q$, and $Q = P'$.

Thus, we have to show that

$$trans_{\text{proc}}(a.Q) \xhookrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(Q).$$

With the definition of $trans_{\text{proc}}$ in Fig. 6.17, we have

$$trans_{\text{proc}}(a.Q) = trans_{\text{act}}(a); trans_{\text{proc}}(Q).$$

To evolve this process expression in PromelaLight, we have to distinguish four types of actions.

Case 2a: $a = X \leftarrow \mathbf{create}(rt_j)$

- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$trans_{\text{act}}(a); trans_{\text{proc}}(Q) = \textbf{run}\ rt_j(X); trans_{\text{proc}}(Q).$$

- By applying the *sequential composition* rule of PROMELA-LIGHT in Fig. 6.13, we evolve this expression to

$$\textbf{run}\ rt_j(X); trans_{\text{proc}}(Q) \xrightarrow{\textbf{run}\ rt_j(X)} trans_{\text{proc}}(Q).$$

- By using the notation $trans_{\text{act}}$, we get

$$\textbf{run}\ rt_j(X); trans_{\text{proc}}(Q) \xrightarrow{trans_{\text{act}}(X \leftarrow \textbf{create}(rt_j))} trans_{\text{proc}}(Q).$$

Thus, it holds that

$$\boxed{trans_{\text{proc}}(a.Q) \xrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(Q)}$$

since we can always substitute τ^* by an empty sequence of τ actions.

Case 2b: $a = Y!msgnm(X)$

- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$trans_{\text{act}}(a); trans_{\text{proc}}(Q) = Y!msgnm, X; trans_{\text{proc}}(Q).$$

- By applying the *sequential composition* rule of PROMELA-LIGHT in Fig. 6.13, we evolve this expression to

$$Y!msgnm, X; trans_{\text{proc}}(Q) \xrightarrow{Y!msgnm,X} trans_{\text{proc}}(Q).$$

- By using the notation $trans_{\text{act}}$, we get

$$Y!msgnm, X; trans_{\text{proc}}(Q) \xrightarrow{trans_{\text{act}}(Y!msgnm(X))} trans_{\text{proc}}(Q).$$

Thus, it holds that

$$\boxed{trans_{\text{proc}}(a.Q) \xrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(Q)}$$

since we can always substitute τ^* by an empty sequence of τ actions.

Case 2c: $a = ?msgnm(X{:}rt_j)$

- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$trans_{\text{act}}(a); trans_{\text{proc}}(Q) = \textbf{self}?msgnm, X; trans_{\text{proc}}(Q).$$

- By applying the *sequential composition* rule of PROMELA-LIGHT in Fig. 6.13, we evolve this expression to

$$\textbf{self}?msgnm, X; trans_{\text{proc}}(Q) \xrightarrow{\textbf{self}?msgnm,X} trans_{\text{proc}}(Q).$$

- By using the notation $trans_{\text{act}}$, we get

$$\textbf{self}?msgnm, X; trans_{\text{proc}}(Q) \xhookrightarrow{trans_{\text{act}}(?msgnm(X))} trans_{\text{proc}}(Q).$$

Thus, it holds that

$$\boxed{trans_{\text{proc}}(a.Q) \xhookrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(Q)}$$

since we can always substitute τ^* by an empty sequence of τ actions.

Case 2d: $a = label$

- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$trans_{\text{act}}(a); trans_{\text{proc}}(Q) = label : \textbf{true}; trans_{\text{proc}}(Q).$$

- By applying the *sequential composition* rule of PROMELALIGHT in Fig. 6.13, we evolve this expression to

$$label : \textbf{true}; trans_{\text{proc}}(Q) \xhookrightarrow{label:\textbf{true}} trans_{\text{proc}}(Q).$$

- By using the notation $trans_{\text{act}}$, we get

$$label : \textbf{true}; trans_{\text{proc}}(Q) \xhookrightarrow{trans_{\text{act}}(label)} trans_{\text{proc}}(Q).$$

Thus, it holds that

$$\boxed{trans_{\text{proc}}(a.Q) \xhookrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(Q)}$$

since we can always substitute τ^* by an empty sequence of τ actions.

Case 3: $P = P_1 + P_2$, $P_1 + P_2 \overset{a}{\hookrightarrow} P_1'$, and $P_1' = P'$

Thus, we have to show that

$$\boxed{trans_{\text{proc}}(P_1 + P_2) \xhookrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(P_1').}$$

- With the definition of $trans_{\text{proc}}$ in Fig. 6.17, we have

$$trans_{\text{proc}}(P_1 + P_2) = \textbf{if} :: trans_{\text{proc}}(P_1) :: trans_{\text{proc}}(P_2)\, \textbf{fi}.$$

- Since we assumed $P_1 + P_2 \overset{a}{\hookrightarrow} P_1'$, we know from the premise of the *nondet. choice-1* rule in HELENALIGHT in Fig. 6.9 that $P_1 \overset{a}{\hookrightarrow} P_1'$. Thus, since P_1 is structurally smaller than $P_1 + P_2$, we can assume by induction that $trans_{\text{proc}}(P_1) \xhookrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(P_1')$.
- With this as premise, we can evolve the original process expression by applying the *nondet. choice-1* rule in PROMELALIGHT in Fig. 6.13 to

$$\textbf{if} :: trans_{\text{proc}}(P_1) :: trans_{\text{proc}}(P_2)\, \textbf{fi} \xhookrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(P_1').$$

Thus, it holds that

$$trans_{\text{proc}}(P_1 + P_2) \xhookrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(P_1').$$

Case 4: $P = P_1 + P_2$, $P_1 + P_2 \overset{a}{\hookrightarrow} P_2'$, and $P_2' = P'$

Proof analog to Case 2.

Case 5: $P = N$, $N \overset{a}{\hookrightarrow} Q'$ with **roleBehavior** $N = Q$, and $Q' = P'$

Thus, we have to show that

$$trans_{\text{proc}}(N) \xhookrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(Q').$$

- With the definition of $trans_{\text{proc}}$ in Fig. 6.17, we have

$$trans_{\text{proc}}(N) = \textbf{goto } \text{start}_N.$$

- Additionally, with the definition of $trans_{\text{role}}$ in Fig. 6.17, we have

$$\begin{aligned} trans_{\text{role}}(\textbf{roleBehavior } N = Q) \quad &= \textbf{proctype } N(\textbf{chan self}) \ \{ \\ &\quad chandecls_{\text{proc}}(Q) \\ &\quad \text{start}_N : \textbf{true}; trans_{\text{proc}}(Q) \ \}. \end{aligned}$$

- With this as side-condition (and since we assumed that no role behavior contains a start state label), we can evolve the original process expression by applying the *goto* rule of PromelaLight in Fig. 6.13 to

$$\textbf{goto } \text{start}_N \xhookrightarrow{\textbf{goto } \text{start}_N} \text{start}_N : \textbf{true}; trans_{\text{proc}}(Q).$$

- By applying the *sequential composition* rule of PromelaLight in Fig. 6.13, we evolve this expression further to

$$\text{start}_N : \textbf{true}; trans_{\text{proc}}(Q) \xhookrightarrow{\text{start}_N : \textbf{true}} trans_{\text{proc}}(Q).$$

- Since we assumed $N \overset{a}{\hookrightarrow} Q'$, we know from the premise of the *process invocation* rule in HelenaLight in Fig. 6.9 that $Q \overset{a}{\hookrightarrow} Q'$. Thus, since Q is structurally smaller than N, we can assume by induction that $trans_{\text{proc}}(Q) \xhookrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(Q')$.

Concluding, we know that

$$\begin{aligned} trans_{\text{proc}}(N) = \textbf{goto } \text{start}_N \\ &\xhookrightarrow{\textbf{goto } \text{start}_N} \text{start}_N : \textbf{true}; trans_{\text{proc}}(Q) \\ &\xhookrightarrow{\text{start}_N : \textbf{true}} trans_{\text{proc}}(Q) \\ &\xhookrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(Q'). \end{aligned}$$

Proof of Lemma A.7

Thus, since **goto** start_N and start label start_N : **true** steps are silent τ actions, it holds that

$$trans_{\text{proc}}(N) \xrightarrow{\tau^* trans_{\text{act}}(a)} trans_{\text{proc}}(Q').$$

Thus, the structure of P was completely covered. ■

With Lemma A.4 and Lemma A.7, we are able to show that the relation $\sim$ is a $\approx$-stutter simulation of HELENALIGHT by its PROMELALIGHT translation. Although it would be enough to use the relation $\approx$ itself as a $\approx$-stutter simulation, we employ the two relations $\sim$ and $\approx$ since they make the proof easier to read. The reason is that the relation $\sim$ requires a stronger assumption about the process expression representing the progress of a role instance and its translation in PROMELALIGHT. A HELENALIGHT ensemble state and a global PROMELALIGHT state are related by the relation $\sim$ if for each role instance in the HELENALIGHT ensemble state, the translation of the process expression describing the role's progress of execution is exactly the same as the process expression of the corresponding process in the global PROMELALIGHT state (amongst others). In contrast, in the relation $\approx$, the translation of the process expression must only be able to evolve via τ actions to the process expression in PROMELALIGHT.

In the proof, we exploit this stronger assumption to show more easily that a HELENALIGHT action can be simulated in PROMELALIGHT. We use the relation $\sim$ to prove that each action in HELENALIGHT can either directly be simulated in PROMELALIGHT or PROMELALIGHT employs some stutter steps (keeping the states in the relation $\approx$) before actually performing the translated action. Nevertheless, we want to stress here that also the relation $\approx$ could be used as a $\approx$-stutter simulation. It could even be shown that the relation $\approx$ is a weak bisimulation. However, for the lack of understandability we will rely on the stronger assumption of the relation $\sim$ in the following proof.

Prop. A.8: ≈-Stutter Simulation of HELENALIGHT Specifications

Let

- $K(T_{\text{HEL}}) = (S_{\text{HEL}}, A_{\text{HEL}}, \rightarrow^{\bullet}_{\text{HEL}}, F_{\text{HEL}})$ *be the induced Kripke structure of a* HELENALIGHT *ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *with* $\Sigma = (nm, roletypes, roleconstraints)$ *such that no role behavior in behaviors contains a start state label and*
- $K(T_{\text{PRM}}) = (S_{\text{PRM}}, A_{\text{PRM}}, \rightarrow^{\bullet}_{\text{PRM}}, F_{\text{PRM}})$ *be the induced Kripke structure of its* PROMELALIGHT *translation* $trans(EnsSpec)$.

$\sim$ *is a* $\approx$*-stutter simulation of* $K(T_{\text{HEL}})$ *by* $K(T_{\text{PRM}})$.

Proof of Prop. A.8

We have to show that the relation $\sim$ satisfies the property of a $\approx$-stutter simulation described in Def. 6.2. In the proof, we rely on the underlying labeled transition systems T_{HEL} and T_{PRM} of $K(T_{\text{HEL}})$ and $K(T_{\text{PRM}})$. We show the following property which obviously entails the required property for

a $\approx$-stutter simulation:

$$\begin{array}{c}\text{For all } \sigma \in S_{\text{HEL}}, \gamma \in S_{\text{PRM}} \text{ with } \sigma \sim \gamma,\\ \text{if there exists } \sigma' \in S_{\text{HEL}}, a \in A_{\text{HEL}} \text{ such that } \sigma \xrightarrow{a}_{\text{HEL}} \sigma',\\ \text{then there exist } n \geq 0 \text{ and } \gamma_1, \ldots, \gamma_n, \gamma' \in S_{\text{PRM}} \text{ such that}\\ \gamma \xrightarrow{\tau}_{\text{PRM}} \gamma_1 \ldots \xrightarrow{\tau}_{\text{PRM}} \gamma_n \xrightarrow{trans_{\text{act-global}}(a)}_{\text{PRM}} \gamma',\\ \text{for all } 1 \leq i \leq n : \sigma \approx \gamma_i, \text{ and } \sigma' \sim \gamma'.\end{array}$$

The proof has to consider all four types of global actions possible in the transition $\sigma \xrightarrow{a}_{\text{HEL}} \sigma'$ in HELENALIGHT.

We assume that $\sigma \sim \gamma = (\mathsf{ch}, \mathsf{proc})$ and $\sigma \xrightarrow{a}_{\text{HEL}} \sigma'$.

Case 1: $a = i : X \leftarrow \mathbf{create}(rt_j)$ for $i \in dom(\sigma)$.
We know from the notation in Sec. 6.5.1 that

$$trans_{\text{act-global}}(i : X \leftarrow \mathbf{create}(rt_j)) = i : \mathbf{run}\ rt_j(X).$$

For this case, we show that

there exist $n \geq 0$ and $\gamma_1, \ldots, \gamma_n, \gamma' \in S_{\text{PRM}}$ such that

(1) $\gamma \xrightarrow{\tau}_{\text{PRM}} \gamma_1 \ldots \xrightarrow{\tau}_{\text{PRM}} \gamma_n$ with $\sigma \approx \gamma_i$ for all $1 \leq i \leq n$,

(2) $\gamma_n \xrightarrow{i:\mathbf{run}\ rt_j(X)}_{\text{PRM}} \gamma'$ with $\sigma' \sim \gamma'$.

Before tackling each item separately, we summarize what we can deduce from the assumption $\sigma \xrightarrow{i:X \leftarrow \mathbf{create}(rt_j)}_{\text{HEL}} \sigma'$.

- From the premise of the *create* rule in HELENALIGHT in Fig. 6.10, we know that the premise $P_i \xrightarrow{X \leftarrow \mathbf{create}(rt_j)} P_i'$ holds. With Lemma A.7, we can therefore conclude that

$$trans_{\text{proc}}(P_i) \xrightarrow{\tau^* trans_{\text{act}}(X \leftarrow \mathbf{create}(rt_j))} trans_{\text{proc}}(P_i').$$

- Formulated differently, we can conclude that there exists a PROMELALIGHT process expression π_i' such that

$$trans_{\text{proc}}(P_i) \xhookrightarrow{\tau^*} \pi_i' \xrightarrow{\mathbf{run}\ rt_j(X)} trans_{\text{proc}}(P_i').$$

With this knowledge on the level of process types in PROMELALIGHT, we tackle each item separately.

We first map the evolution of the single process type by a sequence of τ actions to the evolution of the global PROMELALIGHT state by a sequence of τ actions. Since we assumed that $\sigma \sim \gamma = (\mathsf{ch}, \mathsf{proc})$, we know that there exists $\mathsf{proc}(i) = (pt_i, \beta_i, \pi_i)$ with $\pi_i = trans_{\text{proc}}(P_i)$ or $\pi_i = chandecls_{\text{proc}}(P_i)\ \text{start}_{rt_i} : \mathbf{true}; trans_{\text{proc}}(P_i)$.

Let's first assume that $\pi_i = trans_{\text{proc}}(P_i)$. We furthermore know from above that $trans_{\text{proc}}(P_i) \xhookrightarrow{\tau^*} \pi_i'$. We want to prove that by applying a

sequence of τ actions to the process instance i, we also evolve the global PROMELALIGHT state γ by a sequence of τ actions and only traverse γ_i with $\sigma \approx \gamma_i$, i.e., there exist $n \geq 0$ and $\gamma_1, \ldots, \gamma_{n-1} \in S_{\text{PRM}}$ such that

$$\boxed{\gamma \xrightarrow{\tau}_{\text{PRM}} \gamma_1 \ldots \xrightarrow{\tau}_{\text{PRM}} \gamma_n \text{ with } \sigma \approx \gamma_i \text{ for all } 1 \leq i \leq n.}$$

- $trans_{\text{proc}}(P_i) \xrightarrow{\tau^*} \pi_i'$ satisfies the premise of all semantic rules for silent actions in PROMELALIGHT in Fig. 6.14 and Fig. 6.15, i.e., the rules *goto*, *label*, *chan-1*, and *chan-2*.
- The side-conditions of all semantic rules for silent actions in PROMELALIGHT in Fig. 6.14 and Fig. 6.15 trivially hold.
- Thus, there exist $n \geq 0$ and $\gamma_1, \ldots, \gamma_n$ such that
$$\boxed{\begin{aligned} \gamma \xrightarrow{\tau}_{\text{PRM}} \gamma_1 \ldots \xrightarrow{\tau}_{\text{PRM}} \gamma_n &= (\mathsf{ch}_n, \mathsf{proc}_n) \\ &= (\mathsf{ch}, \mathsf{proc}[i \mapsto (rt_i, \beta_i, \pi_i')]) \end{aligned}}$$
- Since we assumed $\sigma \sim \gamma$ and $\sim \subseteq \approx$, we can apply Lemma A.4 to conclude $\boxed{\sigma \approx \gamma_i \text{ for all } 1 \leq i \leq n.}$

Let's now assume that

$$\pi_i = chandecls_{\text{proc}}(P_i)\ \text{start}_{rt_i} : \mathbf{true}; trans_{\text{proc}}(P_i).$$

Since all channel declarations and the start state label are silent actions, we know that $\pi_i \xrightarrow{\tau^*} trans_{\text{proc}}(P_i)$. For the evolution of $trans_{\text{proc}}(P_i)$, we follow the same argumentation as for the first case such we can also conclude that $\boxed{\sigma \approx \gamma_i \text{ for all } 1 \leq i \leq n.}$

Secondly, we show that there exists $\gamma' \in S_{\text{PRM}}$ such that $\boxed{\gamma_n \xrightarrow{i:\mathbf{run}\ rt_j(X)}_{\text{PRM}} \gamma' \text{ with } \sigma' \sim \gamma'}$. That means that γ_n satisfies the premise and all side-conditions of the *run* rule in the PROMELALIGHT semantics in Fig. 6.14 and therefore the *run* rule can be applied to γ_n resulting in the state γ' which is $\sim$-equivalent to σ'.

To this end, we know that the premise and all side-conditions of the *create* rule of the HELENALIGHT semantics in Fig. 6.10 are satisfied by σ since we assumed $\sigma \xrightarrow{a}_{\text{HEL}} \sigma'$.

- From above, we know that the premise of the *create* rule in PROMELALIGHT in Fig. 6.14 is satisfied by
$$\pi_i' \xrightarrow{\mathbf{run}\ rt_j(X)} trans_{\text{proc}}(P_i')$$
for $i \in dom(\mathsf{proc}_n)$ and with the definition of γ_n we know that item (1) of the side-condition is satisfied by
$$\mathsf{proc}_n(i) = (rt_i, \beta_i, \pi_i').$$

- Since we consider the PROMELALIGHT translation $trans(EnsSpec)$, we can assume that the PROMELALIGHT process first declared and initialized the channel variable X which is used in a **run** $rt_i(X)$ before (cf. definition of $trans_{\text{role}}$ in Fig. 6.17 and $chandecls_{\text{proc}}$ in Fig. 6.18).
- Therefore, we know that item (2) of the side-condition of the *run* rule in PROMELALIGHT in Fig. 6.14 is satisfied by $\beta_i(X) \in dom(\mathsf{ch}_n)$. Additionally, we only consider well-formed process expressions in HELENALIGHT. Therefore, the process expression in HELENALIGHT did not use the variable X before to send messages to it. Thus, we know that the corresponding channel variable in PROMELALIGHT was only declared and initialized, but never used. With the definition of $chandecls_{\text{proc}}$ in Fig. 6.18, we know that the variable X was declared by

$$\textbf{chan } X = [roleconstraints(rt_j)] \textbf{ of } \{\textbf{mtype}, \textbf{chan}\}.$$

- From the *create* rule in HELENALIGHT in Fig. 6.10, we know that **roleBehavior** $rt_j = P_j$.
- Thus, since we know that $trans(EnsSpec)$ contains $trans_{\text{role}}(b)$ for every role behavior b in $EnsSpec$, item (3) of the side-condition of the *run* rule in PROMELALIGHT in Fig. 6.14 is satisfied by

$$\begin{aligned} &trans_{\text{role}}(\textbf{roleBehavior } rt_j = P_j) = \\ &\quad \textbf{proctype } rt_j(\textbf{chan self}) \ \{ \\ &\quad\quad chandecls_{\text{proc}}(P_j) \\ &\quad\quad \text{start}_{rt_j} : \textbf{true}; \\ &\quad\quad trans_{\text{proc}}(P_j)\}. \end{aligned}$$

Concluding, the premise and all side-conditions of applying the *run* rule in PROMELALIGHT in Fig. 6.14 are satisfied by γ_n and we can conclude that $\gamma_n \xrightarrow{i:\textbf{run } rt_j(X)}_{\text{PRM}} \gamma'$ with

$$\begin{aligned} \gamma' \quad = (&\mathsf{ch}[\beta_i(X) \mapsto ((\textbf{mtype}, \textbf{chan}), roleconstraints(rt_j), \varepsilon)], \\ &\mathsf{proc}[i \mapsto (rt_i, \beta_i, trans_{\text{proc}}(P_i'))] \\ &\quad [next(\mathsf{proc}) \mapsto (rt_j, \emptyset[\textbf{self} \mapsto \beta_i(X)], \\ &\quad\ chandecls_{\text{proc}}(P_j) \ \text{start}_{rt_j} : \textbf{true}; trans_{\text{proc}}(P_j))]). \end{aligned}$$

Lastly, we have to show that $\sigma' \sim \gamma'$. From the *create* rule in HELENALIGHT in Fig. 6.10, we know that

$$\begin{aligned} \sigma' \quad = &\ \sigma[i \mapsto (rt_i, v_i[X \mapsto next(\sigma)], q_i, P_i')] \\ &[next(\sigma) \mapsto (rt_j, \emptyset[\textbf{self} \mapsto next(\sigma)], \varepsilon, P_j)]. \end{aligned}$$

Since in this case the definition of the relation $\sim$ in Def. 6.31 requires amongst others that if the role instance $next(\sigma)$ has to execute the process expression P_j, then the process instance $next(\mathsf{proc})$ has to execute

the process expression $chandecls_{\text{proc}}(P_j)$ start_{rt_j} : **true**; $trans_{\text{proc}}(P_j)$, it holds that $\sigma' \sim \gamma'$.

Case 2: $a = i : Y!msgnm(X)$ for $i \in dom(\sigma)$.
We know from the notation in Sec. 6.5.1 that

$$trans_{\text{act-global}}(i : Y!msgnm(X)) = i : Y!msgnm, X.$$

For this case, we show that

there exist $n \geq 0$ and $\gamma_1, \ldots, \gamma_n, \gamma' \in S_{\text{PRM}}$ such that

(1) $\gamma \xrightarrow{\tau}_{\text{PRM}} \gamma_1 \ldots \xrightarrow{\tau}_{\text{PRM}} \gamma_n$ with for all $1 \leq i \leq n : \sigma \approx \gamma_i$,

(2) $\gamma_n \xrightarrow{i:Y!msgnm,X}_{\text{PRM}} \gamma'$ with $\sigma' \sim \gamma'$.

Before tackling each item separately, we summarize what we can deduce from the assumption $\sigma \xrightarrow{i:Y!msgnm(X)}_{\text{HEL}} \sigma'$.

- From the premise of the *send* rule in HELENALIGHT in Fig. 6.10, we know that the premise $P_i \xrightarrow{Y!msgnm(X)} P_i'$ holds. With Lemma A.7, we can therefore conclude that

$$trans_{\text{proc}}(P_i) \xrightarrow{\tau^* trans_{\text{act}}(Y!msgnm(X))} trans_{\text{proc}}(P_i').$$

- Formulated differently, we can conclude that there exists a PROMELALIGHT process expression π_i' such that

$$trans_{\text{proc}}(P_i) \xrightarrow{\tau^*} \pi_i' \xrightarrow{Y!msgnm,X} trans_{\text{proc}}(P_i').$$

With this knowledge on the level of process types in PROMELALIGHT, we tackle each item separately.

We first map the evolution of the single process type by a sequence of τ actions to the evolution of the global PROMELALIGHT state by a sequence of τ actions. The proof proceeds analogously to Case 1 and we know that there exist $n \geq 0$ and $\gamma_1, \ldots, \gamma_n \in S_{\text{PRM}}$ such that

$$\begin{aligned} \gamma \xrightarrow{\tau}_{\text{PRM}} \gamma_1 \ldots \xrightarrow{\tau}_{\text{PRM}} \gamma_n \quad &= (\mathsf{ch}_n, \mathsf{proc}_n) \\ &= (\mathsf{ch}, \mathsf{proc}[i \mapsto (rt_i, \beta_i, \pi_i')]) \end{aligned}$$

and for all $1 \leq i \leq n : \sigma \approx \gamma_i$.

Secondly, we show that there exists $\gamma' \in S_{\text{PRM}}$ such that $\gamma_n \xrightarrow{i:Y!msgnm,X}_{\text{PRM}} \gamma'$ with $\sigma' \sim \gamma'$. That means that γ_n satisfies the premise and all side-conditions of the *send* rule in the PROMELALIGHT semantics in Fig. 6.14 and therefore the *send* rule can be applied to γ_n resulting in a state γ' which is $\sim$-equivalent to σ'.

To this end, we know that the premise and all side-conditions of the *send* rule of the HELENALIGHT semantics in Fig. 6.10 are satisfied by σ since we assumed $\sigma \xrightarrow{a}_{\text{HEL}} \sigma'$.

- We repeat: From the premise of the *send* rule in HelenaLight in Fig. 6.10, we know that the premise $P_i \xrightarrow{Y!msgnm(X)} P_i'$ holds. With Lemma A.7, we can therefore conclude that

$$trans_{\text{proc}}(P_i) \xrightarrow{\tau^* trans_{\text{act}}(Y!msgnm(X))} trans_{\text{proc}}(P_i').$$

  Formulated differently, we can conclude that there exists a PromelaLight process expression π_i' such that

$$trans_{\text{proc}}(P_i) \xrightarrow{\tau^*} \pi_i' \xrightarrow{Y!msgnm,X} trans_{\text{proc}}(P_i'),$$

- Thus, we know that the premise of the *send* rule in PromelaLight in Fig. 6.14 is satisfied by

$$\pi_i' \xrightarrow{Y!msgnm,X} trans_{\text{proc}}(P_i')$$

  for $i \in dom(\mathsf{proc}_n)$ with $\mathsf{proc}_n(i) = (rt_i, \beta_i, \pi_i')$.
- From the side-conditions of the *send* rule in HelenaLight in Fig. 6.10, we know that $i \in dom(\sigma)$ with $\sigma(i) = (rt_i, v_i, q_i, P_i)$.
- Thus, since $\sigma \approx \gamma_n$, we know that item (1) of the side-condition of the *send* rule in PromelaLight in Fig. 6.14 is satisfied by $i \in dom(\mathsf{proc}_n)$ with $\mathsf{proc}_n(i) = (rt_i, \beta_i, \pi_i')$.
- From the side-conditions of the *send* rule in HelenaLight in Fig. 6.10, we know that $v_i(Y) = j \in dom(\sigma)$ with $\sigma(j) = (rt_j, v_j, q_j, P_j)$ and $|q_j| < roleconstraints(rt_j)$.
- Thus, since $\sigma \approx \gamma_n$ and well-definedness of global PromelaLight states, we know that item (2) of the side-condition of the *send* rule in PromelaLight in Fig. 6.14 is satisfied by $\beta_i(Y) = \beta_j(\mathbf{self}) \in dom(\mathsf{ch}_n)$ with $j \in dom(\mathsf{proc}_n)$ and $\mathsf{ch}_n(\beta_j(\mathbf{self})) = (T, \kappa, \omega)$ and $|\omega| < roleconstraints(rt_j) = \kappa$.
- From the side-conditions of the *send* rule in HelenaLight in Fig. 6.10, we know that $v_i(X) = k \in dom(\sigma)$.
- Thus, since $\sigma \approx \gamma_n$, we know that item (3) of the side-condition of the *send* rule in PromelaLight in Fig. 6.14 is satisfied by $\beta_i(X) = \beta_k(\mathbf{self}) \in dom(\mathsf{ch}_n)$.

Concluding, the premise and all side-conditions for applying the *send* rule in PromelaLight in Fig. 6.14 are satisfied by γ_n and we can conclude that $\boxed{\gamma_n \xrightarrow{i:Y!msgnm,X}_{\text{PRM}} \gamma'}$ with

$$\begin{aligned}\gamma' \quad = (&\mathsf{ch}[\beta_j(\mathbf{self}) \mapsto (T, \kappa, \omega \cdot (msgnm, \beta_k(\mathbf{self})))],\\ &\mathsf{proc}[i \mapsto (rt_i, \beta_i, trans_{\text{proc}}(P_i'))]).\end{aligned}$$

Lastly, we have to show that $\sigma' \sim \gamma'$. From the *send* rule in HelenaLight in Fig. 6.10, we know that

$$\sigma' = \sigma[i \mapsto (rt_i, v_i, q_i, P_i')][j \mapsto (rt_j, v_j, q_j \cdot msgnm(k), P_j)].$$

Thus, it trivially holds that $\sigma' \sim \gamma'$.

Case 3: $a = i :?msgnm(X{:}rt_j)$ for $i \in dom(\sigma)$.
We know from the notation in Sec. 6.5.1 that

$$trans_{\text{act-global}}(i :?msgnm(X{:}rt_j)) = i : \textbf{self}?msgnm, X.$$

For this case, we show that

there exist $n \geq 0$ and $\gamma_1, \ldots, \gamma_n, \gamma' \in S_{\text{PRM}}$ such that

(1) $\gamma \xrightarrow{\tau}_{\text{PRM}} \gamma_1 \ldots \xrightarrow{\tau}_{\text{PRM}} \gamma_{n-1}$ with $\sigma \approx \gamma_i$ for all $1 \leq i \leq n$,

(2) $\gamma_n \xrightarrow{i:\textbf{self}?msgnm,X}_{\text{PRM}} \gamma'$ with $\sigma' \sim \gamma'$.

Before tackling each item separately, we summarize what we can deduce from the assumption $\sigma \xrightarrow{i:?msgnm(X:rt_j)}_{\text{HEL}} \sigma'$.

- From the premise of the *receive* rule in HELENALIGHT in Fig. 6.10, we know that the premise $P_i \xrightarrow{i:?msgnm(X:rt_j)} P_i'$ holds. With Lemma A.7, we can therefore conclude that

$$trans_{\text{proc}}(P_i) \xrightarrow{\tau^* trans_{\text{act}}(?msgnm(X:rt_j))} trans_{\text{proc}}(P_i').$$

- Formulated differently, we can conclude that there exists a PROMELALIGHT process expression π_i' such that

$$trans_{\text{proc}}(P_i) \xrightarrow{\tau^*} \pi_i' \xrightarrow{\textbf{self}?msgnm,X} trans_{\text{proc}}(P_i').$$

With this knowledge on the level of process types in PROMELALIGHT, we tackle each item separately. We first map the evolution of the single process type by a sequence of τ actions to the evolution of the global PROMELALIGHT state by a sequence of τ actions. The proof proceeds analog to Case 1 and we know that there exist $n \geq 0$ and $\gamma_1, \ldots, \gamma_n \in S_{\text{PRM}}$ such that

$$\begin{aligned} \gamma \xrightarrow{\tau}_{\text{PRM}} \gamma_1 \ldots \xrightarrow{\tau}_{\text{PRM}} \gamma_n \quad &= (\mathsf{ch}_n, \mathsf{proc}_n) \\ &= (\mathsf{ch}, \mathsf{proc}[i \mapsto (rt_i, \beta_i, \pi_i')]) \end{aligned}$$

and for all $1 \leq i \leq n : \sigma \approx \gamma_i$.

Secondly, we show that there exists $\gamma' \in S_{\text{PRM}}$ such that $\gamma_n \xrightarrow{i:\textbf{self}?msgnm,X}_{\text{PRM}} \gamma'$ with $\sigma' \sim \gamma'$. That means that γ_n satisfies the premise and all side-conditions of the *receive* rule in the PROMELALIGHT semantics in Fig. 6.14 and therefore the *receive* rule can be applied to γ_n resulting in the state γ' which is $\sim$-equivalent to σ'.

To this end, we know that the premise and all side-conditions of the *receive* rule of the HELENALIGHT semantics in Fig. 6.10 are satisfied by σ since we assumed $\sigma \xrightarrow{a}_{\text{HEL}} \sigma'$.

- From before, we know that the premise and the first line of the side-condition of the *receive* rule in PromelaLight in Fig. 6.14 is satisfied by
$$\pi_i' \xrightarrow{\textbf{self}?msgnm,X} trans_{\text{proc}}(P_i')$$
for $i \in dom(\mathsf{proc}_n)$ and with the definition of γ_n we know that item (1) of the side condition is satisfied by
$$\mathsf{proc}_n(i) = (rt_i, \beta_i, \pi_i').$$
- From the side-conditions of the *receive* rule in HelenaLight in Fig. 6.10, we know that $i \in dom(\sigma)$ with
$$\sigma(i) = (rt_i, v_i, msgnm(j) \cdot q_i, P_i).$$
- Thus, since $\sigma \approx \gamma_n$, we know that item (2) of the side-condition of the *receive* rule in PromelaLight in Fig. 6.14 is satisfied by $\beta_i(\textbf{self}) \in dom(\mathsf{ch})$ with
$$\mathsf{ch}(\beta_i(\textbf{self})) = (T, \kappa, (msgnm, \beta_j(\textbf{self})) \cdot \omega).$$
- From the side-conditions of the *receive* rule in HelenaLight in Fig. 6.10, we know that $j \in dom(\sigma)$.
- Thus, since $\sigma \approx \gamma_n$ and because of the well-formedness of global PromelaLight states, we know that the latter part of item (2) of the side-condition of the *receive* rule in PromelaLight in Fig. 6.14 is satisfied by $j \in dom(\mathsf{proc}_n)$ with $\mathsf{proc}_n(j) = (rt_j, \beta_j, \pi_j)$ and $\beta_j(\textbf{self}) \in dom(\mathsf{ch}_n)$.
- Since we consider the PromelaLight translation $trans(EnsSpec)$, we can assume that the PromelaLight process first declared the channel variable X which is used in a $\textbf{self}?msgnm, X$ before (cf. definition of $trans_{\text{role}}$ in Fig. 6.17 and $chandecls_{\text{proc}}$ in Fig. 6.18).
- Therefore, we know that item (3) of the side-condition of the *receive* rule in PromelaLight in Fig. 6.14 is satisfied by $\beta_i(X) \in dom(\mathsf{ch}_n)$.

Concluding, the premise and all side-conditions of applying the *receive* rule in PromelaLight in Fig. 6.14 are satisfied by γ_n and we can conclude that $\boxed{\gamma_n \xrightarrow{i:\textbf{self}?msgnm,X}_{\text{PRM}} \gamma'}$ with

$$\begin{aligned}\gamma' \quad &= (\mathsf{ch}[\beta_i(\textbf{self}) \mapsto (T, \kappa, \omega)],\\ &\quad\ \mathsf{proc}[i \mapsto (rt_i, \beta_i[X \mapsto \beta_j(\textbf{self})], trans_{\text{proc}}(P_i'))]).\end{aligned}$$

Lastly, we have to show that $\sigma' \sim \gamma'$. From the *send* rule in HelenaLight in Fig. 6.10, we know that

$$\sigma' = \sigma[i \mapsto (rt_i, v_i[X \mapsto j], q_i, P_i')].$$

Thus, it trivially holds that $\boxed{\sigma' \sim \gamma'.}$

Case 4: $a = i : label$ for $i \in dom(\sigma)$.

We know from the notation in Sec. 6.5.1 that

$$trans_{\text{act-global}}(i : label) = i : label : \mathbf{true}.$$

For this case, we show that

there exist $n \geq 0$ and $\gamma_1, \ldots, \gamma_n, \gamma' \in S_{\text{PRM}}$ such that

(1) $\gamma \xrightarrow{\tau}_{\text{PRM}} \gamma_1 \ldots \xrightarrow{\tau}_{\text{PRM}} \gamma_n$ with $\sigma \approx \gamma_i$ for all $1 \leq i \leq n$,

(2) $\gamma_n \xrightarrow{i:label:\mathbf{true}}_{\text{PRM}} \gamma'$ with $\sigma' \sim \gamma'$.

Before tackling each item separately, we summarize what we can deduce from the assumption $\sigma \xrightarrow{i:label}_{\text{HEL}} \sigma'$.

- From the premise of the *label* rule in HELENALIGHT in Fig. 6.10, we know that the premise $P_i \xhookrightarrow{label} P_i'$ holds. With Lemma A.7, we can therefore conclude that

$$trans_{\text{proc}}(P_i) \xhookrightarrow{\tau^* trans_{\text{act}}(label)} trans_{\text{proc}}(P_i').$$

- Formulated differently, we can conclude that there exists a PROMELALIGHT process expression π_i' such that

$$trans_{\text{proc}}(P_i) \xhookrightarrow{\tau^*} \pi_i' \xhookrightarrow{label:\mathbf{true}} trans_{\text{proc}}(P_i').$$

With this knowledge on the level of process types in PROMELALIGHT, we tackle each item separately. We first map the evolution of the single process type by a sequence of τ actions to the evolution of the global PROMELALIGHT state by a sequence of τ actions. The proof proceeds analog to Case 1 and we know that there exist $n \geq 0$ and $\gamma_1, \ldots, \gamma_n \in S_{\text{PRM}}$ such that

$$\begin{aligned} \gamma \xrightarrow{\tau}_{\text{PRM}} \gamma_1 \ldots \xrightarrow{\tau}_{\text{PRM}} \gamma_n \quad &= (\mathsf{ch}_n, \mathsf{proc}_n) \\ &= (\mathsf{ch}, \mathsf{proc}[i \mapsto (rt_i, \beta_i, \pi_i')]) \end{aligned}$$

and for all $1 \leq i \leq n : \sigma \approx \gamma_i$.

Secondly, we show that there exists $\gamma' \in S_{\text{PRM}}$ such that $\gamma_n \xrightarrow{i:label:\mathbf{true}}_{\text{PRM}} \gamma'$ with $\sigma' \sim \gamma'$. That means that γ_n satisfies the premise and all side-conditions of the *label* rule in the PROMELALIGHT semantics in Fig. 6.15 and therefore the *label* rule can be applied to γ_n.

To this end, we know that the premise and all side-conditions of the *label* rule of the HELENALIGHT semantics in Fig. 6.10 are satisfied by σ since we assumed $\sigma \xrightarrow{a}_{\text{HEL}} \sigma'$.

Proof of Prop. A.8

- We repeat: From the premise of the *label* rule in HelenaLight in Fig. 6.10, we know that the premise $P_i \xrightarrow{label} P_i'$ holds. With Lemma A.7, we can therefore conclude that

$$trans_{\text{proc}}(P_i) \xrightarrow{\tau^* trans_{\text{act}}(label)} trans_{\text{proc}}(P_i').$$

  Formulated differently, we can conclude that there exists a PromelaLight process expression π_i' such that

$$trans_{\text{proc}}(P_i) \xrightarrow{\tau^*} \pi_i' \xrightarrow{label:\mathbf{true}} trans_{\text{proc}}(P_i'),$$

- Thus, we know that the premise of the *label* rule in PromelaLight in Fig. 6.15 is satisfied by

$$\pi_i' \xrightarrow{label:\mathbf{true}} trans_{\text{proc}}(P_i')$$

  for$i \in dom(\mathsf{proc}_n)$ with $\mathsf{proc}_n(i) = (rt_i, \beta_i, \pi_i')$.
- From the side-condition of the *label* rule in HelenaLight in Fig. 6.10, we know that $i \in dom(\sigma)$ with $\sigma(i) = (rt_i, v_i, q_i, P_i)$.
- Thus, since $\sigma \approx \gamma_n$, we know that the side-condition of the *send* rule in PromelaLight in Fig. 6.15 is satisfied by $i \in dom(\mathsf{proc}_n)$ with $\mathsf{proc}_n(i) = (rt_i, \beta_i, \pi_i')$.

Concluding, the premise and all side-conditions for applying the *label* rule in PromelaLight in Fig. 6.15 are satisfied by γ^* and we can conclude that $\boxed{\gamma_n \xrightarrow{i:label:\mathbf{true}}_{\text{Prm}} \gamma'}$ with

$$\gamma' = (\mathsf{ch}, \mathsf{proc}[i \mapsto (rt_i, \beta_i, trans_{\text{proc}}(P_i'))]).$$

Lastly, we have to show that $\sigma' \sim \gamma'$. From the *send* rule in HelenaLight in Fig. 6.10, we know that

$$\sigma' = \sigma[i \mapsto (rt_i, v_i, q_i, P_i')].$$

Thus, it trivially holds that $\boxed{\sigma' \sim \gamma'.}$

Thus, all four types of global actions possible in the transition $\sigma \xrightarrow{a}_{\text{Hel}} \sigma'$ were covered. ■

A.5 ≈-Stutter Simulation of PromelaLight Translations

The other way round, we can even show that the relation $\approx^{-1}$ itself is a $\approx^{-1}$-stutter simulation of a PromelaLight translation by its corresponding HelenaLight specification. In this direction, we do not need two relations as from HelenaLight to PromelaLight. The reason is that in a PromelaLight translation we may take more steps than in the corresponding HelenaLight specification, but in HelenaLight we never take more steps than in PromelaLight.

Before actually proving that the relation $\approx^{-1}$ is a $\approx^{-1}$-stutter simulation of a PROMELALIGHT translation by its corresponding HELENALIGHT specification, we need an auxiliary lemma which reasons about actions other than τ. The lemma shows that whenever a translated process expression in PROMELALIGHT can evolve by a sequence of τ actions and a translated action other than τ, its corresponding HELENALIGHT process expression can also evolve with the corresponding action.

Lemma A.9: Co-Evolution of HELENALIGHT Process Expressions

Let EnsSpec be a HELENALIGHT *specification such that no role behavior in the specification contains a start state label and PrmSpec its* PROMELALIGHT *translation. In the context of these two specifications, let P, P' be well-formed process expressions in* HELENALIGHT*, π, π', π'' well-formed process expressions in* PROMELALIGHT*, and let a be a* HELENALIGHT *action on the level of roles and b a* PROMELALIGHT *action on the level of process instances.*

$$\text{If } \pi \overset{\tau^*}{\hookrightarrow} \pi' \overset{trans_{act}(a)}{\longrightarrow} \pi'' \text{ and } \pi = trans_{proc}(P),$$
$$\text{then } P \overset{a}{\hookrightarrow} P' \text{ and } \pi'' = trans_{proc}(P').$$

Proof of Lemma A.9

The proof proceeds by induction over the structure of P.

We assume $\pi \overset{\tau^*}{\hookrightarrow} \pi' \overset{trans_{\text{act}}(a)}{\longrightarrow} \pi''$ and $\pi = trans_{\text{proc}}(P)$ and consider all forms which the process expression P can have according to Def. 6.10. Thus, we have to show that $P \overset{a}{\hookrightarrow} P'$ and $\pi'' = trans_{\text{proc}}(P')$.

Case 1: $P = \mathbf{quit}$, i.e., $\pi = trans_{\text{proc}}(P) = trans_{\text{proc}}(\mathbf{quit}) = \mathbf{false}$.
The process expression **false** cannot evolve according to the rules in PROMELALIGHT in Fig. 6.13. Therefore, the left side of the implication is not satisfied.

Case 2: $P = a.Q$, i.e., $\pi = trans_{\text{proc}}(P) = trans_{\text{proc}}(a.Q)$
We can immediately show, that $P \overset{a}{\hookrightarrow} P'$ holds, since in HELENALIGHT, the expression $P = a.Q$ evolves by applying the *action prefix* rule of HELENALIGHT in Fig. 6.9 to $a.Q \overset{a}{\hookrightarrow} Q$. Thus, we know that $P = a.Q \overset{a}{\hookrightarrow} Q = P'$.

It remains to show that

$$\pi = trans_{\text{proc}}(a.Q) \overset{\tau^*}{\hookrightarrow} \pi' \overset{trans_{\text{act}}(a)}{\longrightarrow} \pi''$$

entails $\pi'' = trans_{\text{proc}}(Q)$.

With the definition of $trans_{\text{proc}}$ in Fig. 6.17, we have

$$\pi = trans_{\text{proc}}(a.Q) = trans_{\text{act}}(a); trans_{\text{proc}}(Q).$$

To evolve this process expression in PROMELALIGHT, we have to distinguish four types of actions.

Case 2c: $a = X \leftarrow \mathbf{create}(rt_j)$.

- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$\begin{aligned} \pi &= trans_{\text{act}}(a); trans_{\text{proc}}(Q) \\ &= \mathbf{run}\ rt_j(X); trans_{\text{proc}}(Q). \end{aligned}$$

- The process expression π cannot evolve by τ actions according to the semantic rules of PromelaLight in Fig. 6.13. Therefore, we have

$$\pi = \pi' = \mathbf{run}\ rt_j(X); trans_{\text{proc}}(Q).$$

- By applying the *sequential composition* rule of PromelaLight in Fig. 6.13 again, the process expression evolves to

$$\pi' \xrightarrow{\mathbf{run}\ rt_j(X)} trans_{\text{proc}}(Q).$$

- By using the notation $trans_{\text{act}}(a)$, we get

$$\pi' \xrightarrow{trans_{\text{act}}(X \leftarrow \mathbf{create}(rt_j))} trans_{\text{proc}}(Q).$$

Thus, it holds that $\boxed{\pi'' = trans_{\text{proc}}(Q).}$

Case 2b: $a = Y!msgnm(X)$.

- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$\pi = trans_{\text{act}}(a); trans_{\text{proc}}(Q) = Y!msgnm, X; trans_{\text{proc}}(Q).$$

- The process expression π cannot evolve by τ actions according to the semantic rules of PromelaLight in Fig. 6.13. Therefore, we have

$$\pi = \pi' = trans_{\text{proc}}(a.Q) = Y!msgnm, X; trans_{\text{proc}}(Q).$$

- By applying the *sequential composition* rule of PromelaLight in Fig. 6.13, the process expression evolves to

$$\pi' \xrightarrow{Y!msgnm,X} trans_{\text{proc}}(Q)$$

- By using the notation $trans_{\text{act}}(a)$, we get

$$\pi' \xrightarrow{trans_{\text{act}}(Y!msgnm(X))} trans_{\text{proc}}(Q).$$

Thus, it holds that $\boxed{\pi'' = trans_{\text{proc}}(Q).}$

Case 2c: $a = ?msgnm(X{:}rt_j)$.

- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$\begin{aligned} \pi &= trans_{\text{act}}(a); trans_{\text{proc}}(Q) \\ &= \mathbf{self}?msgnm, X; trans_{\text{proc}}(Q). \end{aligned}$$

- The process expression π cannot evolve by τ actions according to the semantic rules of PROMELALIGHT in Fig. 6.13. Therefore, we have

$$\pi = \pi' = \textbf{self}?msgnm, X; trans_{\text{proc}}(Q).$$

- By applying the *sequential composition* rule of PROMELALIGHT in Fig. 6.13 again, the process expression evolves to

$$\pi' \xhookrightarrow{\textbf{self}?msgnm,X} trans_{\text{proc}}(Q).$$

- By using the notation $trans_{\text{act}}(a)$, we get

$$\pi' \xhookrightarrow{trans_{\text{act}}(?msgnm(X:rt))} trans_{\text{proc}}(Q).$$

Thus, it holds that $\boxed{\pi'' = trans_{\text{proc}}(Q).}$

Case 2d: $a = label$.

- Due to the assumption that such that no role behavior in the HELENA specification contains a start state label, we can assume that $label \neq \text{start}_{rt}$.
- With the definition of $trans_{\text{act}}$ in Fig. 6.17, we have

$$\pi = trans_{\text{act}}(a); trans_{\text{proc}}(Q) = label : \textbf{true}; trans_{\text{proc}}(Q).$$

- Since $label \neq \text{start}_{rt}$, the process expression π cannot evolve by τ actions according to the semantic rules of PROMELALIGHT in Fig. 6.13. Therefore, we have

$$\pi = \pi' = label : \textbf{true}; trans_{\text{proc}}(Q).$$

- By applying the *sequential composition* rule of PROMELALIGHT in Fig. 6.13, the process expression evolves to

$$\pi' \xhookrightarrow{label:\textbf{true}} trans_{\text{proc}}(Q)$$

- By using the notation $trans_{\text{act}}(a)$, we get

$$\pi' \xhookrightarrow{trans_{\text{act}}(label)} trans_{\text{proc}}(Q).$$

Thus, it holds that $\boxed{\pi'' = trans_{\text{proc}}(Q).}$

Case 3: $P = P_1 + P_2$, i.e., $\pi = trans_{\text{proc}}(P) = trans_{\text{proc}}(P_1 + P_2)$.
We assume that

$$\pi \xhookrightarrow{\tau^*} \pi' \xhookrightarrow{trans_{\text{act}}(a)} \pi'' \text{ and } \pi = trans_{\text{proc}}(P_1 + P_2).$$

Thus, we show that $\boxed{P_1 + P_2 \xhookrightarrow{a} P' \text{ and } \pi'' = trans_{\text{proc}}(P').}$

With the definition of $trans_{\text{proc}}$ in Fig. 6.17, we have

$$\pi = trans_{\text{proc}}(P_1 + P_2) = \textbf{if} :: trans_{\text{proc}}(P_1) :: trans_{\text{proc}}(P_2)\ \textbf{fi}.$$

Therefore, the two rules *nondet. choice-1* or *nondet. choice-2* of PROMELALIGHT in Fig. 6.13 might be applicable.

- We start by assuming that the *nondet. choice-1* rule is applicable. Since we assumed that

$$\pi \overset{\tau^*}{\hookrightarrow} \pi' \xhookrightarrow{trans_{\text{act}}(a)} \pi'' \text{ and } \pi = trans_{\text{proc}}(P_1 + P_2),$$

  we know from the premise of the *nondet. choice-1* rule that

$$trans_{\text{proc}}(P_1) \overset{\tau^*}{\hookrightarrow} \pi' \xhookrightarrow{trans_{\text{act}}(a)} \pi''.$$

- Since $trans_{\text{proc}}(P_1)$ is structurally smaller than $trans_{\text{proc}}(P_1 + P_2)$ and $trans_{\text{proc}}(P_1) \overset{\tau^*}{\hookrightarrow} \pi' \xhookrightarrow{trans_{\text{act}}(a)} \pi''$, we can assume by induction that $P_1 \overset{a}{\hookrightarrow} P'$ and $\boxed{\pi'' = trans_{\text{proc}}(P')}$.
- Furthermore, with $P_1 \overset{a}{\hookrightarrow} P'$ as a premise, we can evolve the process expression $P_1 + P_2$ by applying the *nondet. choice-1* rule of HelenaLight in Fig. 6.9 to $\boxed{P_1 + P_2 \overset{a}{\hookrightarrow} P'}$.

- On the other hand, the *nondet. choice-2* rule might be applicable: Since we assumed that

$$\pi \overset{\tau^*}{\hookrightarrow} \pi' \xhookrightarrow{trans_{\text{act}}(a)} \pi'' \text{ and } \pi = trans_{\text{proc}}(P_1 + P_2),$$

  we know from the premise of the *nondet. choice-2* rule that

$$trans_{\text{proc}}(P_2) \overset{\tau^*}{\hookrightarrow} \pi' \xhookrightarrow{trans_{\text{act}}(a)} \pi''.$$

- Since $trans_{\text{proc}}(P_2)$ is structurally smaller than $trans_{\text{proc}}(P_1 + P_2)$ and $trans_{\text{proc}}(P_2) \overset{\tau^*}{\hookrightarrow} \pi' \xhookrightarrow{trans_{\text{act}}(a)} \pi''$, we can assume by induction that $P_2 \overset{a}{\hookrightarrow} P'$ and $\boxed{\pi'' = trans_{\text{proc}}(P')}$.
- Furthermore, with $P_2 \overset{a}{\hookrightarrow} P'$ as a premise, we can evolve the process expression $P_1 + P_2$ by applying the *nondet. choice-2* rule of HelenaLight in Fig. 6.9 to $\boxed{P_1 + P_2 \overset{a}{\hookrightarrow} P'}$.

Case 4: $P = N$, i.e., $\pi = trans_{\text{proc}}(P) = trans_{\text{proc}}(N)$.
We assume that

$$\pi \overset{\tau^*}{\hookrightarrow} \pi' \xhookrightarrow{trans_{\text{act}}(a)} \pi'' \text{ and } \pi = trans_{\text{proc}}(N).$$

Thus, we have to show that $\boxed{N \overset{a}{\hookrightarrow} P' \text{ and } \pi'' = trans_{\text{proc}}(P')}$.

- With the definition of $trans_{\text{proc}}$ in Fig. 6.17, we have

$$\pi = trans_{\text{proc}}(N) = \textbf{goto } \text{start}_N.$$

- Since we assumed $\pi \overset{\tau^*}{\hookrightarrow} \pi' \xhookrightarrow{trans_{\text{act}}(a)} \pi''$, we know that this expression can evolve. The only possibility according to the rules of PROMELALIGHT in Fig. 6.13 is by applying the *goto* rule which evolves the expression to

$$\textbf{goto}\ \text{start}_N \overset{\tau}{\hookrightarrow} \text{start}_N : \textbf{true}; seq$$

and we know that the following side-condition holds:

$$\textbf{proctype}\ N(\textbf{chan self})\{seq_1; \text{start}_N : \textbf{true}; seq_2\}.$$

- Since we consider all process expressions in the context of a HELENALIGHT specification and its PROMELALIGHT translation, we know that the process type N in PROMELALIGHT must be the translation of a corresponding HELENALIGHT role behavior **roleBehavior** $N = Q$ which does not contain a state label start_N and therefore

$$\begin{array}{l}\textbf{proctype}\ N(\textbf{chan self})\ \{\\ \quad chandecls_{\text{proc}}(Q)\ \text{start}_N : \textbf{true}; trans_{\text{proc}}(Q)\ \},\end{array}$$

  i.e., $seq_1 = chandecls_{\text{proc}}(Q)$ and $seq_2 = trans_{\text{proc}}(Q)$.
- The expression $\text{start}_N : \textbf{true}; seq_2 = \text{start}_N : \textbf{true}; trans_{\text{proc}}(Q)$ can again be evolved by the *sequential composition* rule of PROMELALIGHT in Fig. 6.13 to

$$\text{start}_N : \textbf{true}; trans_{\text{proc}}(Q) \overset{\tau}{\hookrightarrow} trans_{\text{proc}}(Q).$$

- Since we assumed that

$$trans_{\text{proc}}(N) \overset{\tau^*}{\hookrightarrow} \pi' \xhookrightarrow{trans_{\text{act}}(a)} \pi''$$

  and $trans_{\text{proc}}(N) \overset{\tau^*}{\hookrightarrow} trans_{\text{proc}}(Q)$, it must hold that

$$trans_{\text{proc}}(Q) \overset{\tau^*}{\hookrightarrow} \pi' \xhookrightarrow{trans_{\text{act}}(a)} \pi''.$$

- Since $trans_{\text{proc}}(Q)$ is structurally smaller than $trans_{\text{proc}}(N)$, we can assume by induction that $Q \overset{a}{\hookrightarrow} P'$ and $\boxed{\pi'' = trans_{\text{proc}}(P').}$
- Furthermore, with $Q \overset{a}{\hookrightarrow} P'$ as a premise and **roleBehavior** $N = Q$ as side-condition, we can evolve the process expression N by applying the *process invocation* rule of HELENALIGHT in Fig. 6.9 to $\boxed{N \overset{a}{\hookrightarrow} P'.}$

Thus, the structure of P was completely covered. ■

Proof of Lemma A.9

With Lemma A.4 and Lemma A.9 which reason about the evolution by silent actions and non-silent actions, we can finally show that $\approx^{-1}$ is a $\approx^{-1}$-stutter simulation of a PromelaLight specification by the corresponding HelenaLight specification. Interestingly, in the direction from PromelaLight to HelenaLight, it is enough to employ the relation $\approx^{-1}$ itself as a $\approx^{-1}$-stutter simulation since a PromelaLight translation might take more steps than the corresponding HelenaLight ensemble specification, but not the other way around.

Prop. A.10: $\approx^{-1}$-Stutter Simulation of PromelaLight Translations

Let

- $K(T_{\text{HEL}}) = (S_{\text{HEL}}, A_{\text{HEL}}, \rightarrow^{\bullet}_{\text{HEL}}, F_{\text{HEL}})$ *be the induced Kripke structure of a* HelenaLight *ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *with* $\Sigma = (nm, roletypes, roleconstraints)$ *such that no role behavior in behaviors contains a start state label and*
- $K(T_{\text{PRM}}) = (S_{\text{PRM}}, A_{\text{PRM}}, \rightarrow^{\bullet}_{\text{PRM}}, F_{\text{PRM}})$ *be the induced Kripke structure of its* PromelaLight *translation* $trans(EnsSpec)$.

$\approx^{-1}$ *is a* $\approx^{-1}$*-stutter simulation of* $K(T_{\text{PRM}})$ *by* $K(T_{\text{HEL}})$.

Proof of Prop. A.10

We have to show that the relation $\approx^{-1}$ satisfies the property of a $\approx^{-1}$-stutter simulation described in Def. 6.2. In the proof, we rely on the underlying labeled transition systems T_{HEL} and T_{PRM} of $K(T_{\text{HEL}})$ and $K(T_{\text{PRM}})$. We show the following property which obviously entails the required property for a $\approx^{-1}$-stutter simulation:
For all $\sigma \in S_{\text{HEL}}, \gamma \in S_{\text{PRM}}$ with $\gamma \approx^{-1} \sigma$,

(1) if there exists $\gamma' \in S_{\text{PRM}}$ such that $\gamma \xrightarrow{\tau}_{\text{PRM}} \gamma'$, then $\gamma' \approx^{-1} \sigma$ holds,

(2) if there exists $\gamma' \in S_{\text{PRM}}, b \in A_{\text{PRM}}$ such that $\gamma \xrightarrow{b}_{\text{PRM}} \gamma'$ and $b \neq \tau$, then there exist $\sigma' \in S_{\text{HEL}}, a \in A_{\text{HEL}}$ such that $\sigma \xrightarrow{a}_{\text{HEL}} \sigma'$, $trans_{\text{act-global}}(a) = b$, and $\gamma' \approx^{-1} \sigma'$.

Item (1) directly follows from Lemma A.4.
The proof of item (2) has to consider all four types of global actions possible in the transition $\gamma \xrightarrow{b}_{\text{PRM}} \gamma'$ which are not silent, i.e., $b \neq \tau$. We assume that $\gamma = (\mathsf{ch}, \mathsf{proc}) \approx^{-1} \sigma$ and $\gamma \xrightarrow{b}_{\text{PRM}} \gamma'$.

Case 1: $b = i : \textbf{run}\ pt_j(var)$ for $i \in dom(\mathsf{proc})$.
We know from the notation in Sec. 6.5.1 that

$$trans_{\text{act-global}}(i : var \leftarrow \textbf{create}(pt_j)) = i : \textbf{run}\ pt_j(var).$$

Thus, we have to show

that there exists $\sigma' \in S_{\text{HEL}}$ such that

(1) $\sigma \xrightarrow{i:var \leftarrow \mathbf{create}(pt_j)}_{\text{HEL}} \sigma'$, and

(2) $\gamma' \approx^{-1} \sigma'$.

To this end, we can assume that the premise and all side-conditions of the *run* rule of the PROMELALIGHT semantics in Fig. 6.14 are satisfied by γ since we assumed $\gamma \xrightarrow{b}_{\text{PRM}} \gamma'$.

Firstly, we show that the premise of the *create* rule in HELENALIGHT in Fig. 6.10 is satisfied.

- From the premise of the *run* rule in PROMELALIGHT in Fig. 6.14, we know that the premise $\pi_i \xrightarrow{\mathbf{run}\ pt_j(var)} \pi_i'$ holds.
- From the notation in Sec. 6.5.1, we know that

$$trans_{\text{act}}(var \leftarrow \mathbf{create}(pt_j)) = \mathbf{run}\ pt_j(var).$$

- Since $\gamma \approx^{-1} \sigma$, we know that

$$trans_{\text{proc}}(P_i) \xrightarrow{\tau^*} \pi_i \text{ or}$$
$$chandecls_{\text{proc}}(P_i)\ \text{start}_{pt_i} : \mathbf{true}; trans_{\text{proc}}(P_i) \xrightarrow{\tau^*} \pi_i.$$

  However, we can simplify the second option to $\pi = trans_{\text{proc}}(P)$ since channel declarations and the start state label cannot evolve by the action $\mathbf{run}\ pt_j(var)$ as required by $\gamma \xrightarrow{b}_{\text{PRM}} \gamma'$. Furthermore, this simplified option $\pi = trans_{\text{proc}}(P)$ is subsumed by the first option $trans_{\text{proc}}(P) \xrightarrow{\tau^*} \pi$ and can therefore be omitted.
- Thus, we can apply Lemma A.9 and can conclude that the premise of the *create* rule in HELENALIGHT in Fig. 6.10 is satisfied by $P_i \xrightarrow{var \leftarrow \mathbf{create}(pt_j)} P_i'$ and $\pi_i' = trans_{\text{proc}}(P_i')$.

Secondly, we show that all side-conditions of the *create* rule in HELENALIGHT in Fig. 6.10 are satisfied.

- From the *run* rule in PROMELALIGHT in Fig. 6.14, we know that $i \in dom(\mathtt{proc})$ with $\mathtt{proc}(i) = (pt_i, \beta_i, \pi_i)$.
- Thus, since $\gamma \approx^{-1} \sigma$, we can conclude that the item (1) of the side-condition of the *create* rule in HELENALIGHT in Fig. 6.10 is satisfied by $i \in dom(\sigma)$ with $\sigma(i) = (pt_i, v_i, q_i, P_i)$.
- From the *run* rule in PROMELALIGHT in Fig. 6.14, we know that

$$\mathbf{proctype}\ pt_j(\mathbf{chan\ self})\{seq_1; \text{start}_{pt_j} : \mathbf{true}; seq_2\}.$$

- Since we know that $trans(EnsSpec)$ contains $trans_{\text{role}}(b)$ for every role behavior b in *EnsSpec*, we can assume that there exists **roleBehavior** $pt_j = P_j$ such that $chandecls_{\text{proc}}(P) = seq_1$ and $trans_{\text{proc}}(P_j) = seq_2$ which satisfies the item (2) of the side-condition of the *create* rule in HelenaLight in Fig. 6.10.

Thus, the premise and all side-conditions for applying the *create* rule in HelenaLight in Fig. 6.10 are satisfied by σ and we can conclude that $\sigma \xrightarrow{i:var \leftarrow \mathbf{create}(pt_j)}_{\text{HEL}} \sigma'$ such that

$$\begin{aligned}\sigma' \quad &= \sigma[i \mapsto (pt_i, v_i[var \mapsto next(\sigma)], q_i, P_i')] \\ &\quad [next(\sigma) \mapsto (pt_j, \emptyset[\mathbf{self} \mapsto next(\sigma)], \varepsilon, P_j)]\end{aligned}$$

with $trans_{\text{proc}}(P_i') = \pi_i'$ and $trans_{\text{proc}}(P_j) = seq$.

Lastly, we have to show that $\gamma' \approx^{-1} \sigma'$. From the *run* rule in PromelaLight in Fig. 6.14, we know that

$$\begin{aligned}\gamma' \quad &= (\mathsf{ch}, \mathsf{proc}[i \mapsto (pt_i, \beta_i, \pi_i')] \\ &\quad [next(\mathsf{proc}) \mapsto (pt_j, \emptyset[\mathbf{self} \mapsto c], \text{start}_{pt_j} : \mathbf{true}; seq)])\end{aligned}$$

where we know from the side-condition of the *run* rule in PromelaLight in Fig. 6.14 that $\beta_i(var) = c \in dom(\mathsf{ch})$. Thus, it obviously holds that $\sigma' \approx \gamma'$.

Case 2: $b = i : var_1!const, var_2$ for $i \in dom(\mathsf{proc})$.
We know from the notation in Sec. 6.5.1 that

$$trans_{\text{act-global}}(i : var_1!const(var_2)) = i : var_1!const, var_2.$$

Thus, we have to show

that there exists $\sigma' \in S_{\text{HEL}}$ such that

(1) $\sigma \xrightarrow{i:var_1!const(var_2)}_{\text{HEL}} \sigma'$, and

(2) $\gamma' \approx^{-1} \sigma'$.

To this end, we can assume that the premise and all side-conditions of the *send* rule of the PromelaLight semantics in Fig. 6.14 are satisfied by γ since we assumed $\gamma \xrightarrow{b}_{\text{PRM}} \gamma'$.

Firstly, we show that the premise of the *send* rule in HelenaLight in Fig. 6.10 is satisfied. The proof proceeds analog to Case 1 and we know that $P_i \xrightarrow{var_1!const(var_2)} P_i'$ and $\pi_i' = trans_{\text{proc}}(P_i')$.

Secondly, we show that all side-conditions of the *send* rule in HelenaLight in Fig. 6.10 are satisfied.

- From the side-conditions of the *send* rule in PromelaLight in Fig. 6.14, we know that $i \in dom(\mathsf{proc})$ with $\mathsf{proc}(i) = (pt_i, \beta_i, \pi_i)$.

- Thus, since $\gamma' \approx^{-1} \sigma$, we can conclude that the item (1) of the side-condition of the *send* rule in HELENALIGHT in Fig. 6.10 is satisfied by $i \in dom(\sigma)$ with $\sigma(i) = (pt_i, v_i, q_i, P_i)$.
- From the *send* rule in PROMELALIGHT in Fig. 6.14, we know that $\beta_i(var_1) = c \in dom(\mathsf{ch})$ with $\mathsf{ch}(c) = (T, \kappa, \omega)$ and $|\omega| < \kappa$.
- Thus, since $\gamma \approx^{-1} \sigma$, we can conclude that the item (2) of the side-condition of the *send* rule in HELENALIGHT in Fig. 6.10 is satisfied by $v_i(var_1) = c \in dom(\sigma)$ with $\sigma(c) = (rt_c, v_c, q_c, P_c)$ and $|q_c| < \kappa = roleconstraints(rt_c)$.
- From the *send* rule in PROMELALIGHT in Fig. 6.14, we know that $\beta_i(var_2) = v \in dom(\mathsf{ch})$.
- Thus, since $\gamma' \approx^{-1} \sigma$, we can conclude that the item (3) of the side-condition of the *send* rule in HELENALIGHT in Fig. 6.10 is satisfied by $v_i(var_2) = v \in dom(\sigma)$.

Thus, the premise and all side-conditions for applying the *send* rule in HELENALIGHT in Fig. 6.10 are satisfied by σ and we can conclude that $\sigma \xrightarrow{i:var_1!const(var_2)}_{\text{HEL}} \sigma'$ with

$$\begin{aligned} \sigma' \quad &= \sigma[i \mapsto (pt_i, v_i, q_i, P_i')] \\ &\quad [c \mapsto (rt_c, v_c, q_c \cdot const(v), P_c)] \end{aligned}$$

and $trans_{\text{proc}}(P_i') = \pi_i'$.

Lastly, we have to show that $\gamma' \approx^{-1} \sigma'$. From the *send* rule in PROMELALIGHT in Fig. 6.14, we know that

$$\gamma' = (\mathsf{ch}[c \mapsto (T, \kappa, \omega \cdot (const, v))], \mathsf{proc}[i \mapsto (pt_i, \beta_i, \pi_i')]).$$

Thus, it obviously holds that $\gamma' \approx^{-1} \sigma'$.

Case 3: $b = i : var_1?const, var_2$ for $i \in dom(\mathsf{proc})$.
Since we consider the labeled transition system of the PROMELALIGHT translation $trans(EnsSpec)$ only (i.e., not an arbitrarily formed PROMELALIGHT specification), we know from the translation function $trans_{\text{act}}$ in Fig. 6.17 that b can only be of the form $i : \textbf{self}?const, var_2$.

Furthermore, we know from the notation in Sec. 6.5.1 that

$$trans_{\text{act-global}}(i :?const(pvar_2{:}rt_v)) = i : \textbf{self}?const, var_2.$$

Thus, we have to show

that there exists $\sigma' \in S_{\text{HEL}}$ such that

(1) $\sigma \xrightarrow{i:?const(pvar_2:rt_v)}_{\text{HEL}} \sigma'$, and

(2) $\gamma' \approx^{-1} \sigma'$.

To this end, we can assume that the premise and all side-conditions of the *receive* rule of the PromelaLight semantics in Fig. 6.14 are satisfied by γ since we assumed $\gamma \xrightarrow{b}_{\text{PRM}} \gamma'$.

Firstly, we show that the premise of the *receive* rule in HelenaLight in Fig. 6.10 is satisfied. The proof proceeds analog to Case 1 and we know that $P_i \xrightarrow{?const(pvar_2:rt_v)} P_i'$ and $\pi_i' = trans_{\text{proc}}(P_i')$.

Secondly, we show that all side-conditions of the *receive* rule in HelenaLight in Fig. 6.10 are satisfied.

- From the *receive* rule in PromelaLight in Fig. 6.14, we know that $i \in dom(\texttt{proc})$ with $\texttt{proc}(i) = (pt_i, \beta_i, \pi_i)$ and $\beta_i(\textbf{self}) = c \in dom(\texttt{ch})$ with $\texttt{ch}(c) = (T, \kappa, (const, v) \cdot \omega)$ and $v \in dom(\texttt{ch})$.
- Thus, since $\gamma \approx^{-1} \sigma$, we can conclude that $i \in dom(\sigma)$ with $\sigma(i) = (pt_i, v_i, const(v) \cdot q_i, P_i)$ and $v \in dom(\sigma)$.

Thus, the premise and all side-conditions for applying the *receive* rule in HelenaLight in Fig. 6.10 are satisfied by σ and we can conclude that $\sigma \xrightarrow{i:?const(pvar_2:rt_v)}_{\text{HEL}} \sigma'$ with

$$\sigma' = \sigma[i \mapsto (pt_i, v_i[var_2 \mapsto v], q_i, P_i')]$$

and $trans_{\text{proc}}(P_i') = \pi_i'$.

Lastly, we have to show that $\gamma' \approx^{-1} \sigma'$. From the *send* rule in PromelaLight in Fig. 6.14, we know that

$$\gamma' = (\texttt{ch}[c \mapsto (T, \kappa, \omega)], \texttt{proc}[i \mapsto (pt_i, \beta_i[var_2 \mapsto v], \pi_i')]).$$

Thus, it obviously holds that $\gamma' \approx^{-1} \sigma'$.

Case 4: $b = i : label : \textbf{true}$,

Since we only consider $b \neq \tau$, it holds that $b \neq i : \text{start}_{pt_j} : \textbf{true}$ for $i, j \in dom(\texttt{proc})$.

We know from the notation in Sec. 6.5.1 that

$$trans_{\text{act-global}}(i : label) = i : label : \textbf{true}.$$

Thus, we have to show

that there exists $\sigma' \in S_{\text{HEL}}$ such that

(1) $\sigma \xrightarrow{i:label}_{\text{HEL}} \sigma'$, and

(2) $\gamma' \approx^{-1} \sigma'$.

To this end, we can assume that the premise and all side-conditions of the *label* rule of the PromelaLight semantics in Fig. 6.15 are satisfied by γ since we assumed $\gamma \xrightarrow{b}_{\text{PRM}} \gamma'$.

Firstly, we show that the premise of the *label* rule in HelenaLight in Fig. 6.10 is satisfied. The proof proceeds analog to Case 1 and we know that $P_i \xhookrightarrow{label} P_i'$ and $trans_{\text{proc}}(P_i') = \pi_i'$.

Secondly, we show that all side-conditions of the *label* rule in HelenaLight in Fig. 6.10 are satisfied.

- From the *label* rule in PromelaLight in Fig. 6.15, we know that $i \in dom(\mathsf{proc})$ with $\mathsf{proc}(i) = (pt_i, \beta_i, \pi_i)$.
- Thus, since $\gamma \approx^{-1} \sigma$, we can conclude that the side-condition of the *label* rule in HelenaLight is satisfied by $i \in dom(\sigma)$ with $\sigma(i) = (pt_i, v_i, q_i, P_i)$.

Thus, the premise and all side-conditions for applying the *label* rule in HelenaLight in Fig. 6.10 are satisfied by σ and we can conclude that $\sigma \xrightarrow{i:label}_{\text{Hel}} \sigma'$ with $\sigma' = \sigma[i \mapsto (pt_i, v_i, q_i, P_i')]$ and $trans_{\text{proc}}(P_i') = \pi_i'$.

Lastly, we have to show that $\gamma' \approx^{-1} \sigma'$. From the *label* rule in PromelaLight in Fig. 6.15, we know that

$$\gamma' = (\mathsf{ch}, \mathsf{proc}[i \mapsto (pt_i, \beta_i, \pi_i')]).$$

Thus, it obviously holds that $\gamma' \approx^{-1} \sigma'$.

Thus, all four types of global actions possible in the transition $\gamma \xrightarrow{b}_{\text{Prm}} \gamma'$ where $b \neq \tau$ were covered. ■

Proof of Prop. A.10

A.6 Stutter Trace Equivalence of HelenaLight and PromelaLight

In the last sections, we showed that

- the relation $\approx$ preserves satisfaction of atomic propositions in Prop. A.2,
- the relation $\approx$ is divergence-sensitive in Prop. A.5,
- any admissible initial state of a HelenaLight ensemble specification and its PromelaLight translation are related by the relation $\sim$ in Prop. A.6,
- the relation $\sim$ is a $\approx$-stutter simulation of the Kripke structure of the HelenaLight specification by the Kripke structure of the PromelaLight translation in Prop. A.8, and
- the inverse relation $\approx^{-1}$ is a $\approx^{-1}$-stutter simulation in the other direction in Prop. A.10.

Therefore, we can directly infer by Def. 6.2 and Thm. 6.5 that the induced Kripke structures of a HelenaLight ensemble specification and its PromelaLight translation are stutter trace equivalent. Note that we have to make three assumptions to be able to apply Prop. A.2, Prop. A.5, Prop. A.8, and Prop. A.10: role behavior declarations may not contain a start state label, role behavior declarations in HelenaLight may not start with a state label and in any nondeterministic choice construct of a role

behavior, process invocation may not be one of the branches. Furthermore, in the translation, we make two assumptions which can be alleviated in future work: each role type can only be adopted by a single component type and we only verify a single ensemble instance per translation.

Thm. A.11: Stutter Trace Equivalence of HelenaLight and PromelaLight

Let

- $K(T_{\text{HEL}}) = (S_{\text{HEL}}, A_{\text{HEL}}, \rightarrow^{\bullet}_{\text{HEL}}, F_{\text{HEL}})$ *be the induced Kripke structure of a* HelenaLight *ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *with* $\Sigma = (nm, roletypes, roleconstraints)$ *such that*
  - *no role behavior in behaviors contains a start state label* $start_{rt}$,
  - *no role behavior in behaviors starts with a state label, and*
  - *in any nondeterministic choice construct in a role behavior in behaviors, process invocation is not one of the branches, and*
- $K(T_{\text{PRM}}) = (S_{\text{PRM}}, A_{\text{PRM}}, \rightarrow^{\bullet}_{\text{PRM}}, F_{\text{PRM}})$ *be the induced Kripke structure of its* PromelaLight *translation* $trans(EnsSpec)$.

Then $K(T_{\text{HEL}})$ *and* $K(T_{\text{PRM}})$ *are stutter trace equivalent.*

Proof of Thm. A.11

We showed in Prop. A.2 that the relation $\approx$ is property-preserving and in Prop. A.5 that it is divergence-sensitive. It only remains to show that $K(T_{\text{HEL}})$ is $\approx$-stutter simulated by $K(T_{\text{PRM}})$ and vice versa to be able to apply Thm. 6.5. To this end, we have already proven that the relation $\sim$ is a $\approx$-stutter simulation of the Kripke structure of the HelenaLight specification by the Kripke structure of the PromelaLight translation in Prop. A.8 and that any admissible initial state of a HelenaLight ensemble specification and its PromelaLight translation are related by the relation $\sim$ in Prop. A.6. Therefore, we can apply Def. 6.2 and deduce that $K(T_{\text{HEL}})$ is $\approx$-stutter simulated by $K(T_{\text{PRM}})$. In the other direction, we have already proven that the inverse relation $\approx^{-1}$ is a $\approx^{-1}$-stutter simulation of the Kripke structure of the PromelaLight translation by the Kripke structure of its original HelenaLight specification. Thus, it remains to show that the initial states of $K(T_{\text{HEL}})$ and $K(T_{\text{PRM}})$ are in the relation $\approx$ (which entails that they are also in the relation $\approx^{-1}$). With Prop. A.6, we proved that any initial state of a HelenaLight specification and its PromelaLight translation are in the relation $\sim$. Since $\sim \subseteq \approx$, we can conclude that any initial state of a HelenaLight specification and its PromelaLight translation are also in the relation $\approx$ (and therefore vice versa in the relation $\approx^{-1}$). Therefore, we can apply Def. 6.2 and deduce that $K(T_{\text{PRM}})$ is $\approx^{-1}$-stutter simulated by $K(T_{\text{HEL}})$. Together, all conditions of Thm. 6.5 are satisfied and therefore we can conclude that $K(T_{\text{HEL}})$ and $K(T_{\text{PRM}})$ are stutter trace equivalent. ■

At this point, we want to mention that the relation $\approx$ could be proven as a $\approx$-stutter simulation in both directions. It could even be shown that the relation $\approx$ is a weak bisimulation. However, for the lack of understandability we relied on the stronger relation $\sim$ in the proof.

On the result from Thm. A.11, we can furthermore apply Thm. 6.6 to show that both Kripke structures satisfy the same $LTL_{\setminus \mathbf{X}}$ formulae.

Cor. A.12: HelenaLight $LTL_{\setminus \mathrm{X}}$ Preservation

Let

- $K(T_{\mathrm{HEL}}) = (S_{\mathrm{HEL}}, A_{\mathrm{HEL}}, \rightarrow^{\bullet}_{\mathrm{HEL}}, F_{\mathrm{HEL}})$ *be the induced Kripke structure of a* HelenaLight *ensemble specification* $EnsSpec = (\Sigma, behaviors)$ *with* $\Sigma = (nm, roletypes, roleconstraints)$ *such that*
  - *no role behavior in behaviors contains a start state label* $start_{rt}$,
  - *no role behavior in behaviors starts with a state label, and*
  - *in any nondeterministic choice construct in a role behavior in behaviors, process invocation is not one of the branches, and*
- $K(T_{\mathrm{PRM}}) = (S_{\mathrm{PRM}}, A_{\mathrm{PRM}}, \rightarrow^{\bullet}_{\mathrm{PRM}}, F_{\mathrm{PRM}})$ *be the induced Kripke structure of its* PromelaLight *translation* $trans(EnsSpec)$.

For any $LTL_{\setminus \mathbf{X}}$ *formula* ϕ *over* $AP(EnsSpec)$, $K(T_{\mathrm{HEL}}) \models \phi \Leftrightarrow K(T_{\mathrm{PRM}}) \models \phi$.

Appendix B

HELENA Workbench

The HELENA workbench provides tool support for the development of ensemble-based systems with HELENA relying on Eclipse. A custom editor in Eclipse allows to define ensemble specifications using the domain-specific language HELENATEXT (cf. Sec. 8.2). The editor provides syntax highlighting, content assist as well as validation of the specification. Furthermore, two generators take the HELENATEXT specification as input to generate an semantically equivalent PROMELA verification model for model-checking with Spin (cf. Chap. 4 and Sec. 8.3) and to generate Java code which can be executed relying on the jHELENA framework (cf. Chap. 7 and Sec. 8.4). This appendix explains in Appendix B.1 how the HELENA workbench can further be developed to include more features and in Appendix B.2 how it can be used to define ensemble specifications.

The complete implementation of the HELENA workbench can be found on the attached CD in the projects `eu.ascens.helenaText`, `eu.ascens.helenaText.sdk`, `eu.ascens.helenaText.tests`, and `eu.ascens.helenaText.ui`. The p2p example is exercised with the HELENA workbench in the project `eu.ascens.helenaText.p2p`.

B.1 User-Guide for Developing the HELENA Workbench

The following sections explain how the HELENA workbench is developed.

B.1.1 Setting-up an XTEXT Project in Eclipse

Initially, an XTEXT project must be set up to develop the HELENA workbench. The following steps were necessary (tested with Eclipse Neon 4.6.0):

- install XTEXT plug-in in the Eclipse installation
  (Xtext 2.9.1 from Eclipse Marketplace)
- install ATL plug-in in the Eclipse installation, it might be necessary to uninstall any previous version of ATL
  (ATL/EMFTVM 3.8.0 from Eclipse Marketplace)
- create new XTEXT project and finish with the default settings → four projects are created (`language`, `sdk`, `tests`, `ui`)

In the generated `.xtext`-file, the XTEXT grammar for HELENATEXT is defined. To generate the HELENATEXT editor as an Eclipse plug-in, this grammar needs to be run such that the plug-in code is generated. To initiate the generation, right-click on the `.xtext`-file and select 'Run As' → 'Generate Xtext Artifacts'. Whenever changes in the grammar are made, the generation of XTEXT artifacts has to be initiated again (for more

information see the tutorial on `http://www.eclipse.org/Xtext/documentation.html#FirstFiveMinutes`).

Afterwards the plug-in can be used either by exporting the plug-in (cf. Appendix B.1.3) or by launching a new Eclipse instance. To do the latter, right-click on the `language`-project and select 'Run As' → Eclipse Application. A new Eclipse instance is launched which contains the HELENATEXT plug-in. The usage of the HELENA workbench and its HELENATEXT editor is explained in Appendix B.2.

B.1.2 Projects and Packages of the HELENA Workbench

The HELENA workbench consists of three projects:

- The project `eu.ascens.helenaText` contains the definition of the grammar of the domain-specific language HELENATEXT, formatting and validation for the generated HELENATEXT editor, and both generators to PROMELA and to jHELENA.
  - The package `eu.ascens` contains the grammar file `HelenaText.xtext` for HELENATEXT and the workflow `GenerateHelenaText.mwe2` to create the Eclipse editor.
  - The package `eu.ascens.validation` contains validation rules which check conformance of the HELENATEXT specification with the formal HELENA definition. Validation checks are executed while typing in the HELENATEXT editor.
  - The package `eu.ascens.scoping` defines the scoping context of certain entities of the HELENATEXT grammar, e.g., all variables in the scope of a role behavior.
  - The package `eu.ascens.formatting` contains rules for formatting a HELENATEXT specification automatically. Formatting is triggered in the HELENATEXT editor by pressing CTRL+SHIFT+F.
  - The package `eu.ascens.generator` contains all generation rules for the translation from HELENATEXT to PROMELA and to jHELENA. Generation is triggered in the HELENATEXT editor whenever a file is saved.
- The project `eu.ascens.helenaText.tests` is a project where unit tests for testing the parser, validator etc. can be defined.
- The project `eu.ascens.helenaText.ui` provides the implementation of the HELENATEXT editor. It is mostly generated and can be adapted to make customize the HELENATEXT editor, e.g., to add quick fixes, to change the appearance of the outline, or to provide code proposals.

B.1.3 Exporting the HELENA Workbench as Eclipse Plug-in

To make the HELENA workbench available to users (instead of just launching an internal Eclipse instance), it must be exported as an Eclipse plug-in. The export creates several jar-archives. To use the HELENA workbench, the user has to install the plug-in into his Eclipse installation by copying the jar files into the `plugins` folder of his Eclipse installation.

The HELENA workbench is exported as an Eclipse plug-in by using the Eclipse export functionality which can only be used if the Eclipse Plug-in Development Environment and Tools are installed in the running Eclipse installation. The following steps are necessary to export the HELENA workbench:

- click 'File' → 'Export'
- choose 'Plug-in Development' → 'Deployable plug-ins and fragments'
- select the projects `eu.ascens.helenaText` and `eu.ascens.helenaText.ui`
- set the target directory for the resulting jar files
- click 'Finish'

B.2 User-Guide for Using the HELENA Workbench

To use the HELENATEXT editor, a running Eclipse installation is needed. The installation was tested with Eclipse Neon 4.6.0 with the plug-ins for XTEXT version 2.9.1 and ATL/EMFTVM 3.8.0 installed from Eclipse Marketplace (any previous version of ATL might have to be uninstalled before). To add the HELENA workbench to this Eclipse installation, two approaches are possible: Either import the jar-archives of the HELENA plug-in (cf. Appendix B.1.3) to the `plugins`-folder of the current Eclipse installation. Or launch a new internal Eclipse instance with the HELENA workbench running (cf. Appendix B.1.1).

In the Eclipse installation with the HELENA plug-in, the HELENATEXT editor can be used. For that, create a new Java project in the new Eclipse instance and create a file in the `src`-folder of the new project with the extension of the defined language, i.e., `.helena`. A window will pop-up which asks whether the XTEXT nature should be added to the new project. By confirming this pop-up, the HELENATEXT editor can be used for the new project.

With that, the Java project is ready to be used for the specification of an ensemble-based system with HELENATEXT. The specification has to be written in a `.helena`-file. Such a file is supported by a custom HELENATEXT editor which provides syntax highlighting, content assist and validation. On save, a valid HELENATEXT file is furthermore automatically translated to PROMELA and Java. The generated PROMELA file resides in the folder `promela-gen` and has the same name as the `.helena`, but with the extension `.pml`. This file can be used for verification with Spin (cf. Sec. 5.3). The generated Java code resides in the folders `src-gen` and `src-user`. To be able to run the generated Java code, several steps are necessary. Firstly, the two folders have to be added to the build-path. Secondly, the jHELENA framework has to be added to the build-path (either as a jar-archive or a project). Thirdly, the missing code in the implementation classes of the folder `src-user` has to be implemented (cf. Sec. 8.4).

B.3 Complete HELENATEXT Grammar

The domain-specific language HELENATEXT provides a concrete textual syntax to specify HELENA ensemble specifications. Its grammar is defined relying on the XTEXT workbench of Eclipse. This section lists the complete HELENATEXT grammar written in XTEXT.

```
1  grammar eu.ascens.HelenaText with org.eclipse.xtext.xbase.Xbase
2
3  generate helenaText "http://www.ascens.eu/HelenaText"
4  import 'http://www.eclipse.org/xtext/xbase/Xbase' as xbase
5
6  ////////// MODEL ////////////
7  Model:
```

```
8     headPkg = PackageDeclaration
9   ;
10
11  PackageDeclaration:
12    'package' name=QualifiedName '{'
13        (compTypes += ComponentType)*
14      & (roleTypes += RoleType)*
15      & (ensStructs += EnsembleStructure)*
16      & (roleBehaviors += RoleBehavior)*
17    '}'
18  ;
19
20  ////////// COMPONENT TYPE ////////////
21  ComponentType:
22    'componentType' name=ValidID '{'
23      (
24          attrs += ComponentAttributeType
25        | assocs  += ComponentAssociationType
26        | ops   += OperationType
27      )*
28    '}'
29  ;
30  AbstractComponentFieldType:
31    ComponentAttributeType |
32    ComponentAssociationType
33  ;
34  ComponentAttributeType:
35    'attr' type=JvmTypeReference name=ValidID ';'
36  ;
37  ComponentAssociationType:
38    'assoc' type=[ComponentType] name=ValidID ';'
39  ;
40  OperationType:
41    'op' returnType=JvmTypeReference  name=ValidID formalDataParamsBlock=
          FormalDataParamsBlock ';'
42  ;
43
44  ////////////// ROLE TYPE /////////////////
45  RoleType:
46    'roleType' name=ValidID 'over' compTypes+=[ComponentType](','compTypes+=[
          ComponentType])* '{'
47      (
48          roleattrs +=  RoleAttributeType
49        | rolemsgs  +=  MessageType
50      )*
51    '}'
52  ;
53  RoleAttributeType:
54    'roleattr' type=JvmTypeReference name=ValidID ';'
55  ;
56  MessageType:
57    'rolemsg' direction=MsgDirection name=ValidID formalRoleParamsBlock=
          FormalRoleParamsBlock formalDataParamsBlock=FormalDataParamsBlock ';'
58  ;
59  enum MsgDirection :
60      IN  = 'in'
61    | OUT = 'out'
62    | INOUT = 'in/out'
63  ;
64
65  ////////////// ENSEMBLE STRUCTURE //////////////
66  EnsembleStructure:
67    'ensembleStructure' name=ValidID '{'
68      rtWithMult+=RoleTypeWithMultiplicity (rtWithMult+=RoleTypeWithMultiplicity)*
69    '}'
70  ;
71  RoleTypeWithMultiplicity:
72    '<'
```

```
73      roleType=[RoleType]
74      ',' 
75      'min' '=' min=MultElem
76      ',' 
77      'max' '=' max=MultElem
78      ',' 
79      'cap' '=' capacity=INT
80    '>'
81    ';'
82  ;
83  MultElem:
84    ('*' | INT)
85  ;
86
87  ////////// ROLE BEHAVIORS ////////
88  RoleBehavior:
89      {DeclaringRoleBehavior} 'roleBehavior' roleTypeRef=[RoleType] '=' processExpr=
            ProcessExpression
90    |   {InvokingRoleBehavior} 'roleBehavior' roleTypeRef=[RoleType] '='
          processInvocation=ProcessInvocation
91        '{' (processes+=Process)* '}'
92  ;
93  Process:
94    'process' name=ValidID '=' processExpr=ProcessExpression
95  ;
96
97  ////////// PROCESS TERMS ////////
98  ProcessExpression:
99      {QuitTerm} 'quit'
100   | ActionPrefix
101   | NondeterministicChoice
102   | IfThenElse
103   | ProcessInvocation
104 ;
105 ActionPrefix:
106   action=Action '.' processExpr=ProcessExpression
107 ;
108 NondeterministicChoice:
109   '(' first=ProcessExpression '+' second=ProcessExpression ')'
110 ;
111 IfThenElse:
112   'if' '(' guard=Guard ')' '{' ifProcessExpr=ProcessExpression '}'
113   'else' '{' elseProcessExpr=ProcessExpression '}'
114 ;
115 ProcessInvocation:
116   process=[Process]
117 ;
118
119 /////////// ACTIONS ///////////////
120 Action:
121     AbstractAssignment
122   | AbstractMessageCall
123   | OperationCall
124   | ComponentAttributeSetter
125   | RoleAttributeSetter
126   | Label
127 ;
128 AbstractAssignment:
129     {GetAssignment} roleInst=RoleInstanceVariable '<-' 'get' '(' roleTypeRef=[RoleType]
            ',' compInstance=ComponentInstance ')'
130   | {CreateAssignment} roleInst=RoleInstanceVariable '<-' 'create' '(' roleTypeRef=[
        RoleType] ',' compInstance=ComponentInstance ')'
131 ;
132 AbstractMessageCall:
133     OutgoingMessageCall
134   | IncomingMessageCall
135 ;
136 OutgoingMessageCall:
```

```
137   receiver=AbstractRoleInstanceReference '!'
138     msgName=ValidID
139     actualRoleParamsBlock=ActualRoleParamsBlock
140     actualDataParamsBlock=ActualDataParamsBlock
141 ;
142 IncomingMessageCall:
143   '?'  msgName=ValidID formalRoleParamsBlock=FormalRoleParamsBlock
          formalDataParamsBlock=FormalDataParamsBlock
144 ;
145 OperationCall:
146   (variable=DataVariable '='? 'owner.' opName=ValidID  actualDataParamsBlock=
          ActualDataParamsBlock
147 ;
148 ComponentAttributeSetter:
149   attr=ComponentAttributeTypeReference '=' value=DataExpression
150 ;
151 RoleAttributeSetter:
152   attr=RoleAttributeTypeReference '=' value=DataExpression
153 ;
154 Label:
155   name=ValidID
156 ;
157
158 /////////////// GUARDS ///////////////
159 Guard:
160   OrTerm
161 ;
162 OrTerm returns Guard:
163   AndTerm ({OrTerm.left=current} '||' right=AndTerm )*
164 ;
165 AndTerm returns Guard:
166   EqualityTerm ({AndTerm.left=current} '&&' right=EqualityTerm)*
167 ;
168 EqualityTerm returns Guard:
169   NotTerm ({EqualityTerm.left=current}  operator=OpEquality right=NotTerm)*
170 ;
171 NotTerm returns Guard:
172   {NotTerm}
173   (not='!')? atom=Atom
174 ;
175 Atom:
176     DataExpression
177   | PlaysQuery
178   | Relation
179   | GuardInParentheses
180 ;
181 PlaysQuery:
182   'plays' '(' roleTypeRef=[RoleType] ',' compInstance=ComponentInstance ')'
183 ;
184 Relation:
185   left=DataExpression operator=OpCompare right=DataExpression
186 ;
187 GuardInParentheses:
188   '(' guard=Guard ')'
189 ;
190 AbstractDataValue:
191   BooleanValue|
192   NumberValue |
193   StringValue
194 ;
195 BooleanValue:
196   value = XBooleanLiteral
197 ;
198 NumberValue:
199   value = XNumberLiteral
200 ;
201 StringValue:
202   value = XStringLiteral
```

```
;
OpEquality:
  '==' | '!=' ;
OpCompare:
  '>=' | '<' '=' | '>' | '<' ;

//////// COMPONENT INSTANCES ///////////////
ComponentInstance:
    ComponentAssociationTypeReference
  | OwnerReference
;
ComponentAssociationTypeReference:
  'owner.' ref=[ComponentAssociationType]
;
OwnerReference:
  {OwnerReference} 'owner'
;

////////// ROLE INSTANCES /////////////////
AbstractRoleInstance:
    RoleInstanceVariable
  | FormalRoleParam
;
RoleInstanceVariable:
  name=ValidID
;
FormalRoleParam :
  type=[RoleType] name=ValidID
;
FormalRoleParamsBlock:
  {FormalRoleParamsBlock}
  '('
    ( params+=FormalRoleParam (',' params += FormalRoleParam)* )?
  ')'
;

//////// ROLE INSTANCE REFERENCES ///////////////
RoleInstanceReference:
    AbstractRoleInstanceReference
  | {SelfReference} 'self'
;
AbstractRoleInstanceReference:
  ref=[AbstractRoleInstance]
;
ActualRoleParamsBlock:
  {ActualRoleParamsBlock}
  '('
    ( params+=RoleInstanceReference (',' params += RoleInstanceReference)* )?
  ')'
;

////////// DATA INSTANCES /////////////////
AbstractDataVariable:
    FormalDataParam
  | DataVariable
;
DataVariable:
  name=ValidID
;
FormalDataParam:
  type=JvmTypeReference name=ValidID
;
FormalDataParamsBlock:
  {FormalDataParamsBlock}
  '('
    ( params+=FormalDataParam (',' params+=FormalDataParam)* )?
  ')'
;
```

```
271
272 /////////// DATA INSTANCE REFERENCES /////////////////
273 DataExpression:
274     SimpleDataExpression
275   | Addition
276   | Subtraction;
277 Addition returns DataExpression:
278   SimpleDataExpression ({Addition.left=current} operator="+" right=SimpleDataExpression
        )
279 ;
280 Subtraction returns DataExpression:
281   SimpleDataExpression ({Subtraction.left=current} operator="-" right=
        SimpleDataExpression)
282 ;
283 SimpleDataExpression returns DataExpression:
284     AbstractDataReference
285   | AbstractDataValue
286 ;
287 AbstractDataReference:
288     AbstractDataVariableReference
289   |   RoleAttributeTypeReference
290   | ComponentAttributeTypeReference
291 ;
292 AbstractDataVariableReference:
293   ref=[AbstractDataVariable]
294 ;
295 RoleAttributeTypeReference:
296   'self.' ref=[RoleAttributeType]
297 ;
298 ComponentAttributeTypeReference:
299   'owner.' ref=[ComponentAttributeType]
300 ;
301 ActualDataParamsBlock:
302   {ActualDataParamsBlock}
303   '('
304     ( params+=DataExpression (',' params += DataExpression)* )?
305   ')'
306 ;
307
308 ////////////// AUXILIARIES ////////////////
309 // Parent for Helena objects that should be duplicate free
310  AbstractDuplicateFreeObject:
311     AbstractHelenaEntity
312   | AbstractFieldType
313   | MessageType
314   | OperationType
315   | RoleBehavior
316   | Process
317   | Label
318   | AbstractRoleInstance
319   | AbstractDataVariable
320 ;
321 // Parent for all structural entities of Helena
322 AbstractHelenaEntity:
323     ComponentType
324   | RoleType
325   | EnsembleStructure
326 ;
327 // Parent for all fields (component or role attributes and component associations)
328 AbstractFieldType:
329     AbstractComponentFieldType
330   | RoleAttributeType
331 ;
332 // Parent for entities used in a role behavior
333 AbstractRoleBehaviorEntity:
334     Process
335   | ProcessExpression
336   | Action
```

```
337     | ComponentInstance
338     | RoleInstanceReference
339     | AbstractRoleInstance
340     | AbstractDataVariable
341     | DataExpression
342   ;
343   // Parent for all instances
344   AbstractInstance:
345     AbstractDataVariable |
346     AbstractRoleInstance
347   ;
```

Appendix C

P2P Example

The HELENA development methodology was showcased at a small peer-to-peer (p2p) example throughout this thesis. The envisioned showcase supports the distributed storage of files in a p2p network. Several peers in this network work together to retrieve a file upon request.

In this appendix, we first show the complete specification of the p2p example in HELENATEXT. Afterwards, the PROMELA specification is given which is automatically generated from the HELENATEXT specification with the HELENA workbench (cf. Chap. 8). The PROMELA specification is enriched by an `init`-process to set up an initial state as given in Sec. 5.3 and with the goals of the p2p example explained in Sec. 5.3 formulated as LTL properties.

The complete p2p example can be found on the attached CD in the project `eu.ascens.helenaText.p2p` which relies on the HELENA workbench.

C.1 Specification in HELENATEXT

This section lists the complete HELENATEXT specification of the p2p example. It was developed relying on the HELENA workbench from Chap. 8. Compared to the initial model of p2p example in Sec. 2.4, it was extended to meet all assumptions described in Sec. 5.1 for model-checking with Spin.

```
1  // p2p example
2  package p2p {
3    componentType Peer {
4      attr boolean hasFile;
5      attr int content;
6      assoc Peer neighbor;
7      op void printFile();
8    }
9
10   roleType Requester over Peer {
11     roleattr boolean hasFile;
12     rolemsg out reqAddr(Requester req)();
13     rolemsg in sndAddr(Provider prov)();
14     rolemsg out reqFile(Requester req)();
15     rolemsg in sndFile()(int content);
16   }
17
18   roleType Router over Peer {
19     rolemsg in/out reqAddr(Requester req)();
20     rolemsg out sndAddr(Provider prov)();
21   }
```

```
22
23    roleType Provider over Peer {
24      rolemsg in reqFile(Requester req)();
25      rolemsg out sndFile()(int content);
26    }
27
28    ensembleStructure TransferEnsemble {
29      <Requester, min=1, max=1, cap=2>;
30      <Router, min=0, max=*, cap=2>;
31      <Provider, min=0, max=1, cap=1>;
32    }
33
34    roleBehavior Requester =
35      router <- create(Router, owner.neighbor) .
36      router ! reqAddr(self)() .
37      ? sndAddr(Provider prov)() .
38      stateSndAddr .
39      prov ! reqFile(self)() .
40      ? sndFile()(int content) .
41      stateSndFile .
42      owner.content = content .
43      owner.hasFile = true .
44      self.hasFile = true .
45      owner.printFile() .
46      quit
47
48    roleBehavior Provider =
49      ? reqFile(Requester req)() .
50      req ! sndFile()(owner.content) .
51      quit
52
53    roleBehavior Router = RouterProc {
54      process RouterProc =
55        ? reqAddr(Requester req)() .
56        if ( owner.hasFile ) {
57          provider <- create(Provider, owner) .
58          req ! sndAddr(provider)() .
59          quit
60        }
61        else {
62          if ( !plays(Router, owner.neighbor) ) {
63            router <- create(Router, owner.neighbor) .
64            router ! reqAddr(req)() .
65            RouterProc
66          }
67          else {
68            quit
69          }
70        }
71    }
72  }
```

C.2 Generated Promela Specification with Goals

This section lists the Promela specification of the p2p example. It was automatically generated from the HelenaText specification of the previous section using the Helena workbench from Chap. 8. The translation is derived following the rules in Sec. 5.2.2. However, the translation of all actions is extracted to `inline`-function such that the generated code is cleaner (e.g., an `inline`-function `send_reqAddr` is generated for sending the message `reqAddr`). Furthermore, the Promela specification is enriched by an `init`-process to set up an initial state as given in Sec. 5.3 and with the goals of the p2p example as explained in Sec. 5.3.

```
1  ////////////// ensemble structure multiplicities ///////////////////////
2  int minRequester = 1;
3  int maxRequester = 1;
4  int currentRequester = 0;
5
6  int minRouter = 0;
7  int maxRouter = (2^31)-1;
8  int currentRouter = 0;
9
10 int minProvider = 0;
11 int maxProvider = 1;
12 int currentProvider = 0;
13
14 ////////////// component operation definitions ///////////////////////
15 typedef PeerOperation {
16   mtype {
17     // operations to access attributes of component
18     GET_HASFILE, SET_HASFILE, GET_CONTENT, SET_CONTENT,
19
20     // operations to access associations to other components
21     GET_NEIGHBOR,
22
23     // operations
24     OP_PRINTFILE,
25
26     // operations to manage role playing
27     CREATE_REQUESTER, GET_REQUESTER, QUIT_REQUESTER, PLAYS_REQUESTER,
28     CREATE_ROUTER, GET_ROUTER, QUIT_ROUTER, PLAYS_ROUTER,
29     CREATE_PROVIDER, GET_PROVIDER, QUIT_PROVIDER, PLAYS_PROVIDER
30   };
31
32   mtype optype;
33   chan parameters;
34   chan answer;
35 }
36
37 //////// helpers for communication between roles and Peer components /////////
38 inline peer_retrieveRole(rtOperation,component,role) {
39   PeerOperation op;
40   op.optype = rtOperation;
41     chan answer = [0] of { chan };
42   op.answer = answer;
43   component!op;
44   answer?role;
45 }
46
47 inline peer_quitRole(rtOperation,component) {
48   PeerOperation op;
49   op.optype = rtOperation;
50   component!op;
51 }
52
53 inline peer_isPlaying(playsOperation,component,plays) {
54   PeerOperation op;
55   op.optype = playsOperation;
56     chan answer = [0] of { bool };
57   op.answer = answer;
58   component!op;
59   answer?plays;
60 }
61
62 inline peer_retrieveAssociation(assocOperation,component,assoc) {
63   PeerOperation op;
64   op.optype = assocOperation;
65     chan answer = [0] of { chan };
66   op.answer = answer;
67   component!op;
68   answer?assoc;
```

```
69  }
70
71  inline peer_getHasFile(component,hasFile) {
72    PeerOperation op;
73    op.optype = GET_HASFILE;
74      chan answer = [0] of { bool };
75    op.answer = answer;
76    component!op;
77    answer?hasFile;
78  }
79
80  inline peer_setHasFile(component,value) {
81    PeerOperation op;
82    op.optype = SET_HASFILE;
83      chan parameters = [0] of { bool };
84    op.parameters = parameters;
85    component!op;
86    parameters!value;
87  }
88
89  inline peer_getContent(component,content) {
90    PeerOperation op;
91    op.optype = GET_CONTENT;
92      chan answer = [0] of { int };
93    op.answer = answer;
94    component!op;
95    answer?content;
96  }
97
98  inline peer_setContent(component,value) {
99    PeerOperation op;
100   op.optype = SET_CONTENT;
101     chan parameters = [0] of { int };
102   op.parameters = parameters;
103   component!op;
104   parameters!value;
105 }
106
107 inline peer_callPrintFile(component) {
108   PeerOperation op;
109   op.optype = OP_PRINTFILE;
110   component!op;
111 }
112
113 ////////////// message definitions ///////////////////////
114 mtype {
115   setOffInitialRole,
116   reqFile,
117   sndFile,
118   reqAddr,
119   sndAddr
120 }
121
122 //////// helper for setting up initial state /////////
123 inline send_setOffInitialRole(receiver) {
124   receiver!setOffInitialRole,1,1;
125 }
126
127 inline receive_setOffInitialRole() {
128   self?setOffInitialRole,1,1;
129 }
130
131 //////// helper for communication between roles /////////
132 inline send_reqFile(receiver,req) {
133   receiver!reqFile,req,1;
134 }
135
136 inline receive_reqFile(req) {
```

```
137   self?reqFile,req,1;
138 }
139
140 inline send_sndFile(receiver,content) {
141   receiver!sndFile,1,content;
142 }
143
144 inline receive_sndFile(content) {
145   self?sndFile,1,content;
146 }
147
148 inline send_reqAddr(receiver,req) {
149   receiver!reqAddr,req,1;
150 }
151
152 inline receive_reqAddr(req) {
153   self?reqAddr,req,1;
154 }
155
156 inline send_sndAddr(receiver,prov) {
157   receiver!sndAddr,prov,1;
158 }
159
160 inline receive_sndAddr(prov) {
161   self?sndAddr,prov,1;
162 }
163
164 ////////////// process definition of component type Peer ////////////
165 proctype Peer(
166     bool hasFile; int content;
167     chan neighbor;
168     chan self) {
169
170   bool playsRequester = false;
171   chan requester = [2] of { mtype,chan,int };
172   bool playsRouter = false;
173   chan router = [2] of { mtype,chan,int };
174   bool playsProvider = false;
175   chan provider = [1] of { mtype,chan,int };
176
177   startPeer: true;
178
179   PeerOperation op;
180
181   do
182   ::atomic {
183     self?op ->
184     if
185     ::op.optype == GET_HASFILE -> op.answer!hasFile
186     ::op.optype == SET_HASFILE -> op.parameters?hasFile
187     ::op.optype == GET_CONTENT -> op.answer!content
188     ::op.optype == SET_CONTENT -> op.parameters?content
189
190     ::op.optype == GET_NEIGHBOR -> op.answer!neighbor
191
192     ::op.optype == OP_PRINTFILE ->
193       // add intended behavior of this operation
194
195     ::op.optype == CREATE_REQUESTER ->
196       if
197       ::!playsRequester && currentRequester < maxRequester ->
198         run Requester(self, requester);
199         playsRequester = true;
200         currentRequester++;
201         op.answer!requester
202       fi
203     ::op.optype == GET_REQUESTER ->
204       if
```

```
205         ::playsRequester ->
206           op.answer!requester
207         fi
208       ::op.optype == QUIT_REQUESTER ->
209         if
210         ::playsRequester && currentRequester > minRequester ->
211           playsRequester = false;
212           currentRequester--
213         fi
214       ::op.optype == PLAYS_REQUESTER ->
215         op.answer!playsRequester
216       ::op.optype == CREATE_ROUTER ->
217         if
218         ::!playsRouter && currentRouter < maxRouter ->
219           run Router(self, router);
220           playsRouter = true;
221           currentRouter++;
222           op.answer!router
223         fi
224       ::op.optype == GET_ROUTER ->
225         if
226         ::playsRouter ->
227           op.answer!router
228         fi
229       ::op.optype == QUIT_ROUTER ->
230         if
231         ::playsRouter && currentRouter > minRouter ->
232           playsRouter = false;
233           currentRouter--
234         fi
235       ::op.optype == PLAYS_ROUTER ->
236         op.answer!playsRouter
237       ::op.optype == CREATE_PROVIDER ->
238         if
239         ::!playsProvider && currentProvider < maxProvider ->
240           run Provider(self, provider);
241           playsProvider = true;
242           currentProvider++;
243           op.answer!provider
244         fi
245       ::op.optype == GET_PROVIDER ->
246         if
247         ::playsProvider ->
248           op.answer!provider
249         fi
250       ::op.optype == QUIT_PROVIDER ->
251         if
252         ::playsProvider && currentProvider > minProvider ->
253           playsProvider = false;
254           currentProvider--
255         fi
256       ::op.optype == PLAYS_PROVIDER ->
257         op.answer!playsProvider
258       fi
259     }
260     od
261   }
262
263   /////////////// process definition for role type Requester ////////////////////
264   proctype Requester(chan owner, self) {
265     // role type attributes
266     bool roleAttr_hasFile;
267
268     // component type attributes
269     bool compAttr_hasFile;
270     int compAttr_content;
271
272     // component type associations
```

```
273     chan compAssoc_neighbor;
274
275     // local variables for all role instance variables (of create/get and incoming
            messages)
276     chan router;
277     chan prov;
278
279     // local variables for all formal data parameters (of incoming messages)
280     int content;
281
282     // local variables for all return values of operations
283
284     // local variables for all plays queries
285
286     // start label
287     startRequester: true;
288
289     //////// role behavior //////////
290
291     // retrieve reference to component instance
292     peer_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
293
294     // create/get role instance
295     peer_retrieveRole(CREATE_ROUTER,compAssoc_neighbor,router);
296
297     // outgoing message
298     send_reqAddr(router,self);
299
300     // incoming message
301     receive_sndAddr(prov);
302
303     // state label
304     stateSndAddr: true;
305
306     // outgoing message
307     send_reqFile(prov,self);
308
309     // incoming message
310     receive_sndFile(content);
311
312     // state label
313     stateSndFile: true;
314
315     // set comp attr
316     peer_setContent(owner,content);
317
318     // set comp attr
319     peer_setHasFile(owner,true);
320
321     // set role attr
322     roleAttr_hasFile = true;
323
324     // call operation at component
325     peer_callPrintFile(owner);
326
327     // quit
328     peer_quitRole(QUIT_REQUESTER,owner);
329     goto endRequester;
330
331     endRequester: false
332   }
333
334   /////////////// process definition for role type Provider ////////////////////
335   proctype Provider(chan owner, self) {
336     // role type attributes
337
338     // component type attributes
339     bool compAttr_hasFile;
```

```
340     int compAttr_content;
341
342     // component type associations
343     chan compAssoc_neighbor;
344
345     // local variables for all role instance variables (of create/get and incoming
            messages)
346     chan req;
347
348     // local variables for all formal data parameters (of incoming messages)
349
350     // local variables for all return values of operations
351
352     // local variables for all plays queries
353
354     // start label
355     startProvider: true;
356
357     //////// role behavior //////////
358
359     // incoming message
360     receive_reqFile(req);
361
362     // outgoing message
363     peer_getContent(owner,compAttr_content);
364     send_sndFile(req,compAttr_content);
365
366     // quit
367     peer_quitRole(QUIT_PROVIDER,owner);
368     goto endProvider;
369
370     endProvider: false
371   }
372
373   /////////////// process definition for role type Router ////////////////////
374   proctype Router(chan owner, self) {
375     // role type attributes
376
377     // component type attributes
378     bool compAttr_hasFile;
379     int compAttr_content;
380
381     // component type associations
382     chan compAssoc_neighbor;
383
384     // local variables for all role instance variables (of create/get and incoming
            messages)
385     chan provider;
386     chan req;
387     chan router;
388
389     // local variables for all formal data parameters (of incoming messages)
390
391     // local variables for all return values of operations
392
393     // local variables for all plays queries
394     bool compAssoc_neighborPlaysRouter;
395
396     // start label
397     startRouter: true;
398
399     //////// role behavior //////////
400
401     // declare local process by label
402     processRouterProc: true;
403
404     // incoming message
405     receive_reqAddr(req);
```

```
406
407   // if-then-else
408   atomic {
409     // retrieve values for guards in guarded choice
410     peer_getHasFile(owner,compAttr_hasFile);;
411
412     if
413     ::(compAttr_hasFile) ->
414
415       // retrieve reference to component instance
416
417       // create/get role instance
418       peer_retrieveRole(CREATE_PROVIDER,owner,provider);
419
420       goto label581711204
421     :: else ->
422
423       // if-then-else
424       // retrieve values for guards in if-then-else
425       peer_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
426       peer_isPlaying(PLAYS_ROUTER,compAssoc_neighbor,compAssoc_neighborPlaysRouter);
427
428       if
429       ::(!compAssoc_neighborPlaysRouter) ->
430
431         // retrieve reference to component instance
432         peer_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
433
434         // create/get role instance
435         peer_retrieveRole(CREATE_ROUTER,compAssoc_neighbor,router);
436
437         goto label1231902650
438
439       :: else ->
440
441         // quit
442         peer_quitRole(QUIT_ROUTER,owner);
443         goto endRouter
444       fi
445     fi
446   };
447
448   label581711204: true;
449
450   // outgoing message
451   send_sndAddr(req,provider);
452
453   // quit
454   peer_quitRole(QUIT_ROUTER,owner);
455   goto endRouter;
456
457   label1231902650: true;
458
459   // outgoing message
460   send_reqAddr(router,req);
461
462   // process invocation by goto label
463   goto processRouterProc;
464
465   endRouter: false
466 }
467
468
469 ///////////////// start ensemble ////////////////////////
470 init {
471   chan p1 = [0] of { PeerOperation };
472   chan p2 = [0] of { PeerOperation };
473   chan p3 = [0] of { PeerOperation };
```

```
474
475   if
476   ::run Peer(false,0,p2,p1);
477   ::run Peer(true,12345,p2,p1);
478   fi;
479
480   if
481   ::run Peer(false,0,p3,p2);
482   ::run Peer(true,12345,p3,p2);
483   fi;
484
485   if
486   ::run Peer(false,0,p1,p3);
487   ::run Peer(true,12345,p1,p3);
488   fi;
489
490   chan req;
491   peer_retrieveRole(CREATE_REQUESTER,p1,req);
492 }
493
494 ////////////////////// Goals ////////////////////////////
495
496 // achieve goal
497 ltl AchievePeer {
498   [] ( Requester@startRequester ->
499     ( (Peer[1]:hasFile || Peer[2]:hasFile || Peer[3]:hasFile)
500     -> <> Requester:roleAttr_hasFile
501     )
502   )
503 }
504
505 // maintain goal
506 ltl MaintainPeer {
507   [] ( Requester@startRequester ->
508     ( (Peer[1]:hasFile || Peer[2]:hasFile || Peer[3]:hasFile)
509     -> [] (Peer[1]:hasFile || Peer[2]:hasFile || Peer[3]:hasFile)
510     )
511   )
512 }
```

Appendix D

SCP Case Study

To apply the HELENA development methodology to a larger software system, we developed the Science Cloud Platform (SCP), one of the case studies from the EU project ASCENS [WHKM15], in Chap. 10. The SCP is a platform of distributed, voluntarily provided computing nodes. The node interact in a peer-to-peer manner to execute, keep alive, and allow usage of user-defined software applications.

In this appendix, we first show the complete specification of the case study in HELENATEXT. Afterwards, the PROMELA specification is given which is automatically generated from the HELENATEXT specification with the HELENA workbench (cf. Chap. 8). The PROMELA specification is enriched by an `init`-process to set up an initial state as given in Sec. 10.5 and with the goals of the SCP case study as explained in Sec. 10.5 formulated as LTL properties.

The complete specification of the SCP case study in HELENATEXT can be found on the attached CD in the project `eu.ascens.helenaText.scp`. A distributed implementation of the SCP based on the HELENATEXT ensemble specification can be retrieved from `http://svn.pst.ifi.lmu.de/trac/scp`, version v3 of the node core implementation with gossip strategy.

D.1 Specification in HELENATEXT

This section lists the complete HELENATEXT specification of the SCP case study. It was developed relying on the HELENA workbench from Chap. 8. Compared to the initial model of SCP case study in Sec. 10.4, it was extended to meet all assumptions described in Sec. 5.1 for model-checking with Spin. The extensions are listed in Sec. 10.5.2.

```
1  package scp {
2
3    componentType Node {
4      attr int id;
5      attr int reqs;
6
7      attr int code;
8      attr boolean isExecuting;
9
10     assoc Node neighbor;
11
12     op void printResult();
13   }
14
15   roleType Deployer over Node {
```

```
16      roleattr int appID;
17      roleattr int appReqs;
18      roleattr int appCode;
19
20      rolemsg out findStorage(Deployer d)(int appID, int foundID);
21      rolemsg in foundStorage(Storage s)();
22
23      rolemsg out store()(int appID, int appReqs, int appCode);
24    }
25
26    roleType PotStorage over Node {
27      rolemsg in/out findStorage(Deployer d)(int appID, int foundID);
28      rolemsg in/out createStorage(Deployer d)(int foundID, int startID);
29      rolemsg out foundStorage(Storage s)();
30    }
31
32    roleType Storage over Node {
33      rolemsg in store()(int appID, int appReqs, int appCode);
34
35      rolemsg out initiate(Storage s)(int appID, int appReqs);
36
37      rolemsg in reqCode(Executor e)();
38      rolemsg out sndCode()(int appCode);
39    }
40
41    roleType Initiator over Node {
42      rolemsg in initiate(Storage s)(int appID, int appReqs);
43
44      rolemsg out findExecutor(Initiator i)(int appReqs);
45      rolemsg in foundExecutor(Executor e)();
46      rolemsg out execute(Initiator i, Storage s)(int appID);
47      rolemsg in executing()();
48
49      rolemsg out findRequester(Executor e)(int startID);
50    }
51
52    roleType PotExecutor over Node {
53      rolemsg in/out findExecutor(Initiator i)(int appReqs);
54      rolemsg out foundExecutor(Executor e)();
55    }
56
57    roleType Executor over Node {
58      rolemsg in execute(Initiator i, Storage s)(int appID);
59      rolemsg out reqCode(Executor e)();
60      rolemsg in sndCode()(int appCode);
61      rolemsg out executing()();
62
63      rolemsg in reqService(Requester r)();
64      rolemsg out sndService()();
65    }
66
67    roleType PotRequester over Node {
68      rolemsg in/out findRequester(Executor e)(int startID);
69      rolemsg out inform(Executor e, PotRequester pr)();
70      rolemsg in ackInformation()();
71    }
72
73    roleType Requester over Node {
74      roleattr boolean hasResult;
75
76      rolemsg in inform(Executor e, PotRequester pr)();
77      rolemsg out ackInformation()();
78
79      rolemsg out reqService(Requester r)();
80      rolemsg in sndService()();
81    }
82
83    ensembleStructure ScienceCloudPlatform {
```

```
84      <Deployer, min = 0, max = 1, cap = 1>;
85      <PotStorage, min = 0, max = *, cap = 2>;
86      <Storage, min = 0, max = 1, cap = 2>;
87      <Initiator, min = 0, max = 1, cap = 2>;
88      <PotExecutor, min = 0, max = *, cap = 2>;
89      <Executor, min = 0, max = 1, cap = 2>;
90      <PotRequester, min = 0, max = *, cap = 2>;
91      <Requester, min = 0, max = *, cap = 2>;
92    }
93
94    ////////////////////////// BEHAVIORS ////////////////////////////
95
96    roleBehavior Deployer =
97      self.appID = 1234 .
98      self.appReqs = 1234 .
99      self.appCode = 1234 .
100     ps <- create(PotStorage,owner.neighbor) .
101     // assumption: node IDs > 0
102     ps ! findStorage(self)(self.appID,owner.id) .
103     ? foundStorage(Storage s)() .
104     s ! store()(self.appID,self.appReqs,self.appCode) .
105     quit
106
107   roleBehavior PotStorage = PotStorageProcess {
108     process PotStorageProcess =
109       ? findStorage(Deployer depl)(int appID,int foundID) .
110
111       if ( (owner.id-appID < 0 && foundID-appID < 0 && appID-owner.id < appID-foundID)
112         ||
113          (owner.id-appID >= 0 && foundID-appID >= 0 && owner.id-appID < foundID-appID)
114         ||
115          (owner.id-appID < 0 && foundID-appID >= 0 && appID-owner.id < foundID-appID)
116         ||
117          (owner.id-appID >= 0 && foundID-appID < 0 && owner.id-appID < appID-foundID)
118       ) {
119         if (!plays(PotStorage,owner.neighbor)) {
120           psSmallest1 <- create(PotStorage,owner.neighbor) .
121           psSmallest1 ! findStorage(depl)(appID,owner.id) .
122           SecondRoundTrip
123         }
124         else {
125           psSmallest2 <- get(PotStorage,owner.neighbor) .
126           psSmallest2 ! createStorage(depl)(owner.id,owner.id) .
127           SecondRoundTrip
128         }
129       }
130       // if owner.id is not smallest ID so far
131       else {
132         if (!plays(PotStorage,owner.neighbor))  {
133           psNotSmallest1 <- create(PotStorage,owner.neighbor) .
134           psNotSmallest1 ! findStorage(depl)(appID,foundID) .
135           SecondRoundTrip
136         }
137         else {
138           psNotSmallest2 <- get(PotStorage,owner.neighbor) .
139           psNotSmallest2 ! createStorage(depl)(foundID,owner.id) .
140           SecondRoundTrip
141         }
142       }
143
144     process SecondRoundTrip =
145       ? createStorage(Deployer depl2)(int foundID2,int startID) .
146
147       if (owner.id == foundID2) {
148         if (!plays(Storage,owner)) {
149           s <- create(Storage,owner) .
150           depl2 ! foundStorage(s)() .
151           if (owner.id != startID)  {
```

```
152             if (plays(PotStorage,owner.neighbor)) {
153               psFwd1 <- get(PotStorage,owner.neighbor) .
154               psFwd1 ! createStorage(depl2)(foundID2,startID) .
155               quit
156             }
157             // should never happen
158             else {
159               quit
160             }
161           }
162           else {
163             quit
164           }
165         }
166         // should not happen
167         else {
168           quit
169         }
170       }
171       else {
172         if (owner.id != startID)  {
173           if (plays(PotStorage,owner.neighbor)) {
174             psFwd2 <- get(PotStorage,owner.neighbor) .
175             psFwd2 ! createStorage(depl2)(foundID2,startID) .
176             quit
177           }
178           // should never happen
179           else {
180             quit
181           }
182         }
183         else {
184           quit
185         }
186       }
187     }
188
189     roleBehavior Storage =
190       ? store()(int appID, int appReqs, int appCode) .
191       owner.code = appCode .
192
193       i <- create(Initiator,owner) .
194       i ! initiate(self)(appID,appReqs) .
195
196       ? reqCode(Executor e)() .
197       e ! sndCode()(owner.code) .
198
199       quit //finished since code is stored on Node (we do not have the concept of
              undeploying)
200
201     roleBehavior Initiator =
202       ? initiate(Storage s)(int appID,int appReqs) .
203
204       pe <- create(PotExecutor,owner) .
205       pe ! findExecutor(self)(appReqs) .
206       ? foundExecutor(Executor e)() .
207       e ! execute(self,s)(appID) .
208       ? executing()() .
209
210       pr <- create(PotRequester,owner.neighbor) .
211       pr ! findRequester(e)(owner.id) .
212
213       quit //finished since all requesters have been informed (we do not have the concept
               of explicitly stopping)
214
215
216     roleBehavior PotExecutor =
217       ? findExecutor(Initiator i)(int appReqs) .
```

```
218
219     // assumption: at least one node in the network that satisfies requirements
220
221     // if the node satisfies the requirements
222     if (owner.reqs == appReqs) {
223       if (!plays(Executor,owner)) {
224         // if the node is ready to execute
225         e <- create(Executor,owner) .
226         i ! foundExecutor(e)() .
227         quit
228       }
229       // should never happen
230       else {
231         quit
232       }
233     }
234     // if the node does not satisfy the requirements
235     else {
236       if (!plays(PotExecutor,owner.neighbor)) {
237         pe <- create(PotExecutor,owner.neighbor) .
238         pe ! findExecutor(i)(appReqs) .
239         quit
240       }
241       // should never happen
242       else {
243         quit
244       }
245     }
246
247   roleBehavior Executor = ExecutorProcess {
248     process ExecutorProcess =
249       ? execute(Initiator i, Storage s)(int appID) .
250
251       s ! reqCode(self)() .
252       ? sndCode()(int appCode) .
253       owner.isExecuting = true .
254       i ! executing()() .
255
256       ExecutorRunning
257
258     process ExecutorRunning =
259       ? reqService(Requester r)() .
260       r ! sndService()() .
261       ExecutorRunning
262   }
263
264   roleBehavior PotRequester = PotRequesterProcess {
265     process PotRequesterProcess =
266       ? findRequester(Executor e)(int startID) .
267
268       if (plays(Requester,owner)) {
269         r <- get(Requester,owner) .
270         r ! inform(e,self)() .
271         ? ackInformation()() .
272         if (owner.id != startID) {
273           if (!plays(PotRequester,owner.neighbor)) {
274             pr1 <- create(PotRequester,owner.neighbor) .
275             pr1 ! findRequester(e)(startID) .
276             quit
277           }
278           // should never happen
279           else {
280             quit
281           }
282         }
283         else {
284           quit
285         }
```

```
286       }
287       else {
288         if (owner.id != startID) {
289           if (!plays(PotRequester,owner.neighbor)) {
290             pr2 <- create(PotRequester,owner.neighbor) .
291             pr2 ! findRequester(e)(startID) .
292             quit
293           }
294           // should never happen
295           else {
296             quit
297           }
298         }
299         else {
300           quit
301         }
302       }
303   }
304
305   roleBehavior Requester =
306     ? inform(Executor e, PotRequester pr)() .
307     pr ! ackInformation()() .
308
309     e ! reqService(self)() .
310     ? sndService()() .
311     self.hasResult = true .
312     owner.printResult() .
313     quit
314 }
```

D.2 Generated PROMELA Specification with Goals

This section lists the PROMELA specification of the SCP case study. It was automatically generated from the HELENATEXT specification of the previous section using the HELENA workbench from Chap. 8. The translation is derived following the rules in Sec. 5.2.2. However, the translation of all actions is extracted to `inline`-function such that the generated code is cleaner (e.g., an `inline`-function `send_findStorage` is generated for sending the message `findStorage`). Furthermore, the PROMELA specification is enriched by an `init`-process to set up an initial state as given in Sec. 10.5 and with the goals of the SCP case study as explained in Sec. 10.5.

```
1  ////////////// ensemble structure multiplicities ///////////////////////
2  int minDeployer = 0;
3  int maxDeployer = 1;
4  int currentDeployer = 0;
5
6  int minPotStorage = 0;
7  int maxPotStorage = (2^31)-1;
8  int currentPotStorage = 0;
9
10 int minStorage = 0;
11 int maxStorage = 1;
12 int currentStorage = 0;
13
14 int minInitiator = 0;
15 int maxInitiator = 1;
16 int currentInitiator = 0;
17
18 int minPotExecutor = 0;
19 int maxPotExecutor = (2^31)-1;
```

```
20  int currentPotExecutor = 0;
21
22  int minExecutor = 0;
23  int maxExecutor = 1;
24  int currentExecutor = 0;
25
26  int minPotRequester = 0;
27  int maxPotRequester = (2^31)-1;
28  int currentPotRequester = 0;
29
30  int minRequester = 0;
31  int maxRequester = (2^31)-1;
32  int currentRequester = 0;
33
34  ////////////// component operation definitions ///////////////////////
35  typedef NodeOperation {
36    mtype {
37      // operations to access attributes of component
38      GET_ID, SET_ID, GET_REQS, SET_REQS, GET_CODE, SET_CODE, GET_ISEXECUTING,
            SET_ISEXECUTING,
39
40      // operations to access associations to other components
41      GET_NEIGHBOR,
42
43      // operations
44      OP_PRINTRESULT,
45
46      // operations to manage role playing
47      CREATE_DEPLOYER, GET_DEPLOYER, QUIT_DEPLOYER, PLAYS_DEPLOYER,
48      CREATE_POTSTORAGE, GET_POTSTORAGE, QUIT_POTSTORAGE, PLAYS_POTSTORAGE,
49      CREATE_STORAGE, GET_STORAGE, QUIT_STORAGE, PLAYS_STORAGE,
50      CREATE_INITIATOR, GET_INITIATOR, QUIT_INITIATOR, PLAYS_INITIATOR,
51      CREATE_POTEXECUTOR, GET_POTEXECUTOR, QUIT_POTEXECUTOR, PLAYS_POTEXECUTOR,
52      CREATE_EXECUTOR, GET_EXECUTOR, QUIT_EXECUTOR, PLAYS_EXECUTOR,
53      CREATE_POTREQUESTER, GET_POTREQUESTER, QUIT_POTREQUESTER, PLAYS_POTREQUESTER,
54      CREATE_REQUESTER, GET_REQUESTER, QUIT_REQUESTER, PLAYS_REQUESTER
55    };
56
57    mtype optype;
58    chan parameters;
59    chan answer;
60  }
61
62  //////// helpers for communication between roles and Node components /////////
63  inline node_retrieveRole(rtOperation,component,role) {
64    NodeOperation op;
65    op.optype = rtOperation;
66      chan answer = [0] of { chan };
67    op.answer = answer;
68    component!op;
69    answer?role;
70  }
71
72  inline node_quitRole(rtOperation,component) {
73    NodeOperation op;
74    op.optype = rtOperation;
75    component!op;
76  }
77
78  inline node_isPlaying(playsOperation,component,plays) {
79    NodeOperation op;
80    op.optype = playsOperation;
81      chan answer = [0] of { bool };
82    op.answer = answer;
83    component!op;
84    answer?plays;
85  }
86
```

```
87  inline node_retrieveAssociation(assocOperation,component,assoc) {
88    NodeOperation op;
89    op.optype = assocOperation;
90      chan answer = [0] of { chan };
91    op.answer = answer;
92    component!op;
93    answer?assoc;
94  }
95
96  inline node_getId(component,id) {
97    NodeOperation op;
98    op.optype = GET_ID;
99      chan answer = [0] of { int };
100   op.answer = answer;
101   component!op;
102   answer?id;
103 }
104
105 inline node_setId(component,value) {
106   NodeOperation op;
107   op.optype = SET_ID;
108     chan parameters = [0] of { int };
109   op.parameters = parameters;
110   component!op;
111   parameters!value;
112 }
113
114 inline node_getReqs(component,reqs) {
115   NodeOperation op;
116   op.optype = GET_REQS;
117     chan answer = [0] of { int };
118   op.answer = answer;
119   component!op;
120   answer?reqs;
121 }
122
123 inline node_setReqs(component,value) {
124   NodeOperation op;
125   op.optype = SET_REQS;
126     chan parameters = [0] of { int };
127   op.parameters = parameters;
128   component!op;
129   parameters!value;
130 }
131
132 inline node_getCode(component,code) {
133   NodeOperation op;
134   op.optype = GET_CODE;
135     chan answer = [0] of { int };
136   op.answer = answer;
137   component!op;
138   answer?code;
139 }
140
141 inline node_setCode(component,value) {
142   NodeOperation op;
143   op.optype = SET_CODE;
144     chan parameters = [0] of { int };
145   op.parameters = parameters;
146   component!op;
147   parameters!value;
148 }
149
150 inline node_getIsExecuting(component,isExecuting) {
151   NodeOperation op;
152   op.optype = GET_ISEXECUTING;
153     chan answer = [0] of { bool };
154   op.answer = answer;
```

```
155   component!op;
156   answer?isExecuting;
157 }
158
159 inline node_setIsExecuting(component,value) {
160   NodeOperation op;
161   op.optype = SET_ISEXECUTING;
162     chan parameters = [0] of { bool };
163   op.parameters = parameters;
164   component!op;
165   parameters!value;
166 }
167
168 inline node_callPrintResult(component) {
169   NodeOperation op;
170   op.optype = OP_PRINTRESULT;
171   component!op;
172 }
173
174 ////////////// message definitions ///////////////////////
175 mtype {
176   setOffInitialRole,
177   reqCode,
178   createStorage,
179   executing,
180   findStorage,
181   foundExecutor,
182   sndCode,
183   ackInformation,
184   findRequester,
185   inform,
186   initiate,
187   foundStorage,
188   findExecutor,
189   execute,
190   reqService,
191   sndService,
192   store
193 }
194
195 //////// helper for setting up initial state /////////
196 inline send_setOffInitialRole(receiver) {
197   receiver!setOffInitialRole,1,1,1,1,1;
198 }
199
200 inline receive_setOffInitialRole() {
201   self?setOffInitialRole,1,1,1,1,1;
202 }
203
204 //////// helper for communication between roles /////////
205 inline send_reqCode(receiver,e) {
206   receiver!reqCode,e,1,1,1,1;
207 }
208
209 inline receive_reqCode(e) {
210   self?reqCode,e,1,1,1,1;
211 }
212
213 inline send_createStorage(receiver,d,foundID,startID) {
214   receiver!createStorage,d,1,foundID,startID,1;
215 }
216
217 inline receive_createStorage(d,foundID,startID) {
218   self?createStorage,d,1,foundID,startID,1;
219 }
220
221 inline send_executing(receiver) {
222   receiver!executing,1,1,1,1,1;
```

```
223  }
224
225  inline receive_executing() {
226    self?executing,1,1,1,1,1;
227  }
228
229  inline send_findStorage(receiver,d,appID,foundID) {
230    receiver!findStorage,d,1,appID,foundID,1;
231  }
232
233  inline receive_findStorage(d,appID,foundID) {
234    self?findStorage,d,1,appID,foundID,1;
235  }
236
237  inline send_foundExecutor(receiver,e) {
238    receiver!foundExecutor,e,1,1,1,1;
239  }
240
241  inline receive_foundExecutor(e) {
242    self?foundExecutor,e,1,1,1,1;
243  }
244
245  inline send_sndCode(receiver,appCode) {
246    receiver!sndCode,1,1,appCode,1,1;
247  }
248
249  inline receive_sndCode(appCode) {
250    self?sndCode,1,1,appCode,1,1;
251  }
252
253  inline send_ackInformation(receiver) {
254    receiver!ackInformation,1,1,1,1,1;
255  }
256
257  inline receive_ackInformation() {
258    self?ackInformation,1,1,1,1,1;
259  }
260
261  inline send_findRequester(receiver,e,startID) {
262    receiver!findRequester,e,1,startID,1,1;
263  }
264
265  inline receive_findRequester(e,startID) {
266    self?findRequester,e,1,startID,1,1;
267  }
268
269  inline send_inform(receiver,e,pr) {
270    receiver!inform,e,pr,1,1,1;
271  }
272
273  inline receive_inform(e,pr) {
274    self?inform,e,pr,1,1,1;
275  }
276
277  inline send_initiate(receiver,s,appID,appReqs) {
278    receiver!initiate,s,1,appID,appReqs,1;
279  }
280
281  inline receive_initiate(s,appID,appReqs) {
282    self?initiate,s,1,appID,appReqs,1;
283  }
284
285  inline send_foundStorage(receiver,s) {
286    receiver!foundStorage,s,1,1,1,1;
287  }
288
289  inline receive_foundStorage(s) {
290    self?foundStorage,s,1,1,1,1;
```

```
291 }
292
293 inline send_findExecutor(receiver,i,appReqs) {
294   receiver!findExecutor,i,1,appReqs,1,1;
295 }
296
297 inline receive_findExecutor(i,appReqs) {
298   self?findExecutor,i,1,appReqs,1,1;
299 }
300
301 inline send_execute(receiver,i,s,appID) {
302   receiver!execute,i,s,appID,1,1;
303 }
304
305 inline receive_execute(i,s,appID) {
306   self?execute,i,s,appID,1,1;
307 }
308
309 inline send_reqService(receiver,r) {
310   receiver!reqService,r,1,1,1,1;
311 }
312
313 inline receive_reqService(r) {
314   self?reqService,r,1,1,1,1;
315 }
316
317 inline send_sndService(receiver) {
318   receiver!sndService,1,1,1,1,1;
319 }
320
321 inline receive_sndService() {
322   self?sndService,1,1,1,1,1;
323 }
324
325 inline send_store(receiver,appID,appReqs,appCode) {
326   receiver!store,1,1,appID,appReqs,appCode;
327 }
328
329 inline receive_store(appID,appReqs,appCode) {
330   self?store,1,1,appID,appReqs,appCode;
331 }
332
333 ////////////// process definition of component type Node ////////////
334 proctype Node(
335     int id; int reqs; int code; bool isExecuting;
336     chan neighbor;
337     chan self) {
338
339   bool playsDeployer = false;
340   chan deployer = [1] of { mtype,chan,chan,int,int,int };
341   bool playsPotStorage = false;
342   chan potstorage = [2] of { mtype,chan,chan,int,int,int };
343   bool playsStorage = false;
344   chan storage = [2] of { mtype,chan,chan,int,int,int };
345   bool playsInitiator = false;
346   chan initiator = [2] of { mtype,chan,chan,int,int,int };
347   bool playsPotExecutor = false;
348   chan potexecutor = [2] of { mtype,chan,chan,int,int,int };
349   bool playsExecutor = false;
350   chan executor = [2] of { mtype,chan,chan,int,int,int };
351   bool playsPotRequester = false;
352   chan potrequester = [2] of { mtype,chan,chan,int,int,int };
353   bool playsRequester = false;
354   chan requester = [2] of { mtype,chan,chan,int,int,int };
355
356   startNode: true;
357
358   NodeOperation op;
```

```
359
360     do
361     ::atomic {
362       self?op ->
363       if
364       ::op.optype == GET_ID -> op.answer!id
365       ::op.optype == SET_ID -> op.parameters?id
366       ::op.optype == GET_REQS -> op.answer!reqs
367       ::op.optype == SET_REQS -> op.parameters?reqs
368       ::op.optype == GET_CODE -> op.answer!code
369       ::op.optype == SET_CODE -> op.parameters?code
370       ::op.optype == GET_ISEXECUTING -> op.answer!isExecuting
371       ::op.optype == SET_ISEXECUTING -> op.parameters?isExecuting
372
373       ::op.optype == GET_NEIGHBOR -> op.answer!neighbor
374
375       ::op.optype == OP_PRINTRESULT ->
376         // add intended behavior of this operation
377
378       ::op.optype == CREATE_DEPLOYER ->
379         if
380         ::!playsDeployer && currentDeployer < maxDeployer ->
381             run Deployer(self, deployer);
382             playsDeployer = true;
383             currentDeployer++;
384             op.answer!deployer
385         fi
386       ::op.optype == GET_DEPLOYER ->
387         if
388         ::playsDeployer ->
389           op.answer!deployer
390         fi
391       ::op.optype == QUIT_DEPLOYER ->
392         if
393         ::playsDeployer && currentDeployer > minDeployer ->
394           playsDeployer = false;
395           currentDeployer--
396         fi
397       ::op.optype == PLAYS_DEPLOYER ->
398         op.answer!playsDeployer
399       ::op.optype == CREATE_POTSTORAGE ->
400         if
401         ::!playsPotStorage && currentPotStorage < maxPotStorage ->
402             run PotStorage(self, potstorage);
403             playsPotStorage = true;
404             currentPotStorage++;
405             op.answer!potstorage
406         fi
407       ::op.optype == GET_POTSTORAGE ->
408         if
409         ::playsPotStorage ->
410           op.answer!potstorage
411         fi
412       ::op.optype == QUIT_POTSTORAGE ->
413         if
414         ::playsPotStorage && currentPotStorage > minPotStorage ->
415           playsPotStorage = false;
416           currentPotStorage--
417         fi
418       ::op.optype == PLAYS_POTSTORAGE ->
419         op.answer!playsPotStorage
420       ::op.optype == CREATE_STORAGE ->
421         if
422         ::!playsStorage && currentStorage < maxStorage ->
423             run Storage(self, storage);
424             playsStorage = true;
425             currentStorage++;
426             op.answer!storage
```

```
427         fi
428       ::op.optype == GET_STORAGE ->
429         if
430         ::playsStorage ->
431           op.answer!storage
432         fi
433       ::op.optype == QUIT_STORAGE ->
434         if
435         ::playsStorage && currentStorage > minStorage ->
436           playsStorage = false;
437           currentStorage--
438         fi
439       ::op.optype == PLAYS_STORAGE ->
440         op.answer!playsStorage
441       ::op.optype == CREATE_INITIATOR ->
442         if
443         ::!playsInitiator && currentInitiator < maxInitiator ->
444             run Initiator(self, initiator);
445             playsInitiator = true;
446             currentInitiator++;
447             op.answer!initiator
448         fi
449       ::op.optype == GET_INITIATOR ->
450         if
451         ::playsInitiator ->
452           op.answer!initiator
453         fi
454       ::op.optype == QUIT_INITIATOR ->
455         if
456         ::playsInitiator && currentInitiator > minInitiator ->
457           playsInitiator = false;
458           currentInitiator--
459         fi
460       ::op.optype == PLAYS_INITIATOR ->
461         op.answer!playsInitiator
462       ::op.optype == CREATE_POTEXECUTOR ->
463         if
464         ::!playsPotExecutor && currentPotExecutor < maxPotExecutor ->
465             run PotExecutor(self, potexecutor);
466             playsPotExecutor = true;
467             currentPotExecutor++;
468             op.answer!potexecutor
469         fi
470       ::op.optype == GET_POTEXECUTOR ->
471         if
472         ::playsPotExecutor ->
473           op.answer!potexecutor
474         fi
475       ::op.optype == QUIT_POTEXECUTOR ->
476         if
477         ::playsPotExecutor && currentPotExecutor > minPotExecutor ->
478           playsPotExecutor = false;
479           currentPotExecutor--
480         fi
481       ::op.optype == PLAYS_POTEXECUTOR ->
482         op.answer!playsPotExecutor
483       ::op.optype == CREATE_EXECUTOR ->
484         if
485         ::!playsExecutor && currentExecutor < maxExecutor ->
486             run Executor(self, executor);
487             playsExecutor = true;
488             currentExecutor++;
489             op.answer!executor
490         fi
491       ::op.optype == GET_EXECUTOR ->
492         if
493         ::playsExecutor ->
494           op.answer!executor
```

```
495         fi
496       ::op.optype == QUIT_EXECUTOR ->
497         if
498         ::playsExecutor && currentExecutor > minExecutor ->
499           playsExecutor = false;
500           currentExecutor--
501         fi
502       ::op.optype == PLAYS_EXECUTOR ->
503         op.answer!playsExecutor
504       ::op.optype == CREATE_POTREQUESTER ->
505         if
506         ::!playsPotRequester && currentPotRequester < maxPotRequester ->
507             run PotRequester(self, potrequester);
508             playsPotRequester = true;
509             currentPotRequester++;
510             op.answer!potrequester
511         fi
512       ::op.optype == GET_POTREQUESTER ->
513         if
514         ::playsPotRequester ->
515           op.answer!potrequester
516         fi
517       ::op.optype == QUIT_POTREQUESTER ->
518         if
519         ::playsPotRequester && currentPotRequester > minPotRequester ->
520           playsPotRequester = false;
521           currentPotRequester--
522         fi
523       ::op.optype == PLAYS_POTREQUESTER ->
524         op.answer!playsPotRequester
525       ::op.optype == CREATE_REQUESTER ->
526         if
527         ::!playsRequester && currentRequester < maxRequester ->
528             run Requester(self, requester);
529             playsRequester = true;
530             currentRequester++;
531             op.answer!requester
532         fi
533       ::op.optype == GET_REQUESTER ->
534         if
535         ::playsRequester ->
536           op.answer!requester
537         fi
538       ::op.optype == QUIT_REQUESTER ->
539         if
540         ::playsRequester && currentRequester > minRequester ->
541           playsRequester = false;
542           currentRequester--
543         fi
544       ::op.optype == PLAYS_REQUESTER ->
545         op.answer!playsRequester
546       fi
547     }
548     od
549   }
550
551   /////////////// process definition for role type Deployer ////////////////////
552   proctype Deployer(chan owner, self) {
553     // role type attributes
554     int roleAttr_appID;
555     int roleAttr_appReqs;
556     int roleAttr_appCode;
557
558     // component type attributes
559     int compAttr_id;
560     int compAttr_reqs;
561     int compAttr_code;
562     bool compAttr_isExecuting;
```

```
563
564     // component type associations
565     chan compAssoc_neighbor;
566
567     // local variables for all role instance variables (of create/get and incoming
            messages)
568     chan s;
569     chan ps;
570
571     // local variables for all formal data parameters (of incoming messages)
572
573     // local variables for all return values of operations
574
575     // local variables for all plays queries
576
577     receive_setOffInitialRole();
578
579     // start label
580     startDeployer: true;
581
582     //////// role behavior //////////
583
584     // set role attr
585     roleAttr_appID = 1234;
586
587     // set role attr
588     roleAttr_appReqs = 1234;
589
590     // set role attr
591     roleAttr_appCode = 1234;
592
593     // retrieve reference to component instance
594     node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
595
596     // create/get role instance
597     node_retrieveRole(CREATE_POTSTORAGE,compAssoc_neighbor,ps);
598
599     // outgoing message
600     node_getId(owner,compAttr_id);
601     send_findStorage(ps,self,roleAttr_appID,compAttr_id);
602
603     // incoming message
604     receive_foundStorage(s);
605
606     // outgoing message
607     send_store(s,roleAttr_appID,roleAttr_appReqs,roleAttr_appCode);
608
609     // quit
610     node_quitRole(QUIT_DEPLOYER,owner);
611     goto endDeployer;
612
613     endDeployer: false
614   }
615
616   /////////////// process definition for role type PotStorage ////////////////////
617   proctype PotStorage(chan owner, self) {
618     // role type attributes
619
620     // component type attributes
621     int compAttr_id;
622     int compAttr_reqs;
623     int compAttr_code;
624     bool compAttr_isExecuting;
625
626     // component type associations
627     chan compAssoc_neighbor;
628
```

```
629     // local variables for all role instance variables (of create/get and incoming
            messages)
630     chan psNotSmallest2;
631     chan depl2;
632     chan depl;
633     chan psFwd2;
634     chan s;
635     chan psFwd1;
636     chan psNotSmallest1;
637     chan psSmallest1;
638     chan psSmallest2;
639
640     // local variables for all formal data parameters (of incoming messages)
641     int foundID;
642     int appID;
643     int startID;
644     int foundID2;
645
646     // local variables for all return values of operations
647
648     // local variables for all plays queries
649     bool compAssoc_neighborPlaysPotStorage;
650     bool ownerPlaysStorage;
651
652     // start label
653     startPotStorage: true;
654
655     //////// role behavior //////////
656
657     // declare local process by label
658     processPotStorageProcess: true;
659
660     // incoming message
661     receive_findStorage(depl,appID,foundID);
662
663     // if-then-else
664     atomic {
665       // retrieve values for guards in guarded choice
666       node_getId(owner,compAttr_id);;
667
668       if
669       ::((compAttr_id-appID < 0 && foundID-appID < 0 && appID-compAttr_id < appID-foundID
              ) || (compAttr_id-appID >= 0 && foundID-appID >= 0 && compAttr_id-appID <
              foundID-appID) || (compAttr_id-appID < 0 && foundID-appID >= 0 && appID-
              compAttr_id < foundID-appID) || (compAttr_id-appID >= 0 && foundID-appID < 0
              && compAttr_id-appID < appID-foundID)) ->
670
671         // if-then-else
672         // retrieve values for guards in if-then-else
673         node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
674         node_isPlaying(PLAYS_POTSTORAGE,compAssoc_neighbor,
                compAssoc_neighborPlaysPotStorage);
675
676         if
677         ::(!compAssoc_neighborPlaysPotStorage) ->
678
679           // retrieve reference to component instance
680           node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
681
682           // create/get role instance
683           node_retrieveRole(CREATE_POTSTORAGE,compAssoc_neighbor,psSmallest1);
684
685           goto label856192689
686
687         :: else ->
688
689           // retrieve reference to component instance
690           node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
```

```
691
692           // create/get role instance
693           node_retrieveRole(GET_POTSTORAGE,compAssoc_neighbor,psSmallest2);
694
695           goto label1413919803
696         fi
697       :: else ->
698
699         // if-then-else
700         // retrieve values for guards in if-then-else
701         node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
702         node_isPlaying(PLAYS_POTSTORAGE,compAssoc_neighbor,
                compAssoc_neighborPlaysPotStorage);
703
704         if
705         ::(!compAssoc_neighborPlaysPotStorage) ->
706
707           // retrieve reference to component instance
708           node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
709
710           // create/get role instance
711           node_retrieveRole(CREATE_POTSTORAGE,compAssoc_neighbor,psNotSmallest1);
712
713           goto label1015451289
714
715         :: else ->
716
717           // retrieve reference to component instance
718           node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
719
720           // create/get role instance
721           node_retrieveRole(GET_POTSTORAGE,compAssoc_neighbor,psNotSmallest2);
722
723           goto label910184078
724         fi
725       fi
726     };
727
728     label856192689: true;
729
730     // outgoing message
731     node_getId(owner,compAttr_id);
732     send_findStorage(psSmallest1,depl,appID,compAttr_id);
733
734     // declare local process by label
735     processSecondRoundTrip: true;
736
737     // incoming message
738     receive_createStorage(depl2,foundID2,startID);
739
740     // if-then-else
741     atomic {
742       // retrieve values for guards in guarded choice
743       node_getId(owner,compAttr_id);;
744
745       if
746       ::(compAttr_id == foundID2) ->
747
748         // if-then-else
749         // retrieve values for guards in if-then-else
750         node_isPlaying(PLAYS_STORAGE,owner,ownerPlaysStorage);
751
752         if
753         ::(!ownerPlaysStorage) ->
754
755           // retrieve reference to component instance
756
757           // create/get role instance
```

```
758           node_retrieveRole(CREATE_STORAGE,owner,s);
759
760           goto label813022984
761
762         :: else ->
763
764           // quit
765           node_quitRole(QUIT_POTSTORAGE,owner);
766           goto endPotStorage
767         fi
768       :: else ->
769
770         // if-then-else
771         // retrieve values for guards in if-then-else
772         node_getId(owner,compAttr_id);;
773
774         if
775         ::(compAttr_id != startID) ->
776
777           // if-then-else
778           // retrieve values for guards in if-then-else
779           node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
780           node_isPlaying(PLAYS_POTSTORAGE,compAssoc_neighbor,
                  compAssoc_neighborPlaysPotStorage);
781
782           if
783           ::(compAssoc_neighborPlaysPotStorage) ->
784
785             // retrieve reference to component instance
786             node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
787
788             // create/get role instance
789             node_retrieveRole(GET_POTSTORAGE,compAssoc_neighbor,psFwd2);
790
791             goto label1610764758
792
793           :: else ->
794
795             // quit
796             node_quitRole(QUIT_POTSTORAGE,owner);
797             goto endPotStorage
798           fi
799
800         :: else ->
801
802           // quit
803           node_quitRole(QUIT_POTSTORAGE,owner);
804           goto endPotStorage
805       fi
806     fi
807   };
808
809   label813022984: true;
810
811   // outgoing message
812   send_foundStorage(depl2,s);
813
814   // if-then-else
815   atomic {
816     // retrieve values for guards in guarded choice
817     node_getId(owner,compAttr_id);;
818
819     if
820     ::(compAttr_id != startID) ->
821
822       // if-then-else
823       // retrieve values for guards in if-then-else
824       node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
```

```
825         node_isPlaying(PLAYS_POTSTORAGE,compAssoc_neighbor,
                compAssoc_neighborPlaysPotStorage);
826
827         if
828         ::(compAssoc_neighborPlaysPotStorage) ->
829
830           // retrieve reference to component instance
831           node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
832
833           // create/get role instance
834           node_retrieveRole(GET_POTSTORAGE,compAssoc_neighbor,psFwd1);
835
836           goto label1244505886
837
838         :: else ->
839
840           // quit
841           node_quitRole(QUIT_POTSTORAGE,owner);
842           goto endPotStorage
843         fi
844       :: else ->
845
846         // quit
847         node_quitRole(QUIT_POTSTORAGE,owner);
848         goto endPotStorage
849       fi
850     };
851
852     label1244505886: true;
853
854     // outgoing message
855     send_createStorage(psFwd1,depl2,foundID2,startID);
856
857     // quit
858     node_quitRole(QUIT_POTSTORAGE,owner);
859     goto endPotStorage;
860
861     label1610764758: true;
862
863     // outgoing message
864     send_createStorage(psFwd2,depl2,foundID2,startID);
865
866     // quit
867     node_quitRole(QUIT_POTSTORAGE,owner);
868     goto endPotStorage;
869     label1413919803: true;
870
871     // outgoing message
872     node_getId(owner,compAttr_id);node_getId(owner,compAttr_id);
873     send_createStorage(psSmallest2,depl,compAttr_id,compAttr_id);
874
875     // process invocation by goto label
876     goto processSecondRoundTrip;
877
878     label1015451289: true;
879
880     // outgoing message
881     send_findStorage(psNotSmallest1,depl,appID,foundID);
882
883     // process invocation by goto label
884     goto processSecondRoundTrip;
885     label910184078: true;
886
887     // outgoing message
888     node_getId(owner,compAttr_id);
889     send_createStorage(psNotSmallest2,depl,foundID,compAttr_id);
890
891     // process invocation by goto label
```

```
892   goto processSecondRoundTrip;
893
894   endPotStorage: false
895 }
896
897 /////////////// process definition for role type Storage ////////////////////
898 proctype Storage(chan owner, self) {
899   // role type attributes
900
901   // component type attributes
902   int compAttr_id;
903   int compAttr_reqs;
904   int compAttr_code;
905   bool compAttr_isExecuting;
906
907   // component type associations
908   chan compAssoc_neighbor;
909
910   // local variables for all role instance variables (of create/get and incoming
          messages)
911   chan i;
912   chan e;
913
914   // local variables for all formal data parameters (of incoming messages)
915   int appID;
916   int appReqs;
917   int appCode;
918
919   // local variables for all return values of operations
920
921   // local variables for all plays queries
922
923   // start label
924   startStorage: true;
925
926   //////// role behavior //////////
927
928   // incoming message
929   receive_store(appID,appReqs,appCode);
930
931   // set comp attr
932   node_setCode(owner,appCode);
933
934   // retrieve reference to component instance
935
936   // create/get role instance
937   node_retrieveRole(CREATE_INITIATOR,owner,i);
938
939   // outgoing message
940   send_initiate(i,self,appID,appReqs);
941
942   // incoming message
943   receive_reqCode(e);
944
945   // outgoing message
946   node_getCode(owner,compAttr_code);
947   send_sndCode(e,compAttr_code);
948
949   // quit
950   node_quitRole(QUIT_STORAGE,owner);
951   goto endStorage;
952
953   endStorage: false
954 }
955
956 /////////////// process definition for role type Initiator ////////////////////
957 proctype Initiator(chan owner, self) {
958   // role type attributes
```

```
959
960     // component type attributes
961     int compAttr_id;
962     int compAttr_reqs;
963     int compAttr_code;
964     bool compAttr_isExecuting;
965
966     // component type associations
967     chan compAssoc_neighbor;
968
969     // local variables for all role instance variables (of create/get and incoming
            messages)
970     chan s;
971     chan e;
972     chan pr;
973     chan pe;
974
975     // local variables for all formal data parameters (of incoming messages)
976     int appID;
977     int appReqs;
978
979     // local variables for all return values of operations
980
981     // local variables for all plays queries
982
983     // start label
984     startInitiator: true;
985
986     //////// role behavior //////////
987
988     // incoming message
989     receive_initiate(s,appID,appReqs);
990
991     // retrieve reference to component instance
992
993     // create/get role instance
994     node_retrieveRole(CREATE_POTEXECUTOR,owner,pe);
995
996     // outgoing message
997     send_findExecutor(pe,self,appReqs);
998
999     // incoming message
1000    receive_foundExecutor(e);
1001
1002    // outgoing message
1003    send_execute(e,self,s,appID);
1004
1005    // incoming message
1006    receive_executing();
1007
1008    // retrieve reference to component instance
1009    node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
1010
1011    // create/get role instance
1012    node_retrieveRole(CREATE_POTREQUESTER,compAssoc_neighbor,pr);
1013
1014    // outgoing message
1015    node_getId(owner,compAttr_id);
1016    send_findRequester(pr,e,compAttr_id);
1017
1018    // quit
1019    node_quitRole(QUIT_INITIATOR,owner);
1020    goto endInitiator;
1021
1022    endInitiator: false
1023  }
1024
1025  /////////////// process definition for role type PotExecutor ////////////////////
```

```
1026  proctype PotExecutor(chan owner, self) {
1027    // role type attributes
1028
1029    // component type attributes
1030    int compAttr_id;
1031    int compAttr_reqs;
1032    int compAttr_code;
1033    bool compAttr_isExecuting;
1034
1035    // component type associations
1036    chan compAssoc_neighbor;
1037
1038    // local variables for all role instance variables (of create/get and incoming
          messages)
1039    chan pe;
1040    chan i;
1041    chan e;
1042
1043    // local variables for all formal data parameters (of incoming messages)
1044    int appReqs;
1045
1046    // local variables for all return values of operations
1047
1048    // local variables for all plays queries
1049    bool ownerPlaysExecutor;
1050    bool compAssoc_neighborPlaysPotExecutor;
1051
1052    // start label
1053    startPotExecutor: true;
1054
1055    //////// role behavior //////////
1056
1057    // incoming message
1058    receive_findExecutor(i,appReqs);
1059
1060    // if-then-else
1061    atomic {
1062      // retrieve values for guards in guarded choice
1063      node_getReqs(owner,compAttr_reqs);;
1064
1065      if
1066      ::(compAttr_reqs == appReqs) ->
1067
1068        // if-then-else
1069        // retrieve values for guards in if-then-else
1070        node_isPlaying(PLAYS_EXECUTOR,owner,ownerPlaysExecutor);
1071
1072        if
1073        ::(!ownerPlaysExecutor) ->
1074
1075          // retrieve reference to component instance
1076
1077          // create/get role instance
1078          node_retrieveRole(CREATE_EXECUTOR,owner,e);
1079
1080          goto label1784481001
1081
1082        :: else ->
1083
1084          // quit
1085          node_quitRole(QUIT_POTEXECUTOR,owner);
1086          goto endPotExecutor
1087        fi
1088      :: else ->
1089
1090        // if-then-else
1091        // retrieve values for guards in if-then-else
1092        node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
```

```
1093        node_isPlaying(PLAYS_POTEXECUTOR,compAssoc_neighbor,
                compAssoc_neighborPlaysPotExecutor);
1094
1095        if
1096        ::(!compAssoc_neighborPlaysPotExecutor) ->
1097
1098          // retrieve reference to component instance
1099          node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
1100
1101          // create/get role instance
1102          node_retrieveRole(CREATE_POTEXECUTOR,compAssoc_neighbor,pe);
1103
1104          goto label442612558
1105
1106        :: else ->
1107
1108          // quit
1109          node_quitRole(QUIT_POTEXECUTOR,owner);
1110          goto endPotExecutor
1111        fi
1112      fi
1113    };
1114
1115    label1784481001: true;
1116
1117    // outgoing message
1118    send_foundExecutor(i,e);
1119
1120    // quit
1121    node_quitRole(QUIT_POTEXECUTOR,owner);
1122    goto endPotExecutor;
1123
1124    label442612558: true;
1125
1126    // outgoing message
1127    send_findExecutor(pe,i,appReqs);
1128
1129    // quit
1130    node_quitRole(QUIT_POTEXECUTOR,owner);
1131    goto endPotExecutor;
1132
1133    endPotExecutor: false
1134  }
1135
1136  /////////////// process definition for role type Executor ////////////////////
1137  proctype Executor(chan owner, self) {
1138    // role type attributes
1139
1140    // component type attributes
1141    int compAttr_id;
1142    int compAttr_reqs;
1143    int compAttr_code;
1144    bool compAttr_isExecuting;
1145
1146    // component type associations
1147    chan compAssoc_neighbor;
1148
1149    // local variables for all role instance variables (of create/get and incoming
          messages)
1150    chan i;
1151    chan s;
1152    chan r;
1153
1154    // local variables for all formal data parameters (of incoming messages)
1155    int appCode;
1156    int appID;
1157
1158    // local variables for all return values of operations
```

```
1159
1160    // local variables for all plays queries
1161
1162    // start label
1163    startExecutor: true;
1164
1165    //////// role behavior //////////
1166
1167    // declare local process by label
1168    processExecutorProcess: true;
1169
1170    // incoming message
1171    receive_execute(i,s,appID);
1172
1173    // outgoing message
1174    send_reqCode(s,self);
1175
1176    // incoming message
1177    receive_sndCode(appCode);
1178
1179    // set comp attr
1180    node_setIsExecuting(owner,true);
1181
1182    // outgoing message
1183    send_executing(i);
1184
1185    // declare local process by label
1186    processExecutorRunning: true;
1187
1188    // incoming message
1189    receive_reqService(r);
1190
1191    // outgoing message
1192    send_sndService(r);
1193
1194    // process invocation by goto label
1195    goto processExecutorRunning;
1196
1197    endExecutor: false
1198  }
1199
1200  /////////////// process definition for role type PotRequester ////////////////////
1201  proctype PotRequester(chan owner, self) {
1202    // role type attributes
1203
1204    // component type attributes
1205    int compAttr_id;
1206    int compAttr_reqs;
1207    int compAttr_code;
1208    bool compAttr_isExecuting;
1209
1210    // component type associations
1211    chan compAssoc_neighbor;
1212
1213    // local variables for all role instance variables (of create/get and incoming
          messages)
1214    chan r;
1215    chan e;
1216    chan pr1;
1217    chan pr2;
1218
1219    // local variables for all formal data parameters (of incoming messages)
1220    int startID;
1221
1222    // local variables for all return values of operations
1223
1224    // local variables for all plays queries
1225    bool compAssoc_neighborPlaysPotRequester;
```

```
1226    bool ownerPlaysRequester;
1227
1228    // start label
1229    startPotRequester: true;
1230
1231    //////// role behavior //////////
1232
1233    // declare local process by label
1234    processPotRequesterProcess: true;
1235
1236    // incoming message
1237    receive_findRequester(e,startID);
1238
1239    // if-then-else
1240    atomic {
1241      // retrieve values for guards in guarded choice
1242      node_isPlaying(PLAYS_REQUESTER,owner,ownerPlaysRequester);
1243
1244      if
1245      ::(ownerPlaysRequester) ->
1246
1247        // retrieve reference to component instance
1248
1249        // create/get role instance
1250        node_retrieveRole(GET_REQUESTER,owner,r);
1251
1252        goto label778806867
1253      :: else ->
1254
1255        // if-then-else
1256        // retrieve values for guards in if-then-else
1257        node_getId(owner,compAttr_id);;
1258
1259        if
1260        ::(compAttr_id != startID) ->
1261
1262          // if-then-else
1263          // retrieve values for guards in if-then-else
1264          node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
1265          node_isPlaying(PLAYS_POTREQUESTER,compAssoc_neighbor,
                 compAssoc_neighborPlaysPotRequester);
1266
1267          if
1268          ::(!compAssoc_neighborPlaysPotRequester) ->
1269
1270            // retrieve reference to component instance
1271            node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
1272
1273            // create/get role instance
1274            node_retrieveRole(CREATE_POTREQUESTER,compAssoc_neighbor,pr2);
1275
1276            goto label186050747
1277
1278          :: else ->
1279
1280            // quit
1281            node_quitRole(QUIT_POTREQUESTER,owner);
1282            goto endPotRequester
1283          fi
1284
1285        :: else ->
1286
1287          // quit
1288          node_quitRole(QUIT_POTREQUESTER,owner);
1289          goto endPotRequester
1290        fi
1291      fi
1292    };
```

```
1293
1294    label778806867: true;
1295
1296    // outgoing message
1297    send_inform(r,e,self);
1298
1299    // incoming message
1300    receive_ackInformation();
1301
1302    // if-then-else
1303    atomic {
1304      // retrieve values for guards in guarded choice
1305      node_getId(owner,compAttr_id);;
1306
1307      if
1308      ::(compAttr_id != startID) ->
1309
1310        // if-then-else
1311        // retrieve values for guards in if-then-else
1312        node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
1313        node_isPlaying(PLAYS_POTREQUESTER,compAssoc_neighbor,
                compAssoc_neighborPlaysPotRequester);
1314
1315        if
1316        ::(!compAssoc_neighborPlaysPotRequester) ->
1317
1318          // retrieve reference to component instance
1319          node_retrieveAssociation(GET_NEIGHBOR,owner,compAssoc_neighbor);
1320
1321          // create/get role instance
1322          node_retrieveRole(CREATE_POTREQUESTER,compAssoc_neighbor,pr1);
1323
1324          goto label834441627
1325
1326        :: else ->
1327
1328          // quit
1329          node_quitRole(QUIT_POTREQUESTER,owner);
1330          goto endPotRequester
1331        fi
1332      :: else ->
1333
1334        // quit
1335        node_quitRole(QUIT_POTREQUESTER,owner);
1336        goto endPotRequester
1337      fi
1338    };
1339
1340    label834441627: true;
1341
1342    // outgoing message
1343    send_findRequester(pr1,e,startID);
1344
1345    // quit
1346    node_quitRole(QUIT_POTREQUESTER,owner);
1347    goto endPotRequester;
1348
1349    label186050747: true;
1350
1351    // outgoing message
1352    send_findRequester(pr2,e,startID);
1353
1354    // quit
1355    node_quitRole(QUIT_POTREQUESTER,owner);
1356    goto endPotRequester;
1357
1358    endPotRequester: false
1359  }
```

```
1360
1361  /////////////// process definition for role type Requester ////////////////////
1362  proctype Requester(chan owner, self) {
1363    // role type attributes
1364    bool roleAttr_hasResult;
1365
1366    // component type attributes
1367    int compAttr_id;
1368    int compAttr_reqs;
1369    int compAttr_code;
1370    bool compAttr_isExecuting;
1371
1372    // component type associations
1373    chan compAssoc_neighbor;
1374
1375    // local variables for all role instance variables (of create/get and incoming
          messages)
1376    chan e;
1377    chan pr;
1378
1379    // local variables for all formal data parameters (of incoming messages)
1380
1381    // local variables for all return values of operations
1382
1383    // local variables for all plays queries
1384
1385    receive_setOffInitialRole();
1386
1387    // start label
1388    startRequester: true;
1389
1390    //////// role behavior //////////
1391
1392    // incoming message
1393    receive_inform(e,pr);
1394
1395    // outgoing message
1396    send_ackInformation(pr);
1397
1398    // outgoing message
1399    send_reqService(e,self);
1400
1401    // incoming message
1402    receive_sndService();
1403
1404    // set role attr
1405    roleAttr_hasResult = true;
1406
1407    // call operation at component
1408    node_callPrintResult(owner);
1409
1410    // quit
1411    node_quitRole(QUIT_REQUESTER,owner);
1412    goto endRequester;
1413
1414    endRequester: false
1415  }
1416
1417  /////////////// start ensemble ////////////////////////
1418  init {
1419    chan n1 = [0] of { NodeOperation };
1420    chan n2 = [0] of { NodeOperation };
1421
1422    int id1 = 1;
1423    int id2 = 2;
1424    int reqs1 = 1;
1425    int reqs2 = 2;
1426
```

```
1427    // the ID of one node is closest to the ID 1234 of the app
1428    if
1429    ::id1 = 100;
1430    ::id2 = 100;
1431    fi;
1432
1433    // one of the nodes satisfies the requirements 1234 of the app
1434    if
1435    ::reqs1 = 1234;
1436    ::reqs2 = 1234;
1437    fi;
1438
1439    run Node(id1,reqs1,0,0,n2,n1);
1440    run Node(id2,reqs2,0,0,n1,n2);
1441
1442    // nondeterministically choose owner for initial roles
1443    chan ownerDeploy;
1444    if
1445    ::ownerDeploy = n1;
1446    ::ownerDeploy = n2;
1447    fi;
1448
1449    chan ownerReq;
1450    chan ownerReq2;
1451    if
1452    ::ownerReq = n1;ownerReq2 = n2;
1453    ::ownerReq = n2;ownerReq2 = n1;
1454    fi;
1455
1456    chan deploy;
1457    node_retrieveRole(CREATE_DEPLOYER,ownerDeploy,deploy);
1458
1459    chan req;
1460    node_retrieveRole(CREATE_REQUESTER,ownerReq,req);
1461
1462    chan req2;
1463    node_retrieveRole(CREATE_REQUESTER,ownerReq2,req2);
1464
1465    atomic {
1466      send_setOffInitialRole(deploy);
1467      send_setOffInitialRole(req);
1468      send_setOffInitialRole(req2);
1469    }
1470  }
1471
1472  /////////////// Storage Goals ////////////////////
1473
1474  // code gets stored
1475  ltl Achieve_CodeStored {
1476    [] ( Deployer@startDeployer ->
1477      <> (
1478           (Node[1]:code > 0 && Node[1]:code == Deployer:roleAttr_appCode)
1479        || (Node[2]:code > 0 && Node[2]:code == Deployer:roleAttr_appCode)
1480      )
1481    )
1482  }
1483
1484  //ltl Achieve_CodeStored {
1485  //  [] ( Deployer@startDeployer ->
1486  //    <> ( (Node[1]:code > 0 && Node[1]:code == Deployer:roleAttr_appCode &&
1487  //      ( (Node[1]:id-Deployer:roleAttr_appID < 0
1488  //         && Node[2]:id-Deployer:roleAttr_appID < 0
1489  //         && Deployer:roleAttr_appID-Node[1]:id < Deployer:roleAttr_appID-Node[2]:id)
1490  //      || (Node[1]:id-Deployer:roleAttr_appID >= 0
1491  //         && Node[2]:id-Deployer:roleAttr_appID >= 0
1492  //         && Node[1]:id-Deployer:roleAttr_appID < Node[2]:id-Deployer:roleAttr_appID)
1493  //      || (Node[1]:id-Deployer:roleAttr_appID < 0
1494  //         && Node[2]:id-Deployer:roleAttr_appID >= 0
```

```
1495  //         && Deployer:roleAttr_appID-Node[1]:id < Node[2]:id-Deployer:roleAttr_appID)
1496  //      || (Node[1]:id-Deployer:roleAttr_appID >= 0
1497  //         && Node[2]:id-Deployer:roleAttr_appID < 0
1498  //         && Node[1]:id-Deployer:roleAttr_appID < Deployer:roleAttr_appID-Node[2]:id))
1499  //      )
1500  //      || (Node[2]:code > 0 && Node[2]:code == Deployer:roleAttr_appCode &&
1501  //      ( (Node[2]:id-Deployer:roleAttr_appID < 0
1502  //         && Node[1]:id-Deployer:roleAttr_appID < 0
1503  //         && Deployer:roleAttr_appID-Node[2]:id < Deployer:roleAttr_appID-Node[1]:id)
1504  //      || (Node[2]:id-Deployer:roleAttr_appID >= 0
1505  //         && Node[1]:id-Deployer:roleAttr_appID >= 0
1506  //         && Node[2]:id-Deployer:roleAttr_appID < Node[1]:id-Deployer:roleAttr_appID)
1507  //      || (Node[2]:id-Deployer:roleAttr_appID < 0
1508  //         && Node[1]:id-Deployer:roleAttr_appID >= 0
1509  //         && Deployer:roleAttr_appID-Node[2]:id < Node[1]:id-Deployer:roleAttr_appID)
1510  //      || (Node[2]:id-Deployer:roleAttr_appID >= 0
1511  //         && Node[1]:id-Deployer:roleAttr_appID < 0
1512  //         && Node[2]:id-Deployer:roleAttr_appID < Deployer:roleAttr_appID-Node[1]:id))
1513  //      )
1514  //    )
1515  //  ) }
1516
1517  // code stays stored
1518  ltl Maintain_Storage {
1519    [] ( Deployer@startDeployer ->
1520      [] (
1521        (  (Node[1]:code > 0 && Node[1]:code == Deployer:roleAttr_appCode)
1522        || (Node[2]:code > 0 && Node[2]:code == Deployer:roleAttr_appCode)
1523        )
1524        -> [] (  (Node[1]:code > 0 && Node[1]:code == Deployer:roleAttr_appCode)
1525              || (Node[2]:code > 0 && Node[2]:code == Deployer:roleAttr_appCode)
1526              )
1527      )
1528    )
1529  }
1530
1531  /////////////// Execution Goals ////////////////////
1532
1533  // app gets executed
1534  ltl Achieve_AppExecuted {
1535    [] ( Deployer@startDeployer ->
1536      (Node[1]:reqs == Deployer:roleAttr_appReqs
1537      || Node[2]:reqs == Deployer:roleAttr_appReqs)
1538      ->
1539      <> (  (Node[1]:reqs == Deployer:roleAttr_appReqs && Node[1]:isExecuting)
1540          || (Node[2]:reqs == Deployer:roleAttr_appReqs && Node[2]:isExecuting)
1541      )
1542    )
1543  }
1544
1545  // app stays executed
1546  ltl Maintain_Execution {
1547    [] ( Deployer@startDeployer ->
1548        [] (
1549          (Node[1]:isExecuting || Node[2]:isExecuting)
1550          -> [] (Node[1]:isExecuting || Node[2]:isExecuting)
1551        )
1552    )
1553  }
1554
1555  /////////////// Usage Goals ////////////////////
1556
1557  // requester 4 gets served
1558  ltl Achieve_Usage4 {
1559    [] ( Deployer@startDeployer ->
1560      <> Requester[4]:roleAttr_hasResult
1561    )
1562  }
```

```
1563
1564  // requester 5 gets served
1565  ltl Achieve_Usage5 {
1566    [] ( Deployer@startDeployer ->
1567      <> Requester[5]:roleAttr_hasResult
1568    )
1569  }
```

Appendix E

Search-and-Rescue Scenario

The HELENA development methodology for self-adaptive systems was showcased at a small robotic search-and-rescue scenario in Chap. 11. Robots are distributed over an unknown area where recently some kind of disaster happened. The robots have to find victims and transport them to a rescue area. Thereby, they should self-adaptively manage their behaviors. In Chap. 11, the adaptation specification for the example was given by the signature of a robot in Fig. 11.2 and its adaptation automaton in Fig. 11.3.

In this appendix, we introduce the resulting HELENA model of the example in HELENATEXT after the second model transformation (so far this model transformation has to be done by hand and is not yet included into the HELENA workbench). The HELENA specification is comprised of the role-based architecture with one component type `Robot`, five mode role types `RandomWalk`, `DirectedWalk`, `Rescue`, `LowBattery` and `Recharge`, four sensor role types `BatterySensor`, `AtVictimSensor`, `VictimPosSensor` and `RechargeRequestedSensor`, and a role type for the adaptation manager (cf. Sec. 11.6). The specification furthermore lists role behaviors for all role types. The role behaviors for the mode roles are specified by hand and transformed to be interruptible as explained in Sec. 11.7.3.1. The role behaviors for the sensors roles are systematically derived as explained in Sec. 11.7.3.2. Finally, the role behavior for the adaptation automaton is derived from the adaptation automaton in Fig. 11.3 as explained in Sec. 11.7.3.3. The complete specification in HELENATEXT and implementation in jHELENA can be found on the attached CD in the file `search-and-rescue.helena`.

```
1  package searchAndRescue {
2
3    componentType Robot {
4      attr int ownPos;
5
6      attr int battery;
7      attr int atVictim;
8      attr int victimPos;
9      attr int rechargeRequested;
10
11     op void updateBattery();
12     op void updateAtVictim();
13     op void updateVictimPos();
14     op void updateRechargeRequested();
15
16     op void randomStep();
17     op void directStep();
18     op void getCharged();
19     op void charge();
20   }
```

```
21
22    roleType RandomWalk over Robot {
23      rolemsg in interrupt()();
24      rolemsg in resume()();
25    }
26
27    roleType DirectedWalk over Robot {
28      rolemsg in interrupt()();
29      rolemsg in resume()();
30    }
31
32    roleType Rescue over Robot {
33      rolemsg in interrupt()();
34      rolemsg in resume()();
35    }
36
37    roleType LowBattery over Robot {
38      rolemsg in interrupt()();
39      rolemsg in resume()();
40    }
41
42    roleType Recharge over Robot {
43      rolemsg in interrupt()();
44      rolemsg in resume()();
45    }
46
47    roleType BatterySensor over Robot {
48      rolemsg out inform()(String attribute, Integer value);
49    }
50
51    roleType AtVictimSensor over Robot {
52      rolemsg out inform()(String attribute, Integer value);
53    }
54
55    roleType VictimPosSensor over Robot {
56      rolemsg out inform()(String attribute, Integer value);
57    }
58
59    roleType RechargeRequestedSensor over Robot {
60      rolemsg out inform()(String attribute, Integer value);
61    }
62
63    roleType AdaptationManager over Robot {
64      rolemsg out interrupt()();
65      rolemsg out resume()();
66      rolemsg in inform()(String attribute, Integer value);
67    }
68
69    ensembleStructure RobotEnsemble {
70      <RandomWalk, min = 1, max = 1, cap = 0>;
71      <DirectedWalk, min = 1, max = 1, cap = 0>;
72      <Rescue, min = 1, max = 1, cap = 0>;
73      <LowBattery, min = 1, max = 1, cap = 0>;
74      <Recharge, min = 1, max = 1, cap = 0>;
75      <BatterySensor, min = 1, max = 1, cap = 0>;
76      <AtVictimSensor, min = 1, max = 1, cap =  0>;
77      <VictimPosSensor, min = 1, max = 1, cap = 0>;
78      <RechargeRequestedSensor, min = 1, max = 1, cap = 0>;
79      <AdaptationManager, min = 1, max = 1, cap =0>;
80    }
81
82    roleBehavior RandomWalk = RandomWalkProc {
83      process RandomWalkProc =
84        ? resume()() .
85        RandomWalk1
86
87      process RandomWalk1 =
88        RandomWalk2
```

```
89
90      process RandomWalk2 =
91        (
92          ? interrupt()() .
93          ? resume()() .
94          RandomWalk2
95        +
96          owner.randomStep() .
97          RandomWalk1
98        )
99    }
100
101   roleBehavior DirectedWalk = DirectedWalkProc {
102     process DirectedWalkProc =
103       ? resume()() .
104       DirectedWalk1
105
106     process DirectedWalk1 =
107       DirectedWalk2
108
109     process DirectedWalk2 =
110       (
111         ? interrupt()() .
112         ? resume()() .
113         DirectedWalk2
114       +
115         owner.directStep() .
116         DirectedWalk1
117       )
118   }
119
120   roleBehavior Rescue = RescueProc {
121     process RescueProc =
122       ? resume()() .
123       RescueProc1
124
125     process RescueProc1 =
126       quit
127   }
128
129   roleBehavior LowBattery = LowBatteryProc {
130     process LowBatteryProc =
131       ? resume()() .
132       LowBatteryProc1
133
134     process LowBatteryProc1 =
135       (
136         ? interrupt()() .
137         ? resume()() .
138         LowBatteryProc1
139       +
140         owner.getCharged() .
141         LowBatteryProc1
142       )
143   }
144
145   roleBehavior Recharge = RechargeProc {
146     process RechargeProc =
147       ? resume()() .
148       RechargeProc1
149
150     process RechargeProc1 =
151       (
152         ? interrupt()() .
153         ? resume()() .
154         RechargeProc1
155       +
156         owner.charge() .
```

```
157         RechargeProc1
158       )
159   }
160
161   roleBehavior BatterySensor = BatterySensorProc {
162     process BatterySensorProc =
163       am <- get(AdaptationManager, owner) .
164       BatteryMonitor
165
166     process BatteryMonitor =
167       owner.updateBattery() .
168       am ! inform()("battery", owner.battery) .
169       BatteryMonitor
170   }
171
172   roleBehavior AtVictimSensor = AtVictimSensorProc {
173     process AtVictimSensorProc =
174       am <- get(AdaptationManager, owner) .
175       AtVictimMonitor
176
177     process AtVictimMonitor =
178       owner.updateAtVictim() .
179       am ! inform()("atVictim", owner. atVictim) .
180       AtVictimMonitor
181   }
182
183   roleBehavior VictimPosSensor = VictimPosSensorProc {
184     process VictimPosSensorProc =
185       am <- get(AdaptationManager, owner) .
186       VictimPosMonitor
187
188     process VictimPosMonitor =
189       owner.updateVictimPos() .
190       am ! inform()("victimPos", owner.victimPos) .
191       VictimPosMonitor
192   }
193
194   roleBehavior RechargeRequestedSensor = RechargeRequestedSensorProc {
195     process RechargeRequestedSensorProc =
196       am <- get(AdaptationManager, owner) .
197       RechargeRequestedMonitor
198
199     process RechargeRequestedMonitor =
200       owner.updateRechargeRequested() .
201       am ! inform()("rechargeRequested", owner.rechargeRequested) .
202       RechargeRequestedMonitor
203   }
204
205   roleBehavior AdaptationManager = AM {
206     process AM =
207       randomWalkInst <- create(RandomWalk, owner) .
208       directedWalkInst <- create(DirectedWalk, owner) .
209       rescueInst <- create(Rescue, owner) .
210       lowBatteryInst <- create(LowBattery, owner) .
211       rechargeInst <- create(Recharge, owner) .
212       randomWalkInst ! resume()() .
213       RandomWalkProcess
214
215     process RandomWalkProcess =
216       ? inform()(String attribute, Integer value) .
217       if (attribute == "victimPos") {
218         owner.victimPos = value .
219         RandomWalkChange
220       }
221       else {
222         if (attribute == "atVictim")  {
223           owner.atVictim = value .
224           RandomWalkChange
```

```
225           }
226           else {
227             if (attribute == "rechargeRequested") {
228               owner.rechargeRequested = value .
229               RandomWalkChange
230             }
231             else {
232               if (attribute == "battery") {
233                 owner.battery = value .
234                 RandomWalkChange
235               }
236               else {
237                 RandomWalkChange
238               }
239             }
240           }
241         }
242       process RandomWalkChange =
243         if (owner.victimPos != 0) {
244           randomWalkInst ! interrupt()() .
245           directedWalkInst ! resume()() .
246           DirectedWalkProcess
247         }
248         else {
249           if (owner.atVictim == 1) {
250             randomWalkInst ! interrupt()() .
251             rescueInst ! resume()() .
252             RescueProcess
253           }
254           else {
255             if (owner.rechargeRequested == 1) {
256               randomWalkInst ! interrupt()() .
257               rechargeInst ! resume()() .
258               RechargeFromRandomWalkProcess
259             }
260             else {
261               if (owner.battery < 1) {
262                 randomWalkInst ! interrupt()() .
263                 lowBatteryInst ! resume()() .
264                 LowBatteryFromRandomWalkProcess
265               }
266               else {
267                 RandomWalkProcess
268               }
269             }
270           }
271         }
272
273       process DirectedWalkProcess =
274         ? inform()(String attribute2, Integer value2) .
275         if (attribute2 == "victimPos") {
276           owner.victimPos = value2 .
277           DirectedWalkChange
278         }
279         else {
280           if (attribute2 == "atVictim")  {
281             owner.atVictim = value2 .
282             DirectedWalkChange
283           }
284           else {
285             if (attribute2 == "rechargeRequested") {
286               owner.rechargeRequested = value2 .
287               DirectedWalkChange
288             }
289             else {
290               if (attribute2 == "battery") {
291                 owner.battery = value2 .
292                 DirectedWalkChange
```

```
293             }
294             else {
295               DirectedWalkChange
296             }
297           }
298         }
299       }
300     process DirectedWalkChange =
301       if (owner.atVictim == 1) {
302         directedWalkInst ! interrupt()() .
303         rescueInst ! resume()() .
304         RescueProcess
305       }
306       else {
307         if (owner.rechargeRequested == 1) {
308           directedWalkInst ! interrupt()() .
309           rechargeInst ! resume()() .
310           RechargeFromDirectedWalkProcess
311         }
312         else {
313           if (owner.battery < 1) {
314             directedWalkInst ! interrupt()() .
315             lowBatteryInst ! resume()() .
316             LowBatteryFromDirectedWalkProcess
317           }
318           else {
319             DirectedWalkProcess
320           }
321         }
322       }
323
324     process RescueProcess =
325       ? inform()(String attribute3, Integer value3) .
326       if (attribute3 == "victimPos") {
327         owner.victimPos = value3 .
328         RescueChange
329       }
330       else {
331         if (attribute3 == "atVictim")  {
332           owner.atVictim = value3 .
333           RescueChange
334         }
335         else {
336           if (attribute3 == "rechargeRequested") {
337             owner.rechargeRequested = value3 .
338             RescueChange
339           }
340           else {
341             if (attribute3 == "battery") {
342               owner.battery = value3 .
343               RescueChange
344             }
345             else {
346               RescueChange
347             }
348           }
349         }
350       }
351     process RescueChange =
352       if (owner.rechargeRequested == 1) {
353         rescueInst ! interrupt()() .
354         rechargeInst ! resume()() .
355         RechargeFromRescueProcess
356       }
357       else {
358         if (owner.battery < 1) {
359           rescueInst ! interrupt()() .
360           lowBatteryInst ! resume()() .
```

```
361             LowBatteryFromRescueProcess
362           }
363           else {
364             RescueProcess
365           }
366         }
367
368       process RechargeFromRandomWalkProcess =
369         ? inform()(String attribute4, Integer value4) .
370         if (attribute4 == "victimPos") {
371           owner.victimPos = value4 .
372           RechargeFromRandomWalkChange
373         }
374         else {
375           if (attribute4 == "atVictim")  {
376             owner.atVictim = value4 .
377             RechargeFromRandomWalkChange
378           }
379           else {
380             if (attribute4 == "rechargeRequested") {
381               owner.rechargeRequested = value4 .
382               RechargeFromRandomWalkChange
383             }
384             else {
385               if (attribute4 == "battery") {
386                 owner.battery = value4 .
387                 RechargeFromRandomWalkChange
388               }
389               else {
390                 RechargeFromRandomWalkChange
391               }
392             }
393           }
394         }
395       process RechargeFromRandomWalkChange =
396         if (owner.rechargeRequested == 0) {
397           rechargeInst ! interrupt()() .
398           randomWalkInst ! resume()() .
399           RandomWalkProcess
400         }
401         else {
402           if (owner.battery < 1) {
403             rechargeInst ! interrupt()() .
404             lowBatteryInst ! resume()() .
405             LowBatteryFromRechargeFromRandomWalkProcess
406           }
407           else {
408             RechargeFromRandomWalkProcess
409           }
410         }
411
412       process RechargeFromDirectedWalkProcess =
413         ? inform()(String attribute5, Integer value5) .
414         if (attribute5 == "victimPos") {
415           owner.victimPos = value5 .
416           RechargeFromDirectedWalkChange
417         }
418         else {
419           if (attribute5 == "atVictim")  {
420             owner.atVictim = value5 .
421             RechargeFromDirectedWalkChange
422           }
423           else {
424             if (attribute5 == "rechargeRequested") {
425               owner.rechargeRequested = value5 .
426               RechargeFromDirectedWalkChange
427             }
428             else {
```

```
429             if (attribute5 == "battery") {
430               owner.battery = value5 .
431               RechargeFromDirectedWalkChange
432             }
433             else {
434               RechargeFromDirectedWalkChange
435             }
436           }
437         }
438       }
439     process RechargeFromDirectedWalkChange =
440       if (owner.rechargeRequested == 0) {
441         rechargeInst ! interrupt()() .
442         directedWalkInst ! resume()() .
443         DirectedWalkProcess
444       }
445       else {
446         if (owner.battery < 1) {
447           rechargeInst ! interrupt()() .
448           lowBatteryInst ! resume()() .
449           LowBatteryFromRechargeFromDirectedWalkProcess
450         }
451         else {
452           RechargeFromDirectedWalkProcess
453         }
454       }
455
456     process RechargeFromRescueProcess =
457       ? inform()(String attribute6, Integer value6) .
458       if (attribute6 == "victimPos") {
459         owner.victimPos = value6 .
460         RechargeFromRescueChange
461       }
462       else {
463         if (attribute6 == "atVictim")  {
464           owner.atVictim = value6 .
465           RechargeFromRescueChange
466         }
467         else {
468           if (attribute6 == "rechargeRequested") {
469             owner.rechargeRequested = value6 .
470             RechargeFromRescueChange
471           }
472           else {
473             if (attribute6 == "battery") {
474               owner.battery = value6 .
475               RechargeFromRescueChange
476             }
477             else {
478               RechargeFromRescueChange
479             }
480           }
481         }
482       }
483     process RechargeFromRescueChange =
484       if (owner.rechargeRequested == 0) {
485         rechargeInst ! interrupt()() .
486         rescueInst ! resume()() .
487         RescueProcess
488       }
489       else {
490         if (owner.battery < 1) {
491           rechargeInst ! interrupt()() .
492           lowBatteryInst ! resume()() .
493           LowBatteryFromRechargeFromRescueProcess
494         }
495         else {
496           RechargeFromRescueProcess
```

```
          }
        }


      process LowBatteryFromRandomWalkProcess =
        ? inform()(String attribute7, Integer value7) .
        if (attribute7 == "victimPos") {
          owner.victimPos = value7 .
          LowBatteryFromRandomWalkChange
        }
        else {
          if (attribute7 == "atVictim")  {
            owner.atVictim = value7 .
            LowBatteryFromRandomWalkChange
          }
          else {
            if (attribute7 == "rechargeRequested") {
              owner.rechargeRequested = value7 .
              LowBatteryFromRandomWalkChange
            }
            else {
              if (attribute7 == "battery") {
                owner.battery = value7 .
                LowBatteryFromRandomWalkChange
              }
              else {
                LowBatteryFromRandomWalkChange
              }
            }
          }
        }
      process LowBatteryFromRandomWalkChange =
        if (owner.battery == 100) {
          lowBatteryInst ! interrupt()() .
          randomWalkInst ! resume()() .
          RandomWalkProcess
        }
        else {
          LowBatteryFromRandomWalkProcess
        }

      process LowBatteryFromDirectedWalkProcess =
        ? inform()(String attribute8, Integer value8) .
        if (attribute8 == "victimPos") {
          owner.victimPos = value8 .
          LowBatteryFromDirectedWalkChange
        }
        else {
          if (attribute8 == "atVictim")  {
            owner.atVictim = value8 .
            LowBatteryFromDirectedWalkChange
          }
          else {
            if (attribute8 == "rechargeRequested") {
              owner.rechargeRequested = value8 .
              LowBatteryFromDirectedWalkChange
            }
            else {
              if (attribute8 == "battery") {
                owner.battery = value8 .
                LowBatteryFromDirectedWalkChange
              }
              else {
                LowBatteryFromDirectedWalkChange
              }
            }
          }
        }
```

```
565       process LowBatteryFromDirectedWalkChange =
566         if (owner.battery == 100) {
567           lowBatteryInst ! interrupt()() .
568           directedWalkInst ! resume()() .
569           DirectedWalkProcess
570         }
571         else {
572           LowBatteryFromDirectedWalkProcess
573         }
574
575       process LowBatteryFromRescueProcess =
576         ? inform()(String attribute9, Integer value9) .
577         if (attribute9 == "victimPos") {
578           owner.victimPos = value9 .
579           LowBatteryFromRescueChange
580         }
581         else {
582           if (attribute9 == "atVictim")  {
583             owner.atVictim = value9 .
584             LowBatteryFromRescueChange
585           }
586           else {
587             if (attribute9 == "rechargeRequested") {
588               owner.rechargeRequested = value9 .
589               LowBatteryFromRescueChange
590             }
591             else {
592               if (attribute9 == "battery") {
593                 owner.battery = value9 .
594                 LowBatteryFromRescueChange
595               }
596               else {
597                 LowBatteryFromRescueChange
598               }
599             }
600           }
601         }
602       process LowBatteryFromRescueChange =
603         if (owner.battery == 100) {
604           lowBatteryInst ! interrupt()() .
605           rescueInst ! resume()() .
606           RescueProcess
607         }
608         else {
609           LowBatteryFromRescueProcess
610         }
611
612       process LowBatteryFromRechargeFromRandomWalkProcess =
613         ? inform()(String attribute10, Integer value10) .
614         if (attribute10 == "victimPos") {
615           owner.victimPos = value10 .
616           LowBatteryFromRechargeFromRandomWalkChange
617         }
618         else {
619           if (attribute10 == "atVictim")  {
620             owner.atVictim = value10 .
621             LowBatteryFromRechargeFromRandomWalkChange
622           }
623           else {
624             if (attribute10 == "rechargeRequested") {
625               owner.rechargeRequested = value10 .
626               LowBatteryFromRechargeFromRandomWalkChange
627             }
628             else {
629               if (attribute10 == "battery") {
630                 owner.battery = value10 .
631                 LowBatteryFromRechargeFromRandomWalkChange
632               }
```

```
633             else {
634               LowBatteryFromRechargeFromRandomWalkChange
635             }
636           }
637         }
638       }
639     process LowBatteryFromRechargeFromRandomWalkChange =
640       if (owner.battery == 100) {
641         lowBatteryInst ! interrupt()() .
642         rechargeInst ! resume()() .
643         RechargeFromRandomWalkProcess
644       }
645       else {
646         LowBatteryFromRechargeFromRandomWalkProcess
647       }
648
649     process LowBatteryFromRechargeFromDirectedWalkProcess =
650       ? inform()(String attribute11, Integer value11) .
651       if (attribute11 == "victimPos") {
652         owner.victimPos = value11 .
653         LowBatteryFromRechargeFromDirectedWalkChange
654       }
655       else {
656         if (attribute11 == "atVictim")  {
657           owner.atVictim = value11 .
658           LowBatteryFromRechargeFromDirectedWalkChange
659         }
660         else {
661           if (attribute11 == "rechargeRequested") {
662             owner.rechargeRequested = value11 .
663             LowBatteryFromRechargeFromDirectedWalkChange
664           }
665           else {
666             if (attribute11 == "battery") {
667               owner.battery = value11 .
668               LowBatteryFromRechargeFromDirectedWalkChange
669             }
670             else {
671               LowBatteryFromRechargeFromDirectedWalkChange
672             }
673           }
674         }
675       }
676     process LowBatteryFromRechargeFromDirectedWalkChange =
677       if (owner.battery == 100) {
678         lowBatteryInst ! interrupt()() .
679         rechargeInst ! resume()() .
680         RechargeFromDirectedWalkProcess
681       }
682       else {
683         LowBatteryFromRechargeFromDirectedWalkProcess
684       }
685
686     process LowBatteryFromRechargeFromRescueProcess =
687       ? inform()(String attribute12, Integer value12) .
688       if (attribute12 == "victimPos") {
689         owner.victimPos = value12 .
690         LowBatteryFromRechargeFromRescueChange
691       }
692       else {
693         if (attribute12 == "atVictim")  {
694           owner.atVictim = value12 .
695           LowBatteryFromRechargeFromRescueChange
696         }
697         else {
698           if (attribute12 == "rechargeRequested") {
699             owner.rechargeRequested = value12 .
700             LowBatteryFromRechargeFromRescueChange
```

```
701             }
702             else {
703               if (attribute12 == "battery") {
704                 owner.battery = value12 .
705                 LowBatteryFromRechargeFromRescueChange
706               }
707               else {
708                 LowBatteryFromRechargeFromRescueChange
709               }
710             }
711           }
712         }
713       process LowBatteryFromRechargeFromRescueChange =
714         if (owner.battery == 100) {
715           lowBatteryInst ! interrupt()() .
716           rechargeInst ! resume()() .
717           RechargeFromRescueProcess
718         }
719         else {
720           LowBatteryFromRechargeFromRescueProcess
721         }
722     }
723   }
```

Contents of the Attached CD

The attached CD provides the following content:

- file `PhDThesis_Klarl.pdf`: this thesis as PDF,
- folder `publications`:
  the publications listed in chapter "Publications of Annabelle Klarl" as PDF,
- folder `projects`:
  - project `eu.ascens.helena`: the source code of the jHELENA framework,
  - project `eu.ascens.helena.p2p`: the hand-coded Java implementation of the p2p example,
  - file `HelenaText.xtext`: the grammar of the domain-specific language HELENATEXT,
  - projects `eu.ascens.helenaText`, `eu.ascens.helenaText.sdk`, `eu.ascens.helenaText.tests`, and `eu.ascens.helenaText.ui`:
    the source code of the HELENA workbench,
  - project `eu.ascens.helenaText.p2p`: the p2p example the relying on the HELENA workbench consisting of
    - file `src/p2p.helena`: the specification in HELENATEXT,
    - file `promela-gen/p2p.pml`: the generated PROMELA specification,
    - file `promela-gen/p2p-check.pml`: the adapted PROMELA specification for verification,
    - folder `promela-gen/results`: the results of verification,
    - folders `src-gen` and `src-user`: the generated Java implementation together with the hand-coded set-up of the application relying on the jHELENA framework
  - project `eu.ascens.helenaText.scp`: the SCP case study relying on the HELENA workbench consisting of
    - file `src/scp.helena`: the specification in HELENATEXT,
    - file `promela-gen/scp.pml`: the generated PROMELA specification,
    - folder `promela-gen/2nodes`: the adapted PROMELA specification for verification with two nodes and the results of verification,
    - folder `promela-gen/2nodes`: the adapted PROMELA specification for verification with two nodes and the results of verification,
  - file `search-and-rescue.helena`: the specification of the robotic search-and-rescue scenario in HELENATEXT
- folder `plugins`: the HELENA workbench exported as plug-in consisting of the jar-archives `eu.ascens.helenaText.jar` and `eu.ascens.helenaText.ui.jar`,

Publications by Annabelle Klarl

The contents of this thesis rely on eight peer-reviewed publications. I would like to thank all my co-authors for the fruitful collaboration. The following list gives an overview about my own contributions to the publications.

[HK14] Rolf Hennicker and Annabelle Klarl. Foundations for Ensemble Modeling - The Helena Approach. In *Specification, Algebra, and Software*, volume 8373 of *Lecture Notes in Computer Science*, pages 359–381. Springer, 2014
A. Klarl's contribution: idea of the notion of roles for expressing ensembles of collaborating entities, formalization of ensemble structures and their semantics in discussion with Rolf Hennicker

[KH14] Annabelle Klarl and Rolf Hennicker. Design and Implementation of Dynamically Evolving Ensembles with the Helena Framework. In *Australasian Software Engineering Conference*, pages 15–24. IEEE, 2014
A. Klarl's contribution: idea for the realization of roles with threads on top of components, architecture in collaboration with Rolf Hennicker, implementation of the jHELENA framework and the p2p example

[MKH+13] Philip Mayer, Annabelle Klarl, Rolf Hennicker, Mariachiara Puviani, Francesco Tiezzi, Rosario Pugliese, Jaroslav Keznikl, and Tomas Bures. The Autonomic Cloud: A Vision of Voluntary, Peer-2-Peer Cloud Computing. In *International Conference on Self-Adaptation and Self-Organizing Systems Workshops*, pages 89–94. IEEE, 2013
A. Klarl's contribution: idea for the representation of the science cloud platform as an ensemble of collaborating roles

[KMH14] Annabelle Klarl, Philip Mayer, and Rolf Hennicker. Helena@Work: Modeling the Science Cloud Platform. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, volume 8373 of *Lecture Notes in Computer Science*, pages 99–116. Springer, 2014
A. Klarl's contribution: idea for the representation of the science cloud platform as an ensemble of collaborating roles, design of the HELENA model of the science cloud platform, proposal for the modularization of the existing implementation according to a role-based architecture following HELENA, provision of the jHELENA framework as a blueprint for the implementation of the science cloud platform

[KCH14] Annabelle Klarl, Lucia Cichella, and Rolf Hennicker. From Helena Ensemble Specifications to Executable Code. In *International Symposium on Formal Aspects of Component Software*, volume 8997 of *Lecture Notes in Computer Science*, pages 183–190. Springer, 2014

A. Klarl's contribution: design of the domain-specific language HELENATEXT, sketch of the translation from HELENATEXT to jHELENA, supervision of the master thesis of Lucia Cichella realizing HELENATEXT and the translation to jHELENA with the XTEXT workbench

[HKW15] Rolf Hennicker, Annabelle Klarl, and Martin Wirsing. Model-Checking Helena Ensembles with Spin. In *Logic, Rewriting, and Concurrency*, volume 9200 of *Lecture Notes in Computer Science*, pages 331–360. Springer, 2015
A. Klarl's contribution: idea for goal specifications with linear temporal logic, using PROMELA specifications as target verification models and model-checking with Spin, idea and implementation of the translation from HELENA to PROMELA serving as a basis for the formalization of the translation for a simplified variant of HELENA; elaboration of the equivalence proof (sketched by Rolf Hennicker and Martin Wirsing), detection and verification of edge cases based on discussions with Rolf Hennicker and Martin Wirsing

[Kla15b] Annabelle Klarl. From Helena Ensemble Specifications to Promela Verification Models. In *International Symposium on Model Checking Software*, volume 9232 of *Lecture Notes in Computer Science*, pages 39–45. Springer, 2015
A. Klarl's contribution: whole publication

[Kla15a] Annabelle Klarl. Engineering Self-Adaptive Systems with Role-Based Architectures. In *International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 3–8. IEEE, 2015
A. Klarl's contribution: whole publication

Bibliography

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *International Conference on Software Engineering*, pages 187–197. ACM, 2002.

[ABG+13] Rima Al Ali, Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECo Computational Model - I. Technical Report D3S-TR-2013-01, Charles University in Prague, 2013.

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and Analyzing Dynamic Software Architectures. In *International Conference on Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 1998.

[AG94] Robert Allen and David Garlan. Formalizing Architectural Connection. In *International Conference on Software Engineering*, pages 71–80. IEEE, 1994.

[AG97] Robert Allen and David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engingeering and Methodology*, 6(3):213–249, 1997.

[ADL16] Yehia Abd Alrahman, Rocco De Nicola, and Michele Loreti. On the Power of Attribute-Based Communication. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, volume 9688 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2016.

[And97] Egil P. Andersen. Conceptual Modeling of Objects – A Role Modeling Approach. PhD Thesis, University of Oslo, 1997.

[ATS04] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A Survey of Peer-to-peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, 2004.

[BJK+06] Özalp Babaoglu, Márk Jelasity, Anne-Marie Kermarrec, Alberto Montresor, and Maarten van Steen. Managing clouds: a case for a fresh look at large unreliable dynamic networks. *Operating Systems Review*, 40(3):9–13, 2006.

[BMT12] Özalp Babaoglu, Moreno Marzolla, and Michele Tamburini. Design and implementation of a P2P Cloud System. In *Annual ACM Symposium on Applied Computing*, pages 412–417. ACM, 2012.

[BD77] Charles W. Bachman and Manilal Daya. The Role Concept in Data Models. In *International Conference on Very Large Data Bases*, pages 464–476. VLDB Endowment, 1977.

[BV92] Jos C. M. Baeten and Frits W. Vaandrager. An Algebra for Process Creation. *Springer Acta Informatica*, 29(4):303–334, 1992.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[BSI07] Matteo Baldoni, Universita Studi, and Torino Italy. Interaction between Objects in powerJava. *Journal of Object Technology*, 6(2):5–30, 2007.

[BBvP13] Jiří Barnat, Nikola Beneš, Ivana Černá, and Zuzana Petruchová. DCCL: Verification of Component Systems with Ensembles. In *International ACM SIGSOFT Symposium on Component-based Software Engineering*, pages 43–52. ACM, 2013.

[BABC+09] Tomás Barros, Rabéa Ameur-Boulifa, Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Behavioural Models for Distributed Fractal Components. *Springer Annals of Telecommunications*, 64(1):25–43, 2009.

[BBB+12] Ananda Basu, Saddek Bensalem, Marius Bozga, Paraskevas Bourgos, and Joseph Sifakis. Rigorous System Design: The BIP Approach. In *International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, volume 7119 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2012.

[BMO01] Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. In *International Workshop on Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in Computer Science*, pages 91–103. Springer, 2001.

[BHH+06] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A Component Model for Architectural Programming. *Elsevier Electronic Notes in Theoretical Computer Science*, 160:75–96, 2006.

[BGKM99] Michael Becht, Thorsten Gurzki, Jürgen Klarmann, and Matthias Muscholl. ROPE: Role Oriented Programming Environment for Multiagent Systems. In *International Conference on Cooperative Information Systems*, pages 325–333. IEEE, 1999.

[Ber92] Jan Albert Bergstra. A Process Creation Mechanism in Process Algebra. In *Empowering Networks*, pages 81–88. Educational Technology Publications, 1992.

[BK89] Jan Albert Bergstra and Jan Wilem Klop. ACPτ: A Universal Axiom System for Process Specification. In *Algebraic Methods: Theory, Tools and Applications*, volume 394 of *Lecture Notes in Computer Science*, pages 447–463. Springer, 1989.

[BCD00] Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. On the Formalization of Architectural Types with Process Algebras. In *International ACM SIGSOFT Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, volume 25, pages 140–148. ACM, 2000.

[BCD02] Marco Bernardo, Paolo Ciancarini, and Lorenzo Donatiello. Architecting Families of Software Systems with Process Algebras. *ACM Transactions on Software Engingeering and Methodology*, 11(4):386–426, 2002.

[BO98] Conrad Bock and James Odell. A More Complete Model of Relations and Their Implementation: Roles. *Journal of Object-Oriented Programming*, 11(2):51–54, 1998.

[BJMS12] Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling Dynamic Architectures Using Dy-BIP. In *International Conference on Software Composition*, volume 7306 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

[BvVZ06] Luboš Brim, Ivana Černá, Pavlína Vařeková, and Barbora Zimmerova. Component-Interaction Automata as a Verification-Oriented Component-Based System Specification. *ACM SIGSOFT Software Engineering Notes*, 31(2):4, 2006.

[Bro86] Rodney A. Brooks. A Robust Layered Control System For a Mobile Robot. *IEEE Robotics and Automation*, 1986.

[BMSG+09] Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering Self-Adaptive Systems Through Feedback Loops. In *International Symposium on Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 48–70. Springer, 2009.

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema, and Jean-Bernard Stefani. An Open Component Model and Its Support in Java. In *International Symposium on Component-Based Software Engineering*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004.

[BCG+13] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Adaptable Transition Systems. In *International Workshop on Recent Trends in Algebraic Development Techniques*, volume 7841 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2013.

[BGH+13] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECO: An Ensemble-based Component System. In *International ACM SIGSOFT Symposium on Component-based Software Engineering*, pages 81–90. ACM, 2013.

[BGH+15] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. The Invariant Refinement Method. In

Software Engineering for Collective Autonomic Systems, volume 8998 of *Lecture Notes in Computer Science*, pages 405–428. Springer, 2015.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Conference on Software Engineering Research, Management and Applications*, pages 40–48. IEEE, 2006.

[CDLT08] Francesco Calzolai, Rocco De Nicola, Michele Loreti, and Francesco Tiezzi. TAPAs: A Tool for the Analysis of Process Algebras. In *Transactions on Petri Nets and Other Models of Concurrency I*, volume 5100 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2008.

[CKRW99] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An Event-Based Structural Operational Semantics of Multi-threaded Java. In *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer, 1999.

[CJSZ04] Luca Cernuzzi, Thomas Juan, Leon Sterling, and Franco Zambonelli. The Gaia Methodology: Basic Concepts and Extensions. In *Methodologies and Software Engineering for Agent Systems*, volume 11 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 69–88. Springer, 2004.

[CDP+13] Luca Cesari, Rocco De Nicola, Rosario Pugliese, Mariachiara Puviani, Francesco Tiezzi, and Franco Zambonelli. Formalising Adaptation Patterns for Autonomic Ensembles. In *International Symposium on Formal Aspects of Component Software*, volume 8348 of *Lecture Notes in Computer Science*, pages 100–118. Springer, 2013.

[CW09] Abhishek Chandra and Jon Weissman. Nebulas: Using Distributed Voluntary Resources to Build Clouds. In *Workshop on Hot Topics in Cloud Computing*. USENIX Association, 2009.

[Che76] Peter Pin-Shan Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[C+09] Betty H. C. Cheng et al. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

[Cic14] Lucia Cichella. A Domain-Specific Language and Java Code Generation for Ensemble Systems with the Eclipse Language Workbench Xtext. Master Thesis, Ludwig-Maximilians-Universität München, 2014.

[CBK15] Jacques Combaz, Saddek Bensalem, and Jan Kofron. Correctness of Service Components and Service Component Ensembles. In *Software Engineering for Collective Autonomic Systems*, volume 8998 of *Lecture Notes in Computer Science*, pages 107–159. Springer, 2015.

[CDPY15] Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. A Gentle Introduction to Multiparty Asynchronous Session Types. In *International School on Formal Methods for the Design of*

Computer, Communication, and Software Systems, volume 9104 of *Lecture Notes in Computer Science*, pages 146–178. Springer, 2015.

[CDKB11] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley, 5th edition, 2011.

[CDPS09] Vincenzo D. Cunsolo, Salvatore Distefano, Antonio Puliafito, and Marco Scarpa. Cloud@Home: Bridging the Gap between Volunteer and Cloud Computing. In *Conference on Emerging Intelligent Computing Technology and Applications*, volume 5754 of *Lecture Notes in Computer Science*, pages 423–432. Springer, 2009.

[Cuo12] Nguyen Anh Cuong. JPF-LTL. `https://goo.gl/udD6IG`, 2012. accessed 01/08/2016.

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-Directed Requirements Acquisition. *Elsevier Science of Computer Programming*, 20(1-2):3–50, 1993.

[De 16] Rocco De Nicola. SCEL: Service Component Ensemble Language. `http://goo.gl/5EPqU8`, 2016. accessed 01/08/2016.

[DLL+14] Rocco De Nicola, Alberto Lluch-Lafuente, Michele Loreti, Andrea Morichetta, Rosario Pugliese, Valerio Senni, and Francesco Tiezzi. Programming and Verifying Component Ensembles. In *Workshop on From Programs to Systems*, volume 8415 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2014.

[DLPT14] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A Formal Approach to Autonomic Systems Programming: The SCEL Language. *ACM Transactions on Autonomous and Adaptive Systems*, 9(2):1–7, 2014.

[DGH+87] Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Symposium on Principles of Distributed Computing*, pages 1–12. ACM, 1987.

[DY11] Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic Multirole Session Types. In *Symposium on Principles of Programming Languages*, pages 435–446. ACM, 2011.

[DPC+14] Xavier Devroey, Gilles Perrouin, Maxime Cordy, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. State Machine Flattening: Mapping Study and Assessment. *Computing Research Repository*, abs/1403.5398, 2014.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *International Conference on Software Engineering*, pages 411–420. ACM, 1999.

[EMMW15] Jonas Eckhardt, Tobias Mühlbauer, José Meseguer, and Martin Wirsing. Semantics, Distributed Implementation, and Formal Analysis of KLAIM models in Maude. *Elsevier Science of Computer Programming*, 99:24–74, 2015.

[Fos09] Howard Foster. Architecture and Behaviour Analysis for Engineering Service Modes. In *International Workshop on Principles of Engineering Service-Oriented Systems*, pages 1–8. IEEE, 2009.

[FUKM07] Howard Foster, Sebastián Uchitel, Jeff Kramer, and Jeff Magee. Towards Self-Management in Service-Oriented Computing with Modes. In *Service-Oriented Computing Workshop*, volume 4907 of *Lecture Notes in Computer Science*, pages 338–350. Springer, 2007.

[GLMS13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *Springer International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[GMW97] David Garlan, Robert Monroe, and David Wile. Acme: An Architecture Description Interchange Language. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 169–183. IBM Press, 1997.

[Gat98] Erann Gat. On Three-Layer Architectures. In *Artificial Intelligence and Mobile Robots*, pages 195–210. MIT Press, 1998.

[GSR96] Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996.

[GM14] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.

[GWGvS04] Giancarlo Guizzardi, Gerd Wagner, Nicola Guarino, and Marten van Sinderen. An Ontologically Well-Founded Profile for UML Conceptual Models. In *International Conference on Advanced Information Systems Engineering*, volume 3084 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2004.

[HWH14] Robrecht Haesevoets, Danny Weyns, and Tom Holvoet. Architecture-Centric Support for Adaptive Service Collaborations. *ACM Transaction on Software Engineering and Methodology*, 23:2:1–2:40, 2014.

[Hal09] Terry Halpin. Object-role modeling. In *Encyclopedia of Database Systems*, pages 1941–1946. Springer, 2009.

[HL93] Klaus Havelund and Kim Guldstrand Larsen. The Fork Calculus. In *International Colloquium on Automata, Languages and Programming*, volume 700 of *Lecture Notes in Computer Science*, pages 544–557. Springer, 1993.

[HK14] Rolf Hennicker and Annabelle Klarl. Foundations for Ensemble Modeling - The Helena Approach. In *Specification, Algebra, and Software*, volume 8373 of *Lecture Notes in Computer Science*, pages 359–381. Springer, 2014.

[HKW15] Rolf Hennicker, Annabelle Klarl, and Martin Wirsing. Model-Checking Helena Ensembles with Spin. In *Logic, Rewriting, and Concurrency*, volume 9200 of *Lecture Notes in Computer Science*, pages 331–360. Springer, 2015.

[HK11] Rolf Hennicker and Alexander Knapp. Modal Interface Theories for Communication-Safe Component Assemblies. In *International Colloquium on Theoretical Aspects of Computing*, volume 6916 of *Lecture Notes in Computer Science*, pages 135–153. Springer, 2011.

[Her03] Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *International Conference on Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2003.

[HKKR05] Martin Hitz, Gerti Kappel, Elisabeth Kapsammer, and Werner Retschitzegger. *UML@Work - Objektorientierte Modellierung mit UML 2.* dpunkt Verlag, 3. auflage edition, 2005.

[Hoa78] Charles Anthony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol03] Gerard Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[Hol12] Gerard Holzmann. Parallelizing the Spin Model Checker. In *International Conference on Model Checking Software*, volume 7385 of *Lecture Notes in Computer Science*, pages 155–171. Springer, 2012.

[HB07] Gerard Holzmann and Dragan Bosnacki. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 3rd edition edition, 2006.

[IBM06] IBM Corporation. An architectural blueprint for autonomic computing. `http://goo.gl/5Lo5AO`, 2006. accessed 01/08/2016.

[IW14] Usman Iftikhar and Danny Weyns. ActivFORMS: Active Formal Models for Self-Adaptation. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 125–134. ACM, 2014.

[Jan10] Stephan Janisch. Behaviour and Refinement of Port-Based Components with Synchronous and Asynchronous Communication. Dissertation Thesis, Ludwig-Maximilians-Universität München, 2010.

[JSHS96] Ralf Jungclaus, Gunter Saake, Thorsten Hartmann, and Cristina Sernadas. TROLL - A Language for Object-Oriented Specification of Information Systems. *ACM Transactions on Information Systems*, 14(2):175–211, 1996.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[Kla15a] Annabelle Klarl. Engineering Self-Adaptive Systems with Role-Based Architectures. In *International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 3–8. IEEE, 2015.

[Kla15b] Annabelle Klarl. From Helena Ensemble Specifications to Promela Verification Models. In *International Symposium on Model Checking Software*, volume 9232 of *Lecture Notes in Computer Science*, pages 39–45. Springer, 2015.

[KCH14] Annabelle Klarl, Lucia Cichella, and Rolf Hennicker. From Helena Ensemble Specifications to Executable Code. In *International Symposium on Formal Aspects of Component Software*, volume 8997 of *Lecture Notes in Computer Science*, pages 183–190. Springer, 2014.

[KH14] Annabelle Klarl and Rolf Hennicker. Design and Implementation of Dynamically Evolving Ensembles with the Helena Framework. In *Australasian Software Engineering Conference*, pages 15–24. IEEE, 2014.

[KMH14] Annabelle Klarl, Philip Mayer, and Rolf Hennicker. Helena@Work: Modeling the Science Cloud Platform. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, volume 8373 of *Lecture Notes in Computer Science*, pages 99–116. Springer, 2014.

[KWA+01] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. SETI@home-Massively Distributed Computing for SETI. *IEEE Computing in Science and Engineering*, 3(1):78–83, 2001.

[KM07] Jeff Kramer and Jeff Magee. Self-Managed Systems: An Architectural Challenge. In *Future of Software Engineering*, pages 259–268. IEEE, 2007.

[KM09] Jeff Kramer and Jeff Magee. A Rigorous Architectural Approach to Adaptive Software Engineering. *Springer Journal of Computer Science and Technology*, 24(2):183–188, 2009.

[KØ96] Bent Bruun Kristensen and Kasper Østerbye. Roles: Conceptual Abstraction Theory and Practical Language Issues. *Wiley Theory and Practice of Object Systems*, 2(3):143–160, 1996.

[KLG+14] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A Metamodel Family for Role-Based Modeling and Programming Languages. In *International Conference on Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 2014.

[Lam83] Leslie Lamport. What Good is Temporal Logic? In *IFIP 9th World Congress*, pages 657–668, 1983.

[Lau16] NASA Official Sonie Lau. JPF .. the swiss army knife of Java verification. `http://goo.gl/reTQe`, 2016. accessed 01/08/2016.

[Lor16] Michele Loreti. jRESP: Java Runtime Environment for SCEL Programs. `http://goo.gl/LruK8X`, 2016. accessed 01/08/2016.

[LE13] Markus Luckey and Gregor Engels. High-Quality Specification of Self-Adaptive Software Systems. In *International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 143–152. IEEE, 2013.

[MK96] Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, volume 21, pages 3–14. ACM, 1996.

[MK06] Jeff Magee and Jeff Kramer. *Concurrency-State Models and Java Programs*. Wiley, 2006.

[MMPT14] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. Developing and Enforcing Policies for Access Control, Resource Usage, and Adaptation. In *International Workshop on Web Services and Formal Methods*, volume 8379 of *Lecture Notes in Computer Science*, pages 85–105. Springer, 2014.

[MPT13] Andrea Margheri, Rosario Pugliese, and Francesco Tiezzi. Linguistic Abstractions for Programming and Policing Autonomic Computing Systems. In *International Conference on Ubiquitous Intelligence and Computing*, pages 404–409. IEEE, 2013.

[MKH+13] Philip Mayer, Annabelle Klarl, Rolf Hennicker, Mariachiara Puviani, Francesco Tiezzi, Rosario Pugliese, Jaroslav Keznikl, and Tomas Bures. The Autonomic Cloud: A Vision of Voluntary, Peer-2-Peer Cloud Computing. In *International Conference on Self-Adaptation and Self-Organizing Systems Workshops*, pages 89–94. IEEE, 2013.

[MVK+15] Philip Mayer, José Velasco, Annabelle Klarl, Rolf Hennicker, Mariachiara Puviani, Francesco Tiezzi, Rosario Pugliese, Jaroslav Keznikl, and Tomáš Bureš. *The Autonomic Cloud*, volume 8998 of *Lecture Notes in Computer Science*, pages 495–512. Springer, 2015.

[MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report Special Publication 800-145, NIST - National Institute of Standards and Technology, 2011.

[MPT12] Emanuela Merelli, Nicola Paoletti, and Luca Tesei. A Multi-Level Model for Self-Adaptive Systems. In *International Workshop on Foundations of Coordination Languages and Self Adaptation*, volume 91, pages 112–126. EPCTS, 2012.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[MT11] Supasit Monpratarnchai and Tamai Tetsuo. Applying Adaptive Role-Based Model to Self-Adaptive System Constructing Problems: A Case Study. In *International Conference on Engineering of Autonomic and Autonomous Systems*, pages 69–78. IEEE, 2011.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[PBF+11] Fabio Panzieri, Özalp Babaoglu, Stefano Ferretti, Vittorio Ghini, and Moreno Marzolla. Distributed Computing in the 21st Century: Some Aspects of Cloud Computing. In *Dependable and Historic Computing*, volume 6875 of *Lecture Notes in Computer Science*, pages 393–412. Springer, 2011.

[PCL14] Mariachiara Puviani, Giacomo Cabri, and Letizia Leonardi. Enabling Self-Expression: the Use of Roles to Dynamically Change Adaptation Patterns. In *International Conference on Self-Adaptive and Self-Organizing Systems*, pages 14–19. IEEE, 2014.

[PCZ13] Mariachiara Puviani, Giacomo Cabri, and Franco Zambonelli. A Taxonomy of Architectural Patterns for Self-adaptive Systems. In *International C* Conference on Computer Science and Software Engineering*, pages 77–85. ACM, 2013.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*. Springer, 2008.

[Ree96] Trygve Reenskaug. *Working with Objects: The OOram Framework Design Principles*. Manning Publications, 1996.

[RD01a] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Symposium on Operating Systems Principles*, pages 188–201. ACM, 2001.

[RD01b] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *International Conference on Distributed Systems Platforms*, volume 2218 of *Lecture Notes in Computer Science*, pages 329–350. Springer, 2001.

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2), 2009.

[SD96] Monique Snoeck and Guido Dedene. Generalization/Specialization and Role in Object Oriented Conceptual Modeling. *Elsevier Data Knowledge Engineering*, 19(2):171–195, 1996.

[SWHB05] Elke Steegmans, Danny Weyns, Tom Holvoet, and Yolande Berbers. A Design Process for Adaptive Behavior of Situated Agents. In *International Conference on Agent-Oriented Software Engineering*, volume 3382 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2005.

[Ste00a] Friedrich Steimann. Formale Modellierung mit Rollen. Habilitation Thesis, Universität Hannover, 2000.

[Ste00b] Friedrich Steimann. On the Representation of Roles in Object-Oriented and Conceptual Modelling. *Elsevier Data and Knowledge Engineering*, 35(1):83–106, 2000.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002.

[TUI07] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. Objects as Actors Assuming Roles in the Environment. In *Software Engineering for Multi-Agent Systems*, volume 4408 of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2007.

[tEKR03] Maurice H. ter Beek, Clarence A. Ellis, Jetty Kleijn, and Grzegorz Rozenberg. Synchronizations in Team Automata for Groupware Systems. *Springer Computer Supported Cooperative Work*, 12(1):21–69, 2003.

[vL01] Axel van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *International Symposium on Requirements Engineering*, pages 249–262. IEEE, 2001.

[vL03] Axel van Lamsweerde. From System Goals to Software Architecture. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems: Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 25–43. Springer, 2003.

[vL09] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Springer Automated Software Engineering*, 10(2):203–232, 2003.

[WTM04] Fang Wang, Sofiène Tahar, and Otmane Aït Mohamed. First-Order LTL Model Checking Using MDGs. In *Automated Technology for Verification and Analysis*, volume 3299 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2004.

[Was04] Andrzej Wasowski. Flattening Statecharts Without Explosions. In *Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 257–266. ACM, 2004.

[Wei97] Carsten Weise. An Incremental Formal Semantics for PROMELA. In *Third SPIN Workshop*, 1997.

[Wei99] Mark Weiser. The Computer for the 21st Century. *ACM SIGMOBILE Mobile Computing and Communications Review*, 3(3):3–11, 1999.

[WMA12] Danny Weyns, Sam Malek, and Jesper Andersson. FORMS: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems. *ACM Transactions on Autonomous and Adaptive Systems*, 7(1):8:1–8:61, 2012.

[WHKM15] Martin Wirsing, Matthias Hölzl, Nora Koch, and Philip Mayer. *Software Engineering for Collective Autonomic Systems*, volume 8998 of *Lecture Notes in Computer Science*. Springer, 2015.

[WJK00] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Springer Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[XZP07] Haiping Xu, Xiaoqin Zhang, and Rinkesh J. Patel. Developing Role-Based Open Multi-Agent Software Systems. *International Journal of Computational Intelligence Theory and Practice*, 2, 2007.

[XSCM04] Ying Xu, Xiaoyu Song, Eduard Cerny, and Otmane Aït Mohamed. Model Checking for a First-Order Temporal Logic Using Multiway Decision Graphs. *Elsevier The Computer Journal*, 47(1):71–84, 2004.

[YHNN13] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The Scribble Protocol Language. In *International Symposium Trustworthy Global Computing*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013.

[ZC06] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *International Conference on Software Engineering*, pages 371–380. ACM, 2006.

[ZGC09] Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng. Modular Verification of Dynamically Adaptive Systems. In *International Conference on Aspect-oriented Software Development*, pages 161–172. ACM, 2009.

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud Computing: State-of-the-Art and Research Challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.

[ZMLL11] Yongwang Zhao, Dianfu Ma, Jing Li, and Zhuqing Li. Model Checking of Adaptive Programs with Mode-extended Linear Temporal Logic. In *International Conference on Engineering of Autonomic and Autonomous Systems*, pages 40–48. IEEE, 2011.