Simulation and Statistical Model-Checking of Logic-Based Multi-Agent System Models

Christian Kroiß

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

eingereicht von
Christian Kroiß

München, 14. Juni 2016

Erstgutachter: Prof. Dr. Martin Wirsing
Zweitgutachter: Doc. Ing. Petr Tůma, Dr.
Tag der mündlichen Prüfung: 28. Juli 2016

Eidesstattliche Versicherung

(Siehe Promotionsordnung vom 12.07.11, § 8, Abs. 2 Pkt. .5.)

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

Kroiß, Christian

Name, Vorname

München, 14.06.2016	
Ort, Datum	Unterschrift Doktorand/in

Formular 3.2

Abstract

This thesis presents SALMA (**S**imulation and **A**nalysis of **L**ogic-Based **M**ulti-**A**gent Models), a new approach for simulation and statistical model checking of multi-agent system models.

Statistical model checking is a relatively new branch of model-based approximative verification methods that help to overcome the well-known scalability problems of exact model checking. In contrast to existing solutions, SALMA specifies the mechanisms of the simulated system by means of logical axioms based upon the well-established situation calculus. Leveraging the resulting first-order logic structure of the system model, the simulation is coupled with a statistical model-checker that uses a first-order variant of time-bounded linear temporal logic (LTL) for describing properties. This is combined with a procedural and process-based language for describing agent behavior. Together, these parts create a very expressive framework for modeling and verification that allows direct fine-grained reasoning about the agents' interaction with each other and with their (physical) environment.

SALMA extends the classical situation calculus and linear temporal logic (LTL) with means to address the specific requirements of multi-agent simulation models. In particular, cyber-physical domains are considered where the agents interact with their physical environment. Among other things, the thesis describes a generic situation calculus axiomatization that encompasses sensing and information transfer in multi agent systems, for instance sensor measurements or inter-agent messages. The proposed model explicitly accounts for real-time constraints and stochastic effects that are inevitable in cyber-physical systems.

In order to make SALMA's statistical model checking facilities usable also for more complex problems, a mechanism for the efficient on-the-fly evaluation of first-order LTL properties was developed. In particular, the presented algorithm uses an interval-based representation of the formula evaluation state together with several other optimization techniques to avoid unnecessary computation.

Altogether, the goal of this thesis was to create an approach for simulation and statistical model checking of multi-agent systems that builds upon well-proven logical and statistical foundations, but at the same time takes a pragmatic software engineering perspective that considers factors like usability, scalability, and extensibility. In fact, experience gained during several small to mid-sized experiments that are presented in this thesis suggest that the SALMA approach seems to be able to live up to these expectations.

Zusammenfassung

In dieser Dissertation wird SALMA (**S**imulation and **A**nalysis of **L**ogic-Based **M**ulti-**A**gent Models) vorgestellt, ein im Rahmen dieser Arbeit entwickelter Ansatz für die Simulation und die statistische Modellprüfung (Model Checking) von Multiagentensystemen.

Der Begriff „Statistisches Model Checking" beschreibt modellbasierte approximative Verifikationsmethoden, die insbesondere dazu eingesetzt werden können, um den unvermeidlichen Skalierbarkeitsproblemen von exakten Methoden zu entgehen. Im Gegensatz zu bisherigen Ansätzen werden in SALMA die Mechanismen des simulierten Systems mithilfe logischer Axiome beschrieben, die auf dem etablierten Situationskalkül aufbauen. Die dadurch entstehende prädikatenlogische Struktur des Systemmodells wird ausgenutzt um ein Model Checking Modul zu integrieren, das seinerseits eine prädikatenlogische Variante der linearen temporalen Logik (LTL) verwendet. In Kombination mit einer prozeduralen und prozessorientierten Sprache für die Beschreibung von Agentenverhalten entsteht eine ausdrucksstarke und flexible Plattform für die Modellierung und Verifikation von Multiagentensystemen. Sie ermöglicht eine direkte und feingranulare Beschreibung der Interaktionen sowohl zwischen Agenten als auch von Agenten mit ihrer (physischen) Umgebung.

SALMA erweitert den klassischen Situationskalkül und die lineare temporale Logik (LTL) um Elemente und Konzepte, die auf die spezifischen Anforderungen bei der Simulation und Modellierung von Multiagentensystemen ausgelegt sind. Insbesondere werden cyber-physische Systeme (CPS) unterstützt, in denen Agenten mit ihrer physischen Umgebung interagieren. Unter anderem wird eine generische, auf dem Situationskalkül basierende, Axiomatisierung von Prozessen beschrieben, in denen Informationen innerhalb von Multiagentensystemen transferiert werden – beispielsweise in Form von Sensor-Messwerten oder Netzwerkpaketen. Dabei werden ausdrücklich die unvermeidbaren stochastischen Effekte und Echtzeitanforderungen in cyber-physischen Systemen berücksichtigt.

Um statistisches Model Checking mit SALMA auch für komplexere Problemstellungen zu ermöglichen, wurde ein Mechanismus für die effiziente Auswertung von prädikatenlogischen LTL-Formeln entwickelt. Insbesondere beinhaltet der vorgestellte Algorithmus eine Intervall-basierte Repräsentation des Auswertungszustands, sowie einige andere Optimierungsansätze zur Vermeidung von unnötigen Berechnungsschritten.

Insgesamt war es das Ziel dieser Dissertation, eine Lösung für Simulation und statistisches Model Checking zu schaffen, die einerseits auf fundierten logischen und statistischen Grundlagen aufbaut, auf der anderen Seite jedoch auch pragmatischen Gesichtspunkten wie Benutzbarkeit oder Erweiterbarkeit genügt. Tatsächlich legen erste Ergebnisse und Erfahrungen aus mehreren kleinen bis mittelgroßen Experimenten nahe, dass SALMA diesen Zielen gerecht wird.

Acknowledgements

First of all, I would like to thank my supervisor Martin Wirsing for his strong support during my time at the chair for Programming and Software Engineering (PST). This included not only academic advice but also the creation of an encouraging, creative, and friendly work environment for all members of the chair. Especially important for me was the patience, trust, and flexibility that made it so much easier to arrange my work with my life as a father of two young children.

Additionally, I would like to thank Petr Tůma, who did not hesitate to agree to be the second referee for this thesis. I can well image how high his workload is due to his many academic activities, so I really appreciate the fact that he accepted this additional duty.

I am also particularly grateful to Tomáš Bureš, who acted as a substitute professor at the PST chair from 2013 to 2015. Despite his full schedule, he found time to collaborate with me on an extension of my approach that later became one of the main contributions of my thesis.

Furthermore, I want to thank my colleagues of the PST team, who have always been supportive in so many ways. It all began with Nora Koch and Alexander Knapp, who introduced me to the world of academic research during my diploma thesis. Through the kind and inspiring way with which they let me participate in their work, they are to a large degree responsible for my decision to join the PST group. Also, I am especially grateful to Matthias Hölzl, who introduced me to GoLog and the situation calculus and to many other aspects of logics, planning, and probability theory. Similarly, I want to thank Annabelle Klarl, Andreas Schroeder, Philip Mayer, Lenz Belzner, and Marianne Busch, who worked together with me during the research projects REFLECT, ASCENS, and MAEWA. The countless inspiring and open-minded conversations with them really sharpened my view, strengthened my confidence, and helped me navigate through the sometimes overwhelming amount of ideas and possible research directions. Alongside them, Rolf Hennicker and Anton Fasching have contributed essentially to the warm and positive environment at the PST chair with their overall helpfulness and kindness.

Finally, my deepest gratitude goes to my family for their love, support, and most of all patience throughout all these years. In particular, none of this would have been possible without the help, understanding, and encouragement of my wife Nici, whom I love with all my heart.

Contents

Chapter 1

Introduction

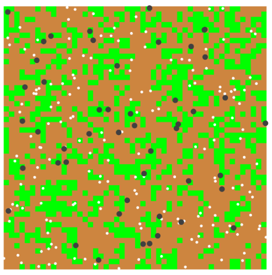
Since the beginning of the modern era of information technology, computer-based simulation has always been a very important application field. The newly gained ability to run large numbers of even complex simulations within acceptable time-frames and with affordable costs had significant impact on the working methods in many fields of science and technology. On the one hand, simulation-based approaches can be used to generate approximative numerical solutions for problems that are too complex to solve analytically, for instance in optimization scenarios. On the other hand, a researcher can inspect the output of a simulation, which is usually visualized in some form, to gain a deeper insight into the behavior of a system. This can help to recognize patterns and interrelations that have not been understood before, in particular by running iterative simulations with systematically varied parameters. Another common task is to use simulations for the validation of a model by comparing the output of simulations of this model with expected results. In particular, if the simulated model represents a concrete system or a class of systems that actually exist in the real world, then the simulation results can be compared with collected observations in order to assess the accuracy of the model. Another typical use case for simulations is to check whether the simulated system can be assumed to fulfill certain requirements. In engineering disciplines like car manufacturing or avionics, this kind of simulation-based validation has become an inevitable part of the production process.

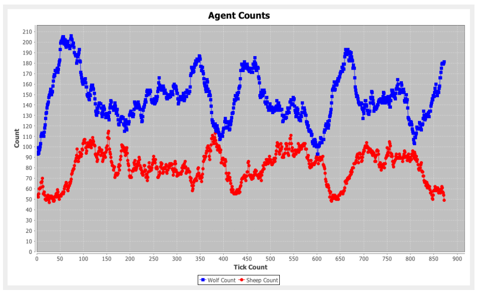
In contrast, a different general way of verifying models has emerged throughout the last decades: automatic model checking. There, the computer analyzes the *state space*, i.e. the set of states that the system can reach according to the model, and attempts to prove or refute that a model fulfills a formally specified property. The main advantage with respect to any kind of simulation-based validation is that model checking produces exact results, i.e. if a model checker does not find a violation of a property, this is understood as a proof that the model contains no violation at all – although this obviously assumes the correctness of the model checker itself. Furthermore, all common automatic model checking tools allow specifying the properties they assert in some kind of temporal logic[Pnu77a, CE82], which allows for a rigorous and concise description not only of desired properties of system *states* but also of temporal paths the system runs through.

However, despite of the rapid improvement of computer system performance and the development of more and more efficient algorithms, the state space of more complex models can get too large to be searched exhaustively. In these cases, exact model checking approaches cannot be applied without introducing significant abstractions in order to reduce the number of states. However, finding appropriate abstractions is often a difficult task that requires much experience by the modeler. In fact, in some cases it might just not be clear at all which details can be omitted without risking to wrongfully dismiss important effects. Consequently, one branch of approaches that has gained much attention recently is *statistical model checking*[LDB10], which attempts to combine the idea behind model checking with the scalability of simulation approaches. Simply put, this is achieved by treating the verified properties as statistical hypotheses that are then tested using data collected through simulations.

The scalability of verification approaches is particularly relevant for *concurrent system* since their state space is effectively a product of the state spaces of the participating processes. This leads to an effect that is known as the state-space explosion problem, which describes the fact that the state space of concurrent system models can become to large to be searched thoroughly when more constituents are added. One broad class of concurrent systems where simulations play a particularly important role are *multi-agent systems* (see, e.g. [Woo09]) in which the active constituents, i.e. agents, are viewed as autonomous actors that are able to make their own decisions and act according to them. On the one hand, this abstract concept of an agent includes artificial entities, such as robots or some other computerized units that can be seen as able to make their own more or less autonomous decisions. On the other hand agents might represent actual life forms, including human beings, whose behavior is modeled in a way that captures the aspects that are important for the simulation experiment. The latter kind of multi-agent simulation has been an important tool in scientific disciplines like sociology, biology, or economics for several decades now. Besides that, the simulation of artificial multi-agent systems has gained importance especially due to the advances in fields like swarm robotics [Şah05] or road vehicle automation, for instance Google's Self-Driving Car Project [Goo16].

One important distinction between multi-agent system models can be made with respect to whether they apply a more *macroscopic* or a more *microscopic* viewpoint. In a macroscopic multi-agent model, agents of the same type are treated as indistinguishable from each other, i.e. the simulation is not interested in the behavior of a particular agent but in the aggregated results that emerge from the interaction of many agents. For instance, the two pictures in Figure 1.1 are a visual representation of a state in a macroscopic simulation of a simple predator-prey scenario in the simulation software Repast Symphony [NCO+13]. The model consists of two types of agents: predators (wolves), and prey (sheep), who act according to a set of simple rules. On the left side, the current state of the simulation is shown by means of a grid whose cells either represent wolves, sheep, or food for the sheep (e.g. grass). In each step of the simulation, both wolves and sheep move to one of the adjacent cells in which they might find a consumable food item, i.e. a sheep or a pile of grass. All agents have an energy level that decreases when they move and increases when they consume food. An agent dies when it is either consumed by a predator or its energy level drops below zero. On the other hand, both sheep and wolves reproduce with a constant rate, i.e. each agent may spawn a new offspring agent in each step with a certain probability. For a model like this, it is unlikely that a researcher would be interested in tracing the behavior of a single sheep or wolf. Instead, what is typically analyzed is the development of the populations of predators and prey, i.e. the agent numbers. For the current example, these agent counts are shown in the right part of Figure 1.1.


(a) 2D visualization of the current state.



(b) Agent counts (sheep vs. wolves).

Figure 1.1: Output of a macroscopic predator-prey simulation in Repast Symphony.

In contrast to macroscopic models like the one described above, other problems require a *microscopic* approach in which the behavior of particular agents is distinguished and can be analyzed separately. One example for such a scenario can be seen in Figure 1.2, which stems from a case study that will be covered more thoroughly in Chapter 6. The example describes an urban traffic scenario in which parking lots for electric vehicles are a assigned by means of a largely automated reservation system. This system consists of a coordinator, denoted as super autonomous manager (SAM), many parking lots with charging stations (PLCS) and the vehicles themselves, which are equipped with navigation systems and on-board computers. These constituents are modeled as autonomous agents that communicate with each other via some wireless connection. The diagram in Figure 1.2 gives an overview of the communication flow: vehicles send requests for parking lot reservations that consist of a time slot and parking lot preferences, which would typically be based on the proximity to some location that the vehicle's driver wants to reach. The SAM agent keeps track of the available parking places of all PLCS, calculates assignments for reservation requests and sends these assignments back to the vehicles, which in turn initiate a separate conversation with the assigned PLCS to make the reservation definite. The idea behind the fact that the SAM does not "order" the assigned PLCS to accept the reservation is to make the protocol more robust with respect to messages being lost or drivers changing their plans.

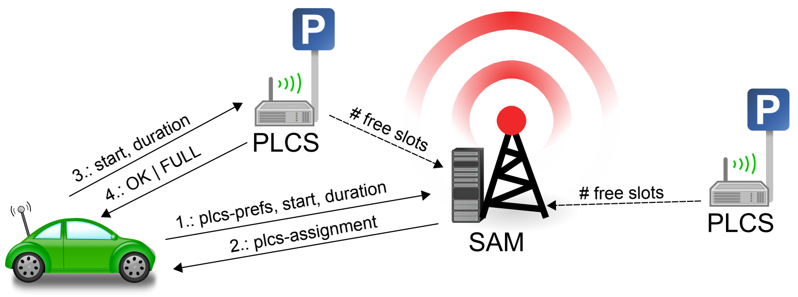


Figure 1.2: Microscopic model for an optimized parking lot assignment scenario.

It is clear that when a system like the one shown in Figure 1.2 is simulated, the distinction between individual agents is inevitable for the analysis of the system. For instance, one of the most important questions that a researcher might want to answer through a simulation experiment is how long a driver has to wait on average until a reservation request is answered. In order to measure these times, the simulation has to distinguish the agents and keep track of their complete communication. Furthermore, it is certainly also important to include the quality of the PLCS assignment in the analysis, i.e. how well the driver's preferences are met.

It turns out that the formal languages that are typically used by simulation and model checking tools are not well suited to describe a scenario like the one above in a way that is formally precise but at the same time accessible enough to be applied also to more complex problems. One of the issues is that the property-specification languages that are used in model-checking tools are typically based on propositional temporal logics like LTL and CTL (see Section 2.4). However, for describing properties of relations and interactions between the constituents of a multi-agent system, it is much more natural to think in terms of first-order logic (FOL) models. While some property specification languages that are used in (statistical) model checking tools provide features like simple quantification over processes or arrays of primitive values, they do not provide full direct support for model-specific functions and relations. Therefore, the modeler is forced to introduce abstractions and indirections that can easily lead to a more and more obfuscated model in which the originally intended properties are no longer visible on first sight.

Besides difficulties in specifying the requirements for a scenario like the one above, the construction of the *system model* can itself become a complicated and error-prone task. One obvious source for complexity that is inherent to any multi-agent system is the fact that there are many interactions between agents that may happen at the same time. If a microscopic view is applied as described above, this typically implies that the situations of individual agents have to be considered both for determining which interactions happen at a particular point in time and for calculating their outcomes. Furthermore, agents can also interact with their *environment*, which could include the physical world around them but also more abstract virtual entities that are shared between agents like a database or a network. This interaction with the environment can mean that agents react to *events* that originate from the environment. At the same time, actions performed by agents can change the state of the environment, which in turn might trigger events or influence other agents. It is obvious that a modeling approach that is suitable in such a setting has to provide means to capture a large number of effects and constraints whose scope ranges over the whole system. Doing this in an imperative programming language can lead to code that is very hard to comprehend and reason about. This can be experienced in typical simulation frameworks for general purpose languages like C/C++, Java, or Python that are designed for optimized performance. A viable alternative for describing the effects of interactions and events on the state of agents and the environment are *rule-based languages* where the outcome of a rule typically determines the value of one or more state variables for the next time step. In fact, the modeling languages that are employed by model checking tools are typically rule-based. However, for agents with more complex control logic, a purely rule-based representation of their behavior can be just as hard to follow. Altogether, the right solution appears to be a proper combination of several paradigms.

Summarizing the mentioned challenges for modeling and verification of multi-agent systems, it is clear that it is hard to find a solution that fits in all circumstances. In particular, compromises have to be made with respect to the following factors:

1. The exactness of the assertions made by the tool, ranging from verdicts that can be understood as proofs as in classical model checking to statistical approximation with varying confidence.

2. Scalability to large and complex models, i.e. whether it is possible to utilize additional computational resources in order to handle larger state spaces.

3. The level of abstraction of the model, i.e. how much detail of the modeled can be captured.

4. The conciseness and clarity of the modeling language. This is obviously also related to scalability towards more complex models since a modeling approach that lacks these qualities will produce incomprehensible and unmaintainable models for complex problems.

This thesis introduces SALMA (Simulation and Analysis of Logic-Based Multi-Agent Systems), a solution for modeling, simulation, and statistical model checking of multi-agent systems that strives to provide as much flexibility as possible so that such a compromise can be made in a way that suits the situation as well as possible. It does that by combining several paradigms and techniques, in particular logical modeling, logic programming, object oriented programming, and statistical model checking. The following chapters will show that the resulting set of modeling languages and the underlying software framework can be applied to a wide range of different domains in a pragmatic way that lets the modeler choose quite freely, which abstraction level to use for particular aspects of the system. At the same time, the SALMA approach is grounded on a solid logical foundation that allows precise reasoning about the mechanisms of the modeled system. A first high-level overview of how this works is presented in he next section.

1.1 An Overview of the SALMA Approach

The SALMA (Simulation and Analysis of Logic-Based Multi-Agent Systems) approach that is presented in this thesis attempts to be a pragmatic and flexible solution for simulation and statistical model checking of multi-agent systems. One of its core goals is to provide the modeler with as much freedom as possible in choosing the right level of detail for describing both the system behavior and the properties that should be used for analysis. On the one hand, this implies that it should be possible to add and change details to the system model with preferably little effort, and without the model getting overloaded and hard to comprehend. As mentioned in the last section, rule-based modeling languages are attractive in this regard as they allow treating different effects within the system separately. On the other hand, the used property specification language should be able to access any detail of the system model as directly as possible in order to make requirement violations traceable and to facilitate keeping the requirements up-to-date with the evolving model.

The approach that SALMA takes to achieve this high cohesion between system model and requirements model is to use first-order logic (FOL) as a common foundation for both. First of all, this means that a first-order version of linear temporal logic (LTL) is used as the basis for SALMA's property specification language (SALMA-PSL), which, as pointed out before, allows for a more direct representation of properties than traditional propositional LTL. Additionally, the core of the system model (the domain model) itself is based on the *situation calculus*[Rei01], a well-established methodology for modeling dynamic systems with first-order logic. Both the core system model and property evaluation subsystem are implemented in Prolog and allow the modeler to leverage the full facilities of logic programming. The *domain* model, i.e. the general mechanisms of the simulated world, is described by means of situation calculus axioms that are encoded in Prolog. Based on this axiomatization, the modeler defines the behavior of agents by equipping them with one or multiple processes that can be defined using SALMA's procedural process definition language (SALMA-PDL). Realized as an internal domain specific language (DSL) [Fow10] within Python, the SALMA-PDL offers the usual control flow constructs like loops and conditionals. At the same time, it provides means to access the underlying situation calculus model, in particular by performing actions, querying the system state, and observing events.

With the models for the domain and the agents' behavior in place, a concrete *simulation experiment* is configured by defining initial values for all system variables and *probability distributions*, for instance for the occurrence of events. This simulation experiment is then used as input for SALMA's Python-based simulation engine, which interprets the agent processes, chooses which events should occur at each time step and updates the system model state according to the situation calculus axioms of the domain model.

This structure of the SALMA approach is outlined in Figure 1.3.

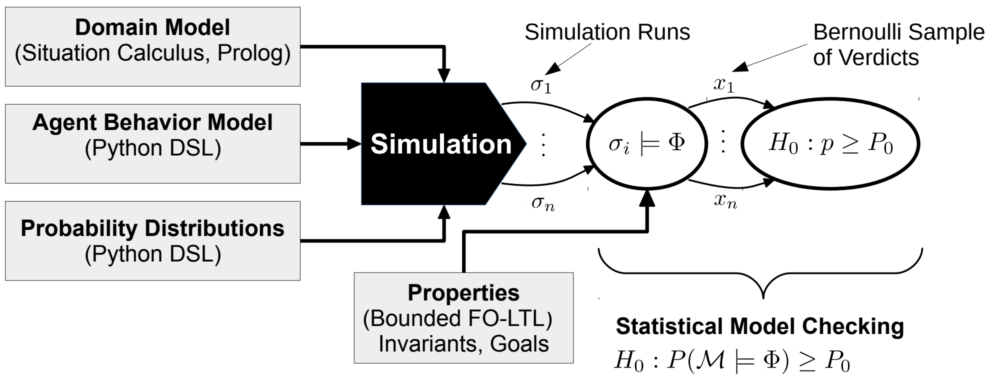


Figure 1.3: Overview of the SALMA approach.

For each simulation run, the engine eventually decides whether it satisfies the properties which the modeler has specified using the SALMA-PSL as described above. The set of resulting verdicts yields a *Bernoulli sample* that is used to test the statistical hypothesis $H_0 : p \geq P_0$ which asserts that the probability of a success (a run fulfills the property) is at least as high as a given lower bound. This way of approximative assertion of properties defined by temporal logics is generally called *statistical model checking* [LDB10], which, as mentioned in the beginning, provides a pragmatic and scalable alternative to exact model checking techniques.

However, it is clear that the logic-based declarative approach of SALMA comes at the price of much higher computational costs than in other solutions for simulation and statistical model checking that take a more low-level perspective or are based on more restricted models. Of course, nowadays long simulation durations are much less problematic than they used to be since cloud computing has made provisioning large clusters with dozens or even hundreds of nodes both easy and affordable. This allows to run large numbers of simulations in parallel and thus getting meaningful results also for complex experiments in acceptable time. However, SALMA is still certainly not a suitable solution for simulating systems with thousands of agents, which is done, for instance, in large-scale traffic simulations.

Instead, SALMA's main focus is on describing heterogeneous models with complex agent behavior from a microscopic perspective. In these cases, it is typically not necessary to simulate that many agents but rather a set of representatives that exhibit all relevant types of behavior profiles. It will be demonstrated throughout the thesis that for such settings, SALMA shows its strengths by allowing the modeler to freely choose adequate levels of detail while still maintaining a comprehensible and clear structure of both the system and the requirements model.

1.2 Main Contributions of the Thesis

This thesis presents, to the author's knowledge, the first consequent attempt to use the situation calculus in discrete event simulation and statistical model checking. Additionally, the author is not aware of any other statistical model checking tool that supports first-order LTL to the same extend as SALMA does. The following chapters will show that this increase of expressiveness is very helpful in creating clear and comprehensible models, especially when the modeled systems involve more complex interactions between agents.

The SALMA framework has been implemented as part of this thesis and is publicly accessible as an open source project at `www.salmatoolkit.org`. By its architecture, which was sketched in Section 1.1, it combines the advantages of a declarative rule-based logical model with the flexibility of the general purpose language Python with its rich ecosystem of libraries. The power of this approach will become visible in several examples that are presented throughout this thesis.

For the evaluation of bounded first-order LTL formulas, a novel algorithm has been developed that uses sequences of temporal intervals to store *evaluation states* of formula instances, i.e. a history of the verdicts for partial formulas that have been found during past simulation steps and a schedule for undecided formula instances. This allows using efficient interval operations for simultaneously querying and updating multiple evaluation states at once, which can significantly improve the performance of formula evaluation for longer simulation runs. The algorithm and its data structures are described in Chapter 5.

An extension for SALMA has been created that allows modeling *information transfer* between agents or between agents and their environment, i.e. inter-agent communication and sensing. For that, a detailed generic axiomatization of information transfer processes in the situation calculus has been created which explicitly incorporates sources of uncertainty like delays and transmission errors. This is a core element of SALMA's support for artificial multi-agent systems that are embedded in the physical world – referred to lately as cyber-physical systems.

The core SALMA approach itself has been presented in [Kro14a] and [Kro14b]. The description of the modeling approach for information transfer processes in Chapter 6 has mainly been adapted from [KB16] which is the result of a collaboration with Tomáš Bureš during his stay at the PST chair.

1.3 Overview of the Thesis

After the introduction, Chapter 2 will shortly introduce the most important concepts that are used later in this thesis. In particular, it will give a brief overview of the situation calculus, temporal logics, and various forms of model checking, including statistical model checking.

Then, in Chapter 3, SALMA is presented as a self-contained approach for discrete event simulation. This includes a detailed description of syntax and semantics of SALMA's situation-calculus based domain specification language, of the agent process definition language (SALMA-APDL), and of the SALMA framework, which is used, among other things, to specify probability distributions and configure simulation experiments. How all these components fit together is shown by means of a simple fictitious example in which strategies of delivery robots are evaluated.

Chapter 4 describes how SALMA can be used for statistical model checking. The property specification language (SALMA-PSL) is described in detail and it is shown how the SALMA framework can be used to run repeated simulations with associated properties in order to estimate satisfaction probabilities or conduct hypothesis tests. SALMA's statistical model checking approach is then validated by means of an example.

In Chapter 5, SALMA's property evaluation mechanism is examined. In particular, it is explained how the evaluation module uses sequences of temporal intervals and a special data structure, the *evaluation goal schedule*, to increase the efficiency of formula evaluation. The main issue that has to be solved by these algorithms is how to keep track of (partial) formulas for which a conclusive verdict can't be made immediately due to temporal operators.

After Chapters 3 to 5 have introduced all basic components of the SALMA approach, Chapter 6 presents how they can be applied to model and simulate *cyber-physical multi-agent systems* in which the stochastic nature of sensing and communication processes have to be considered explicitly. In particular, a generic formalization of information transfer processes in the situation calculus is described together with an extension of the SALMA languages that make these concepts usable in the model. The approach is demonstrated with the help of the parking lot assignment example that was mentioned in the beginning.

The thesis is concluded in Chapter 7 with a summary and a discussion about remaining open questions and possible future research directions.

Chapter 2

Background

This chapter introduces the core concepts on which the SALMA approach, that is presented in this thesis, is based upon. In particular, these are discrete event simulation, statistical model checking, and the situation calculus. Each of these topics are research fields of their own that have produced a large body of literature, which in the case of discrete event simulation dates back more than 40 years. It is therefore clear that any attempt to present a encyclopedic overview of these fields would go way beyond the scope of this thesis. Instead, only key aspects are covered that are necessary for understanding the following chapters. Besides that, references to introductory and overview literature will be given that can be used to acquire more profound information on each of the mentioned topics.

2.1 Multi-Agent Systems

As indicated in the introduction, the term multi-agent system (MAS) is used by different communities to describe systems with sometimes completely different characteristics. In fact, there are two generally different intentions for modeling and analyzing multi-agent systems: on the one hand, multi-agent systems that exist in nature, like swarms of animals or groups of interacting people in a society, are modeled and simulated by scientists in order to understand the mechanisms of these systems. On the other hand, when *artificial multi-agent systems* are developed, models and simulations are used to anticipate the system's behavior before it is deployed. This thesis focuses on such artificial multi-agent systems, although most of the presented approaches could also be applied for models and simulations of natural systems.

Comprehensive introductions to the wide field of multi-agent systems can be found, for instance, in [Woo09] and [SLB08]. Topics that are typically addressed within the MAS community include algorithms for coordination and collaboration, communication protocols, modeling of knowledge and intention, and game-theoretic analysis and design of inter-agent interactions. As mentioned before, the concept of multi-agent systems has not only been used in science but also as a paradigm for developing distributed computer systems. Arguably the most widely used technology in this area is the JAVA Agent Development Framework (JADE) [BPR99, BCR+16], which has been actively developed since at least 1999 and has been used for many different real-world applications.

An obvious question at this point is what actually distinguishes a multi-agent system from an "ordinary" distributed system. In [Woo09, Chap. 1.3], Michael Wooldridge answers this with two main points:

1. The structure of multi-agent systems, with regard to communication and interaction, is mostly not predetermined a-priory but formed at runtime by more or less autonomous coordination processes.

2. Agents are self-interested in the sense that they do not necessarily collaborate to achieve a shared goal like the active entities of a ordinary distributed system but instead act according to their own goals (or the goals of their owner), which may well be conflicting.

While dynamic distributed system structures have become an ubiquitous phenomenon in the age of cloud computing and virtualization, the fact that agents can have different owners introduce many additional challenges for the development of multi-agent systems. The SALMA approach that is presented in this thesis does not enforce any strict requirements on agents and the level of autonomy or independence. An agent is merely understood as an *active entity*, i.e. something that can actively perform tasks and therefore differs from other passive entities that can only react to events or requests. In fact, none of the typical topics from the area of multi-agent systems are covered in depth in this thesis. However, Chapter 6 will demonstrate how aspects like communication and coordination can be addressed within the context of SALMA.

2.2 The Situation Calculus

The *situation calculus* [LPR98, Rei01] is a first order logic (FOL) language for the description of dynamic systems whose main ingredients are *actions* and *fluents*, which are variables whose values depend on the system's *situation*. Actions are encoded in the natural way by FOL functions, and fluents by functions or predicates that take a situation argument as an additional last formal parameter. A situation itself is represented as a sequence of actions, encoded as a recursive FOL term with the special function $do(Action, Situation)$. Accordingly, an action sequence $(a_1, \ldots, a_n)$ is encoded as $do(a_n, do(a_{n-1}, do(\ldots, do(a_1, S_0)\ldots))$. Here, S_0 denotes the *initial situation*, i.e. the reference state that is seen as the start of the analyzed period. In order to express the effects that an action has on the world state, the modeler specifies for each fluent exactly one *successor state axiom (SSA)*, e.g.

$$\begin{aligned} \mathit{broken}(x, do(a,s)) \equiv &(a = \mathit{crash} \wedge \mathit{holding}(x,s)) \vee \\ &(a \neq \mathit{repair}(x) \wedge \mathit{broken}(x,s)). \end{aligned} \tag{2.1}$$

The sentence above describes when an item x is *broken* in the situation $do(a, s)$, which is reached when the system is in situation s and some agent performs the action a. It says that the item will be broken after a if either a denotes a crash of the agent and the agent is currently holding the item, or a is not a repair action and x has been broken before. In this way, each SSA exactly defines the value of a fluent at the situation that results from performing any action, i.e. it captures all ways a fluent's value might change. In fact, it is shown in [Rei01] that successor state axioms like the one above provide a solution to the so called *frame problem* [Rei01, sec. 3.1.4], which basically means that they allow an efficient specification not only of what is changed by an action but also of what is not affected. What is left to specify is the so-called *initial database*, i.e. a collection of sentences that describe the state of the world at the initial situation S_0, such as

$$\mathit{holding}(\mathit{Vase}, S_0) \tag{2.2}$$

$$\neg \mathit{broken}(\mathit{Vase}, S_0) \tag{2.3}$$

Successor state axioms are complemented by *action precondition axioms* that define when the execution of an action instance is *possible*. In the spirit of the simple example from above, a typical precondition for a *pickUp* action would be that the agent is not already holding the item and the item is not too heavy:

$$\mathit{Poss}(\mathit{pickUp}(x), s) \equiv \neg \mathit{holding}(x,s) \wedge \mathit{weight}(x) \leq 50kg \tag{2.4}$$

Using just the ingredients mentioned above, it is possible to model a system in a way so that one can use well-established logical methods to reason about it. One of the most common tasks in the context of the situation calculus is solving the *projection problem*, which means checking whether a given formula $G(s)$ holds in the situation that results from performing a given action sequence starting from the initial situation S_0. The basic computational mechanism that is used there is *regression* [Rei01, Sec. 4.5]. In short, the application of the *regression operator* on a formula recursively replaces fluents by the right-hand side of their successor state axioms. This eventually yields a sentence that only mentions the *initial situation* S_0 and therefore can be treated like any FOL sentence. For instance, applying the regression operator (commonly denoted as $\mathcal{R}$) on the situation $do(drop(Rob_1), S_0)$ given the SSA from above produces the following result:

$$\begin{aligned}
&\mathcal{R}\big[broken(Vase, do(crash, S_0)))\big] \\
&\overset{(2.1)}{\equiv}\big[(a = crash \wedge holding(x, s) \\
&\quad \vee (a \neq repair(x) \wedge broken(x, s))\big]_{(x \mapsto Vase, a \mapsto crash, s \mapsto S_0)} \\
&\equiv\big[holding(Vase, S_0) \ \vee broken(Vase, S_0)) \overset{(2.2),(2.3)}{\equiv} \top
\end{aligned} \tag{2.5}$$

In spite of its elegance, this schema is not well-suited for the purpose of simulating long-running systems. The problem is that the evaluation would gain complexity for each step because with each step, a longer history has to be processed. To avoid this, the SALMA simulation engine uses an alternative mechanism that is called *progression* (see [LR97] and [Rei01, Chap. 9]). Progressing an initial database according to a performed action means to construct a new initial database that models the state of the world after that action was performed. In the general case this construction can become quite complicated and it is even shown in [Rei01] that there are models for which the initial database requires second-order logic to express progression. However, for the type of model that will be used throughout this thesis, i.e. models for discrete event simulation, the initial database consists only of sentences like (2.2) and (2.3). For these cases, progression simply amounts to performing a one-step regression and substituting with sentences from the initial database as it was done above in (2.5).

2.2.1 Quantitative Time and Clocks

Through the structure of the situation terms ($do()$), there is obviously an inherent qualitative temporal relation that describes whether a situation (i.e. state) is reached by the system before or after another situation. However, for the kind of simulation and statistical model checking supported by the SALMA approach, time has to be represented explicitly in a quantitative way. A detailed discussion of the treatment of time in the situation calculus can be found in [Rei01, chap. 7], where the *occurrence time* of each action is added as an additional real-valued temporal argument, e.g. $explode(bomb, 12.32)$. In this modeling style, time becomes an integral part of the situation term and can therefore be elegantly integrated in regression.

However, SALMA as a *discrete event simulation* (DES) approach uses a discrete time base instead of a continuous time model (see Section 2.3 below). This allows using a much simpler representation of time than the one mentioned above. In fact, time is modeled as a regular integer fluent *time* that holds the world time at the current simulation step. This topic will be examined further in Section 3.2.6.

2.2.2 Processes and Concurrency

When a more realistic description of the world is desired, it is necessary to distinguish between *instantaneous actions* and *processes* or *activities* with a certain duration. In the situation calculus this can be achieved easily by modeling *activities* by fluents that indicate their state and. The state transitions are triggered by actions, which on their part are assumed to be timeless. For instance, the following successor state axiom describes an activity *moving* that a robot starts and stops performing with the *start* and *stop* actions, respectively:

$$moving(r, do(a, s)) \equiv a = start(r) \vee (moving(r, s) \wedge a \neq stop(r))$$

Obviously, there can be multiple activities (or processes) ongoing simultaneously, so this allows expressing *concurrency* to some degree. In order to handle simultaneous actions by multiple agents, the SALMA simulation engine uses an *interleaving concurrency model* that is established by gathering executed actions from all agents in each simulation step and then compiling them into a sequence with randomized order (cf. Section 3.6). This model is usually sufficient for the kind of analyses this thesis and the SALMA approach are interested in. As Reiter explains in (see [Rei01, sec. 7.2.2]), other models of concurrency are needed mainly when a very detailed physical view is desired.

2.2.3 Stochastic and Exogenous Actions

Another step towards realism in the simulation model is to consider the inherent uncertainty that is imposed by the agents' *environment*. First of all, there might be *exogenous actions* that are not initiated by the agent but by nature or an external actor (see [Rei01, chap. 8]) and hence occur in a stochastic fashion. Additionally, actions performed by the agents can have stochastic outcomes, e.g. imposed by a failure (see [Rei01, chap.12]). Each of these additional influences can easily be integrated in the simulation control loop. When a stochastic action is executed (exogenous or intentional), the simulation engine compiles a deterministic action by stochastically selecting one of the declared possible outcomes. Similarly, the simulation engine can decide in every step to execute any exogenous actions (aka events) whose preconditions are fulfilled. For both stochastic and exogenous actions, the modeler has to specify probability distributions that are used by the simulation to choose between different outcomes of a stochastic action or to determine the next occurrence time of an event. In Chapter3 it will be shown how the specification of probability distributions is done in the SALMA approach and how the specific choices made at that stage strongly influence the characteristics of the model.

2.2.4 GoLog

The previous sections showed that the situation calculus is well suited for describing how agents interact with their environment. However, although it is possible to express action sequences in situation terms, this is not sufficient for modeling the behavior of an agent. For that, a language is needed that provides control structures for describing the agents decisions and strategies. Accordingly, in [L+97], a programming language called GOLOG is introduced that integrates with the situation calculus and allows defining *procedures* that may contain typical ingredients like loops and `IF-THEN-ELSE` blocks. GOLOG extends the typical structure of a procedural language with principles known from logic programming in order to achieve a higher level of abstraction. In particular, GOLOG adds operators that allow modeling *nondeterministic choices* for both actions and entity values. These choices are made at runtime by an automatic theorem prover (usually Prolog) that searches for a solution to constraints given in the program. A particularly concise example taken from [L+97] is shown below:

$$\begin{aligned}&\textbf{proc}\; clean\, (\forall x)\, [block(x) \supset in(x, Box)]? \mid \\ &\qquad (\pi x)[(\forall y)\, \neg on(y, x)?\; ;\; put(x, Box)]; clean\;\; \textbf{endProc}\end{aligned}$$

The procedure `clean` is meant to describe a part of the behavior of an *agent*, e.g. a robot, in a simplified scenario that is usually referred to as a *blocks world*. In this domain, which is often used as example in the context of automatic planning (see [ST01]), one or several robots have to move some *blocks* around in order to arrange them in some specified way. Here, the robot is given the simple task to move all blocks into a box, using the situation calculus action $put()$. However, the robot cannot move a block x if another one is located on top of it, which is reflected by the fluent $on(y, x)$. The recursive procedure *clean* first uses GoLog's ? operator to test whether all blocks are in the box yet, in which case the recursion is stopped. This works due to the *nondeterministic choice* operator | that can be understood to be selecting one of those control flow branches at random that leads to a successful execution. In this case, the choice is determined by a test of the fluent expression $on(y, x)$ using GoLog's ? operator. While the variable y is bound by an universal quantifier, the value for x itself is chosen nondeterministically with the π operator.

In the original description of GoLog in [L+97] and [Rei01], the language was introduced together with an implementation in Prolog that "executes" a given GoLog program by first expanding it into a logical formula and then searching for a valid situation, i.e. an action sequence that is a solution to that formula. This implies that any nondeterministic choices that are expressed in the program by means of the π or | operator mark decision points to which the search algorithm can backtrack. When it is used in this way, GoLog can be used for planning or for proving that certain states are reachable or not.

Its main advantage in comparison to purely declarative classical approaches is that it allows the modeler to control the search in a much more fine-grained way and to add explicit knowledge about the agent's expected behavior.

Several extensions to the original version of GoLog have been introduced, e.g. *ConGolog* for modeling concurrency [DGLL00] or *DTGolog* [BRS+00] for the integration of decision-theoretic planning on Markov Decision Processes. However, more importantly from the perspective of this thesis is *IndiGolog* [GLLS09], which adds support for sensing and reacting to external events at runtime. This leads to an *online* execution model in contrast to the original one, which is *offline* in the sense that it always performs the full search for a valid action sequence in an isolated phase before the acquired plan is executed. With its online execution strategy, IndiGolog would actually be a viable solution for implementing a simulation engine like the one that was implemented as part of the SALMA approach that is presented in this thesis. However, a different architecture was chosen for SALMA in which the core of the simulation engine was implemented by means of the general-purpose programming language Python that . As it will be shown in Chapter 3, this choice was made mainly due to pragmatic reasons, in particular because it allows exploiting Python's rich ecosystem of scientific libraries and frameworks. On the other hand, SALMA's simulation engine does not support GoLog's high-level constructs for nondeterminism that were described above, although they account for a main part of GoLog's strengths. This decision was made deliberately since the high level of abstraction can be problematic for the type of simulations for which SALMA is mainly intended. In fact, SALMA in contrast prefers a more fine-grained modeling perspective that makes stochastic and timing-based effects along the execution of agent programs explicit. This will probably become most obvious in Chapter 6 when SALMA's application to simulating and analyzing information transfer processes is cyber-physical systems is discussed.

2.3 Discrete Event Simulation

In its core, the SALMA approach provides a solution for defining and performing *discrete event simulations* (DES), a branch of computer-based simulation techniques that are widely used in many different application areas like manufacturing, health care, or sociology, to name only a few. The key characteristic of DES methods is that the time is regarded as a series of discrete steps and the system state (i.e. the set of variables that describe the system) is supposed to change only at certain time points [BCINN04]. This is different from simulation approaches for *continuous systems* that mostly use differential equations for defining how the state variables can evolve.

A well renowned introduction to the topic can be found in [BCINN04]. There, the authors discuss a widely used general way to characterize simulation approaches according to which one of three defined *world-views* they inhibit: the event-scheduling world view, the process-interaction world view, and the activity-scanning world-view [BCINN04, Sec. 3.1.2].

Basically, in the event-scheduling world view, the model directly describes the effect of events and rules for how new events are created and scheduled. it uses an *event schedule* containing all future events and visits only the time points in which events occur. In this approach, the occurrence time of an event is determined as soon as all necessary information are available. At that point, the duration until the event, i.e. the *time advance*, is either calculated directly for deterministic event types or sampled from a probability distribution that models the event source. A similar scheduling structure is also typically used in the process-oriented view, which differs mainly in that it sets the modeling focus on the interaction and dependencies between processes and shared resources. In contrast, simulations based on activity scanning stop at every time step to check for rules that can be applied according to the current world state in order to update the world state.

Typical examples for the use of the event-oriented and process-oriented view include *queuing systems* that are often used, among others, to model production or logistics processes, as demonstrated in [BCINN04]. For such systems that are mostly modeled with a relatively high degree of abstraction, it is obvious that the event scheduling strategy can be much more efficient than activity scanning in many cases. However, there are others in which it is either difficult to define rules for the calculation of event occurrence times in advance, or the computation is not practically feasible. In particular, it is generally hard to schedule an event in advance when its triggering condition depends on the state of time-dependent variables. In this case, the distribution of the time advance can be too complex to allow direct sampling. As a consequence of the trade-off described above, hybrid approaches have evolved that attempt to combine the advantages from multiple world-views. In particular, [BCINN04] mentions the so-called three-phase approach [Pid95], a combination of event scheduling and activity scanning.

2.4 Temporal Logics and Model Checking

Although the situation calculus technically provides means to describe arbitrary temporal relations between situations, actions, and events, more complex scenarios can become rather cumbersome to express. Instead, *temporal logics* are mostly used for characterizing finite or infinite traces of actions or states. Many different variants of temporal logics have been specified and used in the literature throughout the years. The most famous and widely used among them are without a doubt *Linear-time Temporal Logic* (LTL) [Pnu77a], introduced by A. Pnueli in 1977, and *Computation Tree Logic* (CTL) [CE82], which was defined by E.Clarke and E. A. Emerson in the 1980s. Both LTL and CTL extend classical propositional logic with *temporal operators* that allow expressing properties of *execution traces.* A typical example for a formula in LTL taken from [HR04] is

$$G(\text{requested} \implies F\,\text{acknowledged}) \tag{2.6}$$

The temporal operators F and G used in this sentence express that the partial formula in their scope is true at some point in the future (F for finally), or at all time points (G for generally). Here, the property could express an invariant which requires that all requests are eventually acknowledged. In the literature, G and F are often denoted as $\Box$ and $\Diamond$, respectively. Thus, the previous formula can also be written as follows:

$$\Box(\text{requested} \implies \Diamond\,\text{acknowledged}) \tag{2.7}$$

CTL extends the syntax of LTL with the operators E (exists) and A (all) that express existential and universal quantification over paths. With these operators, the example from above could be extended:

$$AG(\text{requested} \implies EF\,\text{acknowledged}) \tag{2.8}$$

Now the formula expresses that it holds for all traces (A) that, for all states (G), requested implies that there exists a trace (E) from this state on which eventually (finally, F) acknowledged holds. Hence, the E operator allows in this case to express a weaker requirement than the formula before.

In CTL, the use of temporal operators is restricted in that every path operator (i.e. G, F, and some others not mentioned here) have to appear together with either E or A like above. This means that not every LTL property can be expressed in CTL, i.e. neither LTL nor CTL can be regarded as more expressive than the other. For instance, the following property can be expressed in LTL but not in CTL:

$$GF\,\text{requested} \implies F\,\text{acknowledged} \tag{2.9}$$

This formula requires that if a request arrives *infinitely often* (GF), then it is eventually acknowledged (cf. [HR04, Chap 3.5]). Formulas with similar structure are typically used to express *fairness constraints*, which play an important role in specifying concurrent systems. On the other hand, the ability to use existential quantification over paths is often necessary to capture the precise meaning of intended requirements. Therefore, many approaches and verification tools support both LTL and CTL, possibly along with various extensions.

Temporal logics have been used in many contexts where logical formalisms are typically used, for instance in requirements engineering (e.g. [DvLF93]) or planning (e.g. [BK00]). However, they are probably most well known for their role in automatic *model checking* approaches that allow verifying whether a system model, specified in some formal graphical or textual modeling language, conforms to a temporal logics formula. The model checker basically explores the state space until it can either conclude that all possible traces the system can take fulfill the formula or it finds a *counter-example*, i.e. a trace in which the formula is violated. This approach is particularly useful for concurrent systems that are inherently hard to analyze by testing or manual inspection. After several decades of research in the field, there are now many well-engineered and mature tools, e.g. SPIN [Hol97] or NuSMV [CCG02] that have been applied successfully on various use cases from academia and industry. In the case of SPIN, one feature that makes the use of this tool particularly appealing for practitioners is its ability to extract models from C source code [Hol00], which allows working with models at a low level of abstraction even for larger programs.

While classical solutions like SPIN have been applied successfully in many different settings - ranging from the verification of control algorithms for a flood control system [Kar96] to the analysis of algorithms used in spacecraft (e.g. [HLP01]) - there are scenarios that demand extensions to the classical model checking approaches. For instance, *real-time systems*, i.e. systems in which timing effects have to be considered, can be modeled by means of *timed automata*[AD94], i.e. automata whose states and edges can be annotated with *clock constraints* that specify time windows for the occurrence of state transitions. One particularly successful tool that is based upon the timed automata formalism is UPPAAL [LPY95], which provides an integrated graphical user interface that can be used for modeling, simulating, and model checking real-time systems. UPPAAL has been used in a large number of studies and projects with both academic and commercial background, of which some are referenced on the UPPAAL website (`www.uppaal.org`). Since the classical timed automata formalism as presented in [AD94] is not expressive enough to model complex scenarios in a concise way, UPPAAL adds several extensions. First, an UPPAAL model can be composed out of multiple automata that are synchronized via send and receive primitives over *channels*. Additionally, a textual language with a C-like syntax and data structures like arrays can be used to model more complex transition effects.

A core aspect of all mentioned model checking approaches in general is that they allow models to comprise nondeterministic aspects. For instance, a state in the model can have multiple *outgoing transitions*, i.e. there are multiple possibilities which state is reached next. Similarly, a constraint in a timed automaton model in UPPAAL usually does not specify an exact delay but an interval. In both cases, the model checking tool will effectively consider all possibilities in order to answer queries of the kind "Is it possible that something happens?" or "Is it guaranteed that something will eventually happen?", with optional restrictions to time limits in the case of UPPAAL. For many scenarios, however, it is important not only to know whether something may or must happen but also with which probability. Such questions can be answered through *stochastic model checking.* A good introduction to this field can be found in [KN07] where the authors explain both theoretical foundations of the basic principles and algorithms but also talk about the practical use of their tool PRISM [HKNP06], which is at the writing of this thesis one of the most established stochastic model checking tools. The first basic step in stochastic model checking is to augment the system model with probability distributions for state transitions. Then, an algorithm is needed that allows calculating the probability that a given *path* through the state space is taken by the system. Usually, stochastic model checking approaches work on models for which exact efficient algorithms are known, such as discrete time Markov chains (DTMCs), continuous time Markov chains (CTMCs) or Markov decision processes (MDPs). With that, it is possible to construct model-checking algorithms along the lines of the basic principles used for the classical deterministic models (see [KN07]). In fact, PRISM uses a variant of CTL, called probabilistic CTL (PCTL). PCTL replaces the path quantifiers E and A of CTL with *probabilistic operators* that allow stating upper or lower bounds for the probability with which a sub-path formula is fulfilled for traces beginning in the current state. Thus, the CTL example in (2.8), can be transformed to the following PCTL formula that expresses a relaxed version of the requirement:

$$P_{\geq 0.9}[\Box(\text{requested} \implies P_{\geq 0.75}[\Diamond\, \text{acknowledged}])] \tag{2.10}$$

Using an intuitive interpretation, the formula in(2.10) requires that in at least 90% of all executions, each request has a chance of 75% to be acknowledged eventually.

Although huge achievements have been made in improving the efficiency of model checking tools during the last decades, they still inherently suffer from the so-called *state explosion problem*, which describes the fact that the number of states in a system model increases exponentially in multiple factors, including the numbers of variables, processes, or nodes in a distributed system (cf. [BK08, Chap. 2.3]). Several approaches for mitigating this problem have been suggested, such as abstraction techniques like the one proposed by Clarke in [CGL94], partial order reduction approaches (see [GvLH$^+$96]), symbolic model checking using binary decision diagrams (BDDs) [BCM92], bounded model checking using SAT solvers [CBRZ01], or distributed model checking like the swarm verification extension of the SPIN model checker [HJG08]. All of these methods have in common that they are committed to solve the model checking problem in an *exact* manner, which means that when the algorithm terminates without finding a counter-example, this is interpreted as *proof* that the checked property is fulfilled by the model. On the other hand, there also exist *approximative* approaches that deliberately give up this claim of preciseness. Since this idea is fundamental to the SALMA approach presented in this thesis, it is discussed separately in the next section.

2.5 Statistical Model Checking

In spite of the endeavors mentioned above to reduce the size of the state space, there are domains where the state space is still too large to make exact model checking practicable. One possible solution in these cases is to use a statistical approximation that deliberately risks making errors within certain probabilistic bounds. For instance, in [GS05], the authors propose an approach called *Monte Carlo Model Checking* that is based on the original automata-based solution for LTL model checking that was introduced in [VW86]. However, instead of constructing an automaton for the complete system as in [VW86], the algorithm performs *random walks* through a state space that is constructed *on-the-fly*. During this traversal, accepting traces are detected and they form a Bernoulli sample that is finally used to assert the checked property with a configurable bound for the error probability for missing a property violation.

While the approach described in [GS05] focuses on mitigating the state-space explosion problem for classical reactive system models and LTL, the use of Monte Carlo methods is much more appealing for *stochastic model checking.* On the one hand, this is because the state space explosion problem is particularly serious for stochastic model checking since existing exact algorithms are computationally expensive. Additionally, all these exact algorithms are limited to certain types of stochastic models, i.e. there are more general models of stochastic systems like generalized semi-Markov processes (GSMPs) [Gly89] for which no exact solutions exist (cf. [YS02, p.1]). This lead to significant research interest for applying Monte Carlo methods to the verification of many different types of systems and the term *statistical model checking (SMC)* was established. An overview of different approaches and research challenges in this field can be found in [LDB10].

The basic general idea of statistical model checking is to conduct discrete event simulations and evaluate the *simulation traces*, i.e. the observed sequences of states and actions or events, against properties that are typically formulated by means of some temporal logic. The *statistical model checker* decides for each inspected simulation run, whether or not a given property ϕ holds. If the simulated model contains factors of uncertainty, like actions with uncertain outcome or stochastic events, then a property ϕ will be *fulfilled* in a given run σ (i.e. $\sigma \models \phi$) with some probability p. Consequently, when N simulation runs σ_i are performed, then the number of successful runs follows a *binomial distribution* $B(N, p)$.

Based on this assumption, it is possible to use common statistical methods to either estimate the probability p, which is sometimes referred to as the *quantitative statistical model checking* problem, or to perform hypothesis tests like $H_0 : p \geq p_{min}, H_1 : p < p_{min}$, aka solving the *qualitative statistical model checking problem.*

Several approaches to statistical model checking have been proposed, which differ with respect to the property specification languages they use, their evaluation algorithms, and the statistical methods they apply. In particular, the tools Ymer [You05b] and VESTA [SVA05b] have been recognized widely in the community and inspired further research. Additionally, both PRISM and UPPAAL, which were mentioned in the last section, have meanwhile been extended with the ability to apply statistical model checking. For PRISM, this adds support for much more complex models than those that can be processed by PRISM's exact numerical algorithms.

Several important aspects of statistical model checking will be mentioned throughout the thesis and shortly discussed in the context of the relevant component of the SALMA approach. However, before getting started, it makes sense to consider one particular topic that immediately suggests itself: deciding how many simulations are "enough"? This is the topic of the next section.

2.5.1 Sample Sizes and Sequential Hypothesis Tests

One of the most important questions that arise when any kind of statistical experiments are performed is how many samples have to be taken in order to achieve a desired level of accuracy. For statistical model checking especially, choosing a minimal sample size can be crucial since producing a sample, i.e. performing a simulation run, can be very expensive for more complex models. Since this issue is found in all domains where statistical inferences are made, many solutions for finding *optimal sampling plans* have been developed. In his PhD thesis [You05a], Younes gives a short overview of these methods, some of which date back to the 1940s. He also introduces an algorithm that is able to find an optimal pair (c, n) of a predefined constant c and a fixed sample length n for a hypothesis test that accepts an hypothesis if more than c of n simulation runs are successful (see [You05a, ch. 2.2.2]).

Another general approach that is well suited for qualitative statistical model checking, i.e. for testing hypotheses about the probability of a model fulfilling a property, are so-called *sequential hypothesis tests*. These tests do not require selecting a fixed sample size beforehand but instead are able to determine "on the fly" during a series of samples when the gathered data is sufficient to accept or reject an hypothesis with the demanded error bounds. One of the earliest, but nevertheless still widely used approaches for sequential hypothesis testing is the *Sequential Probability Ration Test (SPRT)* that was introduced by A. Wald in 1945 [W$^+$45].

Since the original description of the SPRT refers to the probability of a *defect*, i.e. a property violation in SMC, rather a *success*, this perspective is adopted here. This means that for qualitative SMC problems, the null hypothesis $H_0 : p \leq p_max$ is tested against $H_1 : p > p_max$. The first main step for using the SPRT is to define an *indifference region* given by two probabilities p_0 and p_1 around p_{max}. During the test, the actual value of p is estimated by the ratio of defects, and a test decision will only be considered an error if (a) H_0 is rejected and $p \leq p_0$ (type I error), or (b) H_0 is accepted and $p \geq p_1$

(type II error). Additionally, two parameters α and β have to be defined for the maximum probability of type I and type II errors, respectively. Then, after each simulation run, the following *probability ratio* is calculated:

$$\frac{p_{1_m}}{p_{0_m}} = \frac{p_1^{d_m}(1-p_1)^{(m-d_m)}}{p_0^{d_m}(1-p_0)^{(m-d_m)}} \tag{2.11}$$

Here, m is the total number of simulation runs so far, and d_m is the number of simulation runs with a property violation (defects). The nominator (p_{1_m}) and denominator (p_{0_m}) actually denote the posterior probabilities for the current number of defects, given that $p = p_1$ or $p = p_0$. It is shown in [W$^+$45] that the error bounds α and β are respected if the following test procedure is used:

(A) Reject H_0 if $\frac{p_{1_m}}{p_{0_m}} \geq \frac{1-\beta}{\alpha}$.

(B) Accept H_0 if $\frac{p_{1_m}}{p_{0_m}} \leq \frac{\beta}{1-\alpha}$.

(C) Run an additional simulation if $\frac{\beta}{1-\alpha} < \frac{p_{1_m}}{p_{0_m}} < \frac{1-\beta}{\alpha}$.

It is obvious that this procedure can easily be integrated in any simulation approach in which it is possible to iteratively run independent trials. In fact, the SALMA simulation engine integrates a simple implementation of the basic SPRT (see Chapter 4), although this could easily be replaced by any other function that is able to detect when a given stopping criterion is met.

When the original SPRT procedure is used, the parameters α, β, p_0, and p_1 have to be configured according to problem-specific considerations. The tool can then automatically perform the necessary number of simulations until it eventually finds a conclusive result within the specified statistical error bounds. As shown [W$^+$45], the expected number of required simulations N_{req} depends on the values chosen for the parameters p_0, p_1, α, and β, but also on the actual probability p. If p is very close to the center of the indifference region $[p_0, p_1]$, then it is expected that the highest number of simulation runs will be required to find a significant result. In these cases, the test could run for a long time without a decision. This phenomenon is also demonstrated during the example that is described in Section 4.3 of this thesis. The SALMA simulation engine handles this kind of "livelock" situation simply by using a predefined time limit after which the test is stopped and the user may change the parameters. Additionally, a heuristic is described in [W$^+$45] that could be used to stop the sequential test at a certain point.

Chapter 3

Multi-Agent Simulation with SALMA

This chapter discusses SALMA as a full-fledged approach for discrete event simulation of multi-agent systems. It starts with an introduction of the two languages that are used in SALMA to define the simulated system. The first one is SALMA's Domain Description Language (SALMA-DDL) that is integrated in Prolog and provides means for defining the elements that describe the system in the situation calculus. Second, there is SALMA's Agent Process Definition Language (SALMA-APDL), an *internal domain specific language (DSL)* [Fow10] which is embedded in Python and allows defining behavior of agents in a similar procedural style as in typical agent programming languages like GoLog (see Section 2.2.4). How these languages and their underlying concepts are used is demonstrated by means of a simple fictitious example from the realm of multi-robot systems.

Once the domain has been axiomatized and the agent processes have been defined, the SALMA simulation framework can be used to set up and perform a *simulation experiment.* Section 3.4 covers the essential design of this framework and how it is used in all stages of the simulation experiment. Then, after the first part of this chapter has introduced SALMA's discrete event simulation approach from a more practical perspective, Section 3.6 presents an operational semantics for the simulation algorithm in order to allow a precise understanding of the involved concepts and mechanisms.

Remark: *The basic concepts of the modeling languages that are presented in this chapter have been introduced before in [Kro14a] and [Kro14b]. Additionally, parts of the operational semantics have been presented already in [KB16] as original work by the author of this thesis. However, all mentioned content has been extended and improved significantly for this chapter.*

3.1 A Simple Simulation Example: Delivery Robots

The example that will be used throughout this chapter to introduce SALMA's simulation capabilities is a fictional scenario from the realm of assembly automation in which multiple simple *robots* deliver *items* to *workstations*. As a simplification, the physical world is modeled as a discrete two-dimensional grid where stations have fixed locations and in each step a robot can move only to one of its adjacent grid cells in the four main directions up, right, down, and left. In order to pick up an item, a robot has to move to grid cell where the item is located and *grab* it, using some grip mechanism that is not further specified. After it has picked up an item, a robot moves to a workstation and *delivers* the item as soon it has arrived at the station's grid cell.

The decision to deliver an item to a workstation is not made by the robot itself but by a *coordinator agent*, which is a computer system that is able to communicate with both workstations and robots (via wireless transmission). When the coordinator receives a request from a workstation, it is first stored in a *request queue* that is processed periodically. For each request, the coordinator selects a robot that has no current delivery assignment and an item that has not yet been scheduled for delivery. This assignment is then sent to the robot as a command message after which the robot immediately starts moving towards the item's location. To further simplify the model, it is assumed that the robot knows all relevant positions, i.e. the grid coordinates of all items, of all workstations, and of itself. Additionally, the example ignores the fact that the journey of a robot could be blocked by an obstacle and supposes that robots can move freely through the grid cells of workstations and items. However, a robot can *collide* with another robot when both are in the same grid cell, in which case both robots could *break*, i.e. stop working altogether, depending on the intensity of the collision.

Figure 3.1 shows an overview of the described scenario. A common motivation for conducting a simulation study in a case like this is to compare the influence of different factors for the efficiency of the system. For instance, this could be measured by the average rate of item deliveries or by the average time it takes until a request from a workstation is fulfilled. The following sections will use the delivery robots example to demonstrate the introduced concepts and elements of the domain axiomatization and of the agent behavior definition in SALMA. Section 3.5 puts all pieces from the previous sections together and describes how the simulation is performed using the SALMA framework. Additionally, the results of some concrete simulation experiments will be shown and some options for analyzing them are discussed. In particular, the visual inspection of the results will act as a means for validating the correctness of both the model and SALMA's simulation engine. The examples presented in this chapter contain important parts of the delivery robots model but omit others that are either more repetitive or too technical to fit into the structure of this chapter. However, the example's full source code can be found at the

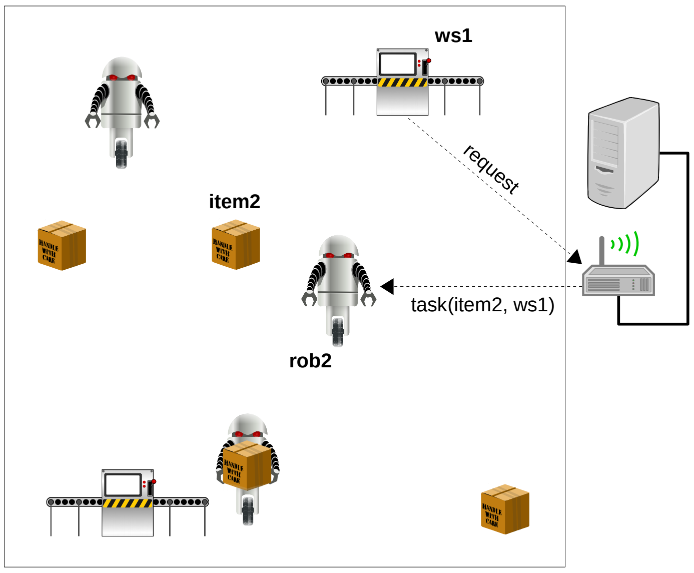


Figure 3.1: Overview of the delivery robot example.

SALMA website (`www.salmatoolkit.org`).

3.2 Axiomatization of System Domains

The first step in modeling a system in SALMA is to describe its *domain*, i.e. the kinds of entities and agents that inhibit the system and the mechanism and constraints that govern the possible interactions between them and with their environment. As mentioned before, this is done by means of the situation calculus that was shortly introduced in Section 2.2. There it was explained that a situation calculus model consists of a set of axioms that define exactly how actions and events influence the value of *fluents*. Although these axioms can easily be defined in plain Prolog [Rei01, chap. 5], the SALMA-DDL extends the classical situation calculus with additional concepts that are used to provide necessary information for the simulation and the property evaluation algorithms. Most importantly, there are the following additions:

1. Constructs that are used to declare the signature of of fluents, actions, and events in the system. They also add further information that is used by the simulation algorithm, for instance to determine whether an action is deterministic or stochastic. These constructs will appear throughout the following sections and will be explained together with the main concepts they refer to.

2. A *sort system* that is used to categorize entities and agents of the system. The SALMA situation calculus variant requires all parameters of fluents

to be typed with finite sorts. This is necessary for simulation because it makes the set of *fluent instances* finite and therefore allows the simulation system to calculate the updates for all values through *progression* (see Section 2.2). The sort system also supports inheritance relations that help creating a more object-oriented view on the model. Section 3.2.1 provides more details.

3. Additional axioms are added that define not only when an event is *possible*, like the *Poss*-axiom of the classical situation calculus, but also when it can be *scheduled* by the simulation algorithm to occur at a later time. These axioms are first introduced in Section 3.2.5 and scheduling in general will play an important role in this chapter.

4. A macro mechanism for defining successor state axioms (SSAs) by means of so-called *effect axioms* that are unfolded automatically by the simulation engine. This basically follows the idea behind the systematic construction of SSAs presented in [Rei01]. On the one hand, this enhances the readability of domain model since effect axioms are often much more concise than fully specified successor state axioms. On the other hand, the structure of effect axioms allows the simulation algorithm to identify precisely if a fluent instance is affected by a given action. Most of all, this is used in the event scheduling mechanism of the SALMA simulation engine to determine whether the condition for the occurrence of an event is *time-dependent* or not (see Definition 3.28 in Section 3.6.2).

In summary, a SALMA domain model description is a Prolog source file that contains a collection of declaration statements and axioms for sorts, fluents, actions, and events. The following sections introduce all elements of the SALMA domain description language and demonstrate their use in the context of the running example that is introduced along the way. The syntax of these elements will be presented with a notation based on a standard for the Extended Backus Naur Form (EBNF) that was proposed as ISO standard ISO/IEC 14977 by Scowen [Sco98]. In particular, this notation uses brackets (`[...]`) to denote an optional appearance of the included content and curly braces (`{...}`) to express that the included content can occur once, repeated, or not at all. However, two deviations from this notation will be used in the following: a) the use of an ellipsis (`...`) to omit terminal symbols that are clear from an intuitive understanding ; and b) the omission of a comma to separate symbols.

With these convention, the syntactic structure of a SALMA domain model description is specified by the following grammar:

Definition 3.1 (Domain Model Structure).

```
Domain = { SortDecl | SubsortDecl | FluentDecl | DerivedFluentDecl |
           ConstantDecl | ActionDecl |  EventDecl |
           EffectAxiom | DerivedFluentAxiom | PossAxiom | SchedAxiom }
```

Before any concrete syntax can be defined, it is necessary to introduce some general syntactical rules and restrictions for the Prolog syntax that are used throughout this section. First, a distinction between character classes and different term types is needed at various points.

Definition 3.2 (Basic Syntax Elements and Term Classes)**.**

```
LowerCaseLetter = "a" | ... | "z" ;
UpperCaseLetter = "A" | ... | "Z" ;
Digit = "0" | ... | "9" ;
Alphanumeric = LowerCaseLetter | UpperCaseLetter | Digit ;
Number = Digit {Digit} ["." Digit {Digit}] ;
Identifier = LowerCaseLetter { Character | "_" } ;
Variable = (UpperCaseLetter | "_" ) Alphanumeric | "_" ;
Term = Variable | Identifier | List | Identifier "(" Term {"," Term } ")" ;
List = "[" "]" | "[" Term {"," Term } "]" ;
```

A particularly important restriction is that all names for sorts, fluents, actions, etc. have to be *identifiers*, i.e. strings that start with a lowercase letter and after that may contain alphanumeric characters (i.e. letters, and numbers) and underscores. This syntax constraint is necessary to distinguish names from Prolog variables. With these general considerations, the elements of a domain description can now be introduced step by step, beginning with the sort system.

3.2.1 The SALMA Sort System

Different from Reiter's original version of the situation calculus, SALMA employs a *multi-sorted* version. This means that all entities and agents in a SALMA model belong to a specific *sort* that is either predefined by the simulation system or defined in the model. The *domain* of a sort is the set of existing entities of that sort. It is also possible to define *sub-sort relations* that declare that the domain of one sort is included in the other.

Definition 3.3 (Sort Declarations)**.** The syntax of sort declarations in a SALMA domain model is defined by the following grammar:

```
SortDecl = "sort" "(" SortName ")" "." | "sorts" "(" SortList ")" "." ;

SortList = "[" SortName  "," SortName  "]" ;
```

```
SubsortDecl = "subsort" "(" SortName "," SortName ")" "."
    | "subsorts" "(" SortList "," SortName ")" "." ;

SortName = Identifier ;
```

The statement **sort**(X) declares a sort with the name X. On the other hand, a **subsort**(X, Y) declares that the sort X is a subsort of the sort Y, i.e. that all entities that belong to the sort X also belong to Y. Finally, **sorts** and **subsorts** are shortcuts that allow declaring several sorts or subsorts at once.

Besides the sorts that are declared explicitly within the model, SALMA supports several predefined *primitive types*.

Definition 3.4 (Primitive Types). The following predefined primitive types exist in SALMA:

integer:	Positive and negative integer numbers.
float:	Floating point numbers.
boolean:	Boolean values that are represented by the literals **true** and **false**.
list:	Lists with elements of arbitrary type which can be accessed as regular Python lists in agent control processes (see Section 3.3) or as regular Prolog lists in axioms (see below) or SALMA-PSL properties (see Chapter 4).
term:	Arbitrary terms that are not interpreted further by the SALMA simulation and evaluation mechanisms but can be processed within the control procedures of agent processes (see Section 3.3).

Primitive types are special sorts that cannot be part of an inheritance hierarchy. Their use is actually restricted further, which is discussed at several points in the following subsections. To distinguish these primitive types from sorts that are declared by **sort** statements, the latter are called *entity sorts*, implying that their domains are finite sets of *entities*.

As the top-level element of the inheritance hierarchy for entity sorts, SALMA defines the sort `object` that is meant to include every other entity sort in the system. Additionally, the sort `agent` is defined as a generalized type for agents that may be used either directly as the type of an agent or as an ancestor for a concrete agent subsort. SALMA's sort system also allows *multiple inheritance*, i.e. a sort could be included in more than one more general sorts.

To start the definition of the running example, a few sorts are declared in Figure 3.2. The sorts `robot` and `coordinator` are declared as subsorts of `agent`, which implies that entities of these sorts will later be equipped with agent processes. On the other hand, both `robot` and `item` are declared

```
sorts([robot, item, coordinator, workstation, movable_object]).
subsorts([robot, coordinator], agent).
subsorts([robot, item], movable_object).
```

Figure 3.2: Sort declarations for the multi-robot example.

as *movable objects*, implying they have variable positions, while entities of type `workstation` are meant to remain at fixed locations. With this first categorization in mind, the properties of the sorts are then further described by associating fluents with them, which is explained next.

3.2.2 Fluents and Constants

Fluents are clearly the core asset of the classical situation calculus and by this also of the SALMA domain model. However, as mentioned in the beginning of this chapter, SALMA requires a more detailed declaration than in the original situation calculus. In particular, both the fluent value itself and the parameters have to be *typed* to make it usable by the simulation algorithm. Additionally, a distinction is made between *regular fluents* that are updated in the progression step, *derived fluents* whose values are calculated based on other fluents, and *constants* that are initialized once and keep their value throughout the simulation. According to these demands, the SALMA domain description language provides three declaration statements.

Definition 3.5 (Fluent and Constant Declarations)**.** The following grammar rules describe the syntax of fluent and constant declarations in the SALMA-DDL:

```
FluentDecl = "fluent" "(" FluentName "," ParamList "," FluentType ")" "." ;

DerivedFluentDecl =
    "derived_fluent" "(" FluentName "," ParamList "," FluentType ")" "." ;

ConstantDecl = "constant" "(" FluentName "," ParamList "," FluentType ")" "." ;

FluentName = Identifier ;

FluentType = Identifier ;

ParamList = "[" "]" | "[" ParamSpec "," ParamSpec "]" ;

ParamSpec = FluentName ":" FluentType ;
```

With **fluent**$(F, [x_1 : t_1, \ldots, x_n : t_n], t_F)$, a fluent F is declared that has the parameters x_1 to x_n whose types are given by t_1 to t_n. The type of the fluent itself is t_f. While there is no restriction on the fluent's type t_f, the parameter types have to be *entity sorts* (see below). The statement **derived_fluent** declares a *derived fluent* that can also have parameters with primitive types like **integer** or **float**. Furthermore, **constant** sets up a *constant* that can be used like a fluent but whose instances are not affected by actions or events.

In the terminology that is used in this thesis, a distinction is made between a fluent and its instances, i.e. the current value that is associated to a specific selection of parameter values. Since the progression mechanism of SALMA's simulation engine relies on the fact that the number of fluent instances is *finite*, this also implies that all parameter types have to be finite. The same applies to instances of constants, in this case not due to constraints of the progression operation but because the simulation engine needs to be able to assure that all constant instances are initialized at the start of the simulation. For *derived fluents*, the restriction to finite parameter sorts is not necessary because a derived fluent is neither initialized nor updated in progression but is instead a *situation dependent function.*

Figure 3.3 demonstrates the declaration of several regular fluents, derived fluents, and constants for the running delivery robots example. At first, the fluent `broken` acts as an essential status flag that keeps track of whether or not a robot is damaged too much to be operational, which might happen in a collision, as will be shown later. After that, the fluents `xpos` and `ypos` set up a 2-dimensional discrete grid for representing the positions of movable objects, i.e. items and robots. Only robots can move actively so the velocity fluents `vx` and `vy` are only associated to agents of this type. Additionally, the derived fluents `moving and ready` will be used to check whether a robot is moving and when it is able to perform the next step, respectively. When an item is carried by a robot, this is expressed by the relational fluent `carrying` being true for this particular robot-item pair. As soon as an item is delivered to a workstation, this fact is recorded in the fluent `delivered_to`, and until that, the item remains in the state `undelivered`, which is marked by a corresponding boolean fluent. Furthermore, the integer fluent `delivered_item_count` keeps track of the number of items that are delivered to a specific workstation during the simulation, which will be used as a performance measure in the analysis of the results. The *allocated task* of a robot, i.e. the item that a robot should pick up next and the workstation it should be delivered to, is stored in the fluent `next_task`, which will be set by the coordinating station agent. This fluent actually stores a *term* of the form `d(item, workstation)` whose components are made accessible by the two derived fluents `task_item` and `task_workstation`. Besides that, `unassigned` is set up as a convenient status flag which indicates that no task is assigned. As described above, the *coordinator* agent receives request messages from the workstations and stores them into a queue that is represented by the fluent `request_queue` that stores a list.

```
fluent(broken, [r:robot], boolean).

fluent(xpos, [o:movable_object], integer).
fluent(ypos, [o:movable_object], integer).

fluent(vx, [r:robot], integer).
fluent(vy, [r:robot], integer).

derived_fluent(moving, [r:robot], boolean).
derived_fluent(ready, [r:robot], boolean).

fluent(carrying, [r:robot, i:item], boolean).
fluent(delivered_to, [i:item], workstation).
derived_fluent(undelivered, [i:item], boolean).
fluent(delivered_item_count, [ws:workstation], integer).

fluent(next_task, [r:robot], term).
derived_fluent(task_item, [r:robot], item).
derived_fluent(task_workstation, [r:robot], workstation).
derived_fluent(unassigned, [r:robot], boolean).

fluent(request_queue, [c:coordinator], list).

constant(stationX, [ws:workstation], integer).
constant(stationY, [ws:workstation], integer).
```

Figure 3.3: Fluent and constant declarations for the multi-robot example.

The position of the coordinator does not matter for the simulation and hence it is not reflected in the model. In contrast, the positions of the workstations are obviously essential for the delivery process. However, a workstation cannot move and therefore its position is represented by two constants, `stationX` and `stationY`.

The fluents and derived fluents defined in Figure 3.3 are not fully specified until their corresponding axioms have been defined. This topic will be discussed below in Section 3.2.4. Before that, however, the actions and events that are responsible for changes to fluent instance values have to be declared.

3.2.3 Actions and Events

Like Reiter [Rei01], SALMA distinguishes three basic types of actions, namely 1) deterministic, so called *primitive actions* ; 2) stochastic actions, i.e. actions with a random outcome ; and 3) exogenous actions aka events. As with fluents, SALMA requires actions and events to be declared explicitly.

Definition 3.6 (Action and Event Declarations)**.** The following grammar rules describe the syntax of action declarations in the SALMA-DDL:

> *ActionDecl* = "primitive_action" "(" *ActionName* "," *ParamList* ")" "." |
> "stochastic_action" "(" *ActionName* "," *ParamList* "," *OutcomeList* ")" "." ;
>
> *EventDecl* = *ExogenousAtionDecl* | *ExogenousActionChoiceDecl* ;
>
> *ExogenousAtionDecl* = "exogenous_action" "(" *ActionName* ","
> *ParamList* "," *ParamList* ")" "." ;
>
> *ExogenousActionChoiceDecl* = "exogenous_action_choice" "(" *ActionName* ","
> *ParamList* "," *OutcomeList* ")" "." ;
>
> *ActionName* = *Identifier* ;
>
> *OutcomeList* = "[" *ActionName* "," *ActionName* "]" ;

Here, **primitive_action**$(A, [x_1 : t_1, \ldots, x_n : t_n], t_A)$ declares a deterministic action that can be performed intentionally by an agent. All parameters $x_1, \ldots, x_n$ are set by the acting agent to concretize the intended effect. The parameter types are not restricted in any way. i.e. both entity sorts and any primitive type can be used. The second action declaration statement, **stochastic_action** declares an action together with a list of possible *outcomes*, i.e. a list of action names that have to refer to primitive actions declared in the same model. The choice between these outcomes is made by the simulation according to some probability distribution that is defined by the modeler (see below).

In contrast to primitive and stochastic actions, the statement **exogenous_action**$(E, [x_1 : t_1, \ldots, x_m : t_m], [x_{m+1} : t_{m+1}, \ldots, x_n : t_n])$ declares E as a type of exogenous action, i.e. an *event* that *originates from the environment* instead of an agent. Concrete *event instances* are formed by combinations of values for the parameters that are declared inside the first parameter list $(x_1, \ldots, x_m)$. Just like in the case of fluent instances, the types of these 'identifying' parameters have to be finite entity sorts. The parameters declared in the second list, $x_{m+1}, \ldots, x_n$, convey information that augment the event instance and are not subject to any type restrictions. In some cases, several events are actually *mutually exclusive*, i.e. only one of them can happen in a given situation. To model this fact, an **exogenous_action_choice** declaration can be used that groups a list of exogenous actions with the same identifying parameter. The simulation will later pick only one of the specified events when the common precondition of the *exogenous_action_choice* is fulfilled (see Section 3.2.5).

Although the use of actions in agent processes and domain axioms is discussed in detail within later sections, it is useful right away to clarify how the concrete *action terms* are formed that determine the state changes within the system. For primitive actions, the answer is simple: the action term is exactly what is issued by the agent in the `Act` statement within the agent procedure (see Section 3.3.2).

For stochastic actions, the situation is different. Although the agent performs the action that is declared in the **stochastic_action** statement, the resulting action term depends on the outcome that is chosen probabilistically by the simulation engine. Each of the possible outcomes that are mentioned in the declaration of the stochastic action has to be declared as a primitive action itself. Since both the stochastic action and the primitive action outcomes could be declared with parameters, the modeler has to specify a *mapping* from the parameters of the stochastic action, which the agent actually sets when it performs the action, to parameters of the outcome action. This is done with the SALMA experiment configuration API, which is described in Section 3.4.2. Another use of the same API is the configuration of probability distributions for exogenous actions (events). Without going into details now, the simulation selects the *event instances* that should occur at a specific point in time according to a set of axioms and probability distributions. To make this possible, each event instance has to be identified by the values for the parameters of the first parameter list in the declaration. Once an event instance has been chosen to occur, the engine chooses values for the parameters of the second list according to additional distributions that are allowed to produce values of any type. For example, this mechanism is used for *information transfer processes* in SALMA to introduce random errors (see Chapter 6). It is important to note, however, that in the final *action term* that represents the event within the situation calculus, the parameters from both lists appear in one flat sequence just like for regular actions.

For the robotics example, the model is extended with the actions shown in Figure 3.4. The four move actions will be used within a robot control process to initiate a movement to the adjacent grid cell in one of the four main directions. A movement step does not happen instantaneously but is instead an *activity* that is started by a move action and ended by either a `step_succeeded` or a `step_failed` event. Both of these concrete outcomes are declared as exogenous actions with a single parameter that identifies the robot. However, since the movement step can either fail or succeed but not both, they are declared as mutually exclusive by means of an `exogenous_action_choice`.

Another related exogenous action is `collision`, which can happen between two robots. The integer parameter `severity`, which is declared in the second parameter list, will convey a random value for each event instance that will later be used to decide whether the robots are damaged during the collision. The stochastic action `pickUp` represents the attempt of a robot to pick up an item. The model acknowledges that this might fail and therefore specifies both a positive outcome, `grab` and a negative one, `drop`, which means that the item slips and the robot cannot get a grip on it. In fact, the `grab` outcome also has

```
primitive_action(move_right,[r:robot]).
primitive_action(move_left, [r:robot]).
primitive_action(move_down, [r:robot]).
primitive_action(move_up, [r:robot]).

exogenous_action(step_succeeded, [r:robot], []).
exogenous_action(step_failed, [r:robot], []).
exogenous_action_choice(step_finished, [r:robot],
                              [step_succeeded, step_failed]).

exogenous_action(collision, [r1:robot, r2:robot], [severity:integer]).

stochastic_action(pickUp, [r:robot, i:item], [grab, drop]).
primitive_action(grab, [r:robot, i:item, grip:integer]).
primitive_action(drop, [r:robot, i:item]).
exogenous_action(accidental_drop, [r:robot, i:item], []).

exogenous_action(request, [ws:workstation, c:coordinator], []).
primitive_action(assign_task, [c:coordinator, r:robot, i:item, ws:workstation]).
```

Figure 3.4: Action and event declarations for the multi-robot example.

a parameter `grip` that could reflect the grip quality. For parameters like that, which are not directly mappable to arguments of the action execution like `r` and `i`, the modeler has to specify a probability distribution from which a value is chosen when the action is performed (see Section 3.4.2). Besides the item being dropped during a pick-up attempt, it could also happen that the robot drops an item during its journey to the workstation, which is modeled by the exogenous action `accidental_drop`. It is clear that the selection of the probability distributions that determine the grip quality and whether an accidental drop occurs obviously affects the outcome of the simulation significantly. In fact, Section 4.3 will come back to this topic and demonstrate how a physical aspect like grip quality can be reflected by a distribution for `accidental_drop` that is conditioned on the grip quality. The last two actions in Figure 3.4 are used to handle the coordination of tasks. Here, the request of a workstation is modeled as an *exogenous action* rather than a primitive, aka intentional, action. This implies what could also be seen in the sort declaration, namely that workstations are not modeled as active agents in the example, but as passive entities without own behavior. On the other hand, the assignment of a task is clearly intentional and therefore added as a primitive action that is performed by the coordination agent.

At this point, all assets of the system, i.e. sorts, fluents, actions, and events, have been declared. The next step towards a full domain specification is to add axioms that describe how the values of fluents come about.

3.2.4 Effect Axioms and Derived Fluent Functions

As mentioned in the beginning of Section 3.2, *effect axioms* are actually macros used in the SALMA-DDL to specify how the value of a fluent instance is affected by an action or event instance. In fact, effect axioms are translated automatically to *successor state axioms* as defined in Section 2.2. Their syntax is specified in Definition 3.7.

Definition 3.7 (Syntax of Effect Axioms)**.** The following grammar rules describe the syntax of effect axioms in the SALMA-DDL:

```
EffectAxiom = "effect" "(" FluentTemplate "," ActionTemplate ","  Term ","
                                Term "," Variable ")" [EffectAxiomBody] "." ;
FluentTemplate = FluentName ["(" VarIdent { "," VarIdent } ")" ;
VarIdent = Variable | Identifier ;
ActionTemplate = ActionName ["(" Term { "," Term } ")" ;
EffectAxiomBody = ":" "-" ? ARBITRARY VALID PROLOG SOURCE CODE ? ;
```

Using the syntax defined above, an effect axiom for a fluent f must have one of the following forms:

i) $\mathbf{effect}(f(x_1, \dots, x_n), a(y_1, \dots, y_m), \vartheta_{old}, \vartheta_{new}, S).$

ii) $\mathbf{effect}(f(x_1, \dots, x_n), a(y_1, \dots, y_m), \vartheta_{old}, \vartheta_{new}, S)$:- $\mathit{\Gamma}$.

Here $x_1, \dots, x_n$ are terms that represent the parameters of the fluent f. During progression, these terms will be *unified* with the parameters of every instance of the fluent f in order to determine whether the effect should be applied or not. Therefore, $x_1, \dots, x_n$ can either be variables that will be bound to the arguments of the fluent instance by unification or ground entity identifiers that act as conditions for the effect. Any named variable among $x_1, \dots, x_n$ can either be used in a clause body $\mathit{\Gamma}$ or can re-appear in another place of the effect axiom. The term $a(y_1, \dots, y_m)$ specifies an *action template* that will be unified with the currently progressed action term, and the effect is only applied when the unification is possible. In this step, variables that were bound in the fluent template before act as conditions, and variables that are introduced in the action template are bound by unification to the arguments of the progressed action term. Following the action template, ϑ_{old} represent the value of the fluent instance in the current situation, i.e. before the effect of the action is applied. Like before, any term that is specified in this argument will be matched against the actual current value of the fluent instance and the effect is only applied when both values can be unified. The next parameter of the effect axiom, ϑ_{new}, is meant to hold the new value to which the fluent instance will be set when the effect is applied. This can either be a predefined value, a variable that is introduced in the fluent or action template, or a fresh variable that is set in the clause body. In order to do this, the clause body Γ can use the last argument S of the effect axiom, which is a variable that will be set to the current situation during progression. Finally, Γ can be any valid Prolog goal that evaluates to true if and only if the effect should be applied and that binds all named variables in the effect axiom except S.

Some concrete examples for effect axioms of varying complexity can be found in Figure 3.5, which shows the effect axioms for the fluents defined in Figure 3.3. The first four specify that the robot's horizontal velocity `vx` is set directly to either a leftward (-1) or rightward (1) movement by the horizontal move actions and set to 0 when the movement step finishes or fails or the robot starts a vertical move. Here, the new value is simply specified in the axiom itself and no calculation is needed. The axioms for the vertical velocity vy are not shown here because they are very similar to those of `vx`. In contrast to that, the effect axioms for the position fluent `xpos` are a bit more complex. Here, for the effect of a successful step, i.e. a `step_succeeded` event, a clause body is specified that not only considers the direct effect on the position of the robot but also the indirect effect on the position of any item that is carried by the robot.

The axioms that follow after the movement effects establish some self-explanatory rules for when an item is carried by a robot. After that, the content of the `next_task` fluent is defined. In this example, a single fluent is used to store both the assigned item and the assigned target workstation. To do that, a Prolog term is created with the arbitrary functor `d`. Alternatively, it would also be possible to use two separate fluents to store the assignment. This choice is up to the modeler. In order to make sure that robots can be assigned new tasks after they have delivered an item, the `netx_task` fluent has to be reset to `none`, which is specified by the second effect axiom for `next_task`. Additionally, the reset also happens when the item is dropped, which indicates that the robots will later be equipped by a relenting control strategy that simply abandons a dropped item and waits for the next assignment (see Section 3.3). On the other hand, when an item is eventually delivered to its target workstation, the following effect axioms establish that this is tracked both by the fluent `delivered_to` for the item itself and by `delivered_item_count` for the workstation. At the other side of the delivery process, the effect axioms for `request_queue` handle the contents of the request queue. When a request message from a workstation arrives, the workstation's id is added to the list that is stored in `request_queue` using Prolog's built-in `append` function. Consistent to that, this entry is removed again when it has been processed, i.e. when an item has been a assigned to be delivered to the requesting station. Finally, in the last effect axiom, it is stated that a collision of a severity level higher than 7 will break both robots.

```
effect(vx(Robot), move_right(Robot), _, 1, _).
effect(vx(Robot), move_left(Robot), _, -1, _).
effect(vx(Robot), move_up(Robot), _, 0, _).
effect(vx(Robot), move_down(Robot), _, 0, _).

effect(vx(Robot), step_succeeded(Robot), _, 0, _).
effect(vx(Robot), step_failed(Robot), _, 0, _).
... similar for vy

effect(xpos(O), step_succeeded(Robot), _, X, S) :-
      (O = Robot, ! ; carrying(Robot, O, S)),
      vx(Robot, Vx, S),
      xpos(Robot, OldX, S),
      X is OldX + Vx.

... similar for ypos

effect(carrying(Rob, Item), grab(Rob, Item, _), _, true, _).
effect(carrying(Rob, Item), drop(Rob, Item), _, false, _).
effect(carrying(Rob, Item), deliver(Rob, Item, _), _, false, _).
effect(carrying(Rob, Item), accidental_drop(Rob, Item), _, false, _).

effect(next_task(Rob), assign_task(_, Rob, Item, Workstation), _,
      d(Item, Workstation), _).
effect(next_task(Rob), deliver(Rob, _, _), _, none, _).
effect(next_task(Rob), drop(Rob, _), _, none, _).
effect(next_task(Rob), accidental_drop(Rob, _), _, none, _).

effect(delivered_to(Item), deliver(_, Item, Station), _, Station, _).
effect(delivered_item_count(Ws), deliver(_, _, Ws), OldCount, NewCount, _) :-
      NewCount is OldCount + 1.

effect(request_queue(C), request(Ws, C), OldQueue, NewQueue, _) :-
      append(OldQueue, [Ws], NewQueue).
effect(request_queue(C), assign_task(C, _, _, Ws), OldQueue, NewQueue, _) :-
      delete(Ws, OldQueue, NewQueue), !.

effect(broken(Rob), collision(R1, R2, Severity), _, true, _) :-
      (R1 = Rob, ! ; R2 = Rob), Severity > 7.
```

Figure 3.5: Effect axioms in the multi-robotics example.

As mentioned before, the SALMA runtime automatically translates effect axioms into successor state axioms in the form that was introduced in Section 2.2. In fact, during the model initialization phase, a new dynamic *fluent clause* is added to the Prolog clause database which combines all effect axioms for the fluent and adds a default case that returns the original value for any action term that is not covered by the effect axioms. The clause for a fluent f with parameters $x_1, \ldots, x_n$ will have the form $f(x_1, \ldots, x_n, S)$:-Γ. for a boolean (i.e. relational) fluent and $f(x_1, \ldots, x_n, V, S)$:-$\Gamma$. for a functional fluent, where S is the situation variable and V will be bound to the value of the fluent instance.

Besides regular fluents, whose contents are updated in each step according to the effect axioms, a model can also contain *derived fluents* whose values are calculated based on the values of others. Hence, derived fluents are just situation dependent functions and predicates that are specified like normal Prolog predicates. In fact, for each derived fluent declaration **derived_fluent**$(f, [x_1 : t_1, \ldots, x_n : t_n], t_f)$., the modeler has to add a Prolog clause with a signature of the form $f(X_1, \ldots, X_n, V, S)$ or $f(X_1, \ldots, X_n, S)$ where $X_1, \ldots X_n$ are Prolog variables that represent the derived fluents parameters, V a variable that will be bound to the return value, and S a variable that will contain the current situation when the derived fluent is evaluated. For a boolean (i.e. relational) derived fluent, the return value is not needed, so the second form is used. Both forms can be seen in Figure 3.6 that shows the derived fluents for the running example. Most of these definitions are quite self-explanatory. The clause for `dist_from_station` calculates the Manhattan distance (see [Bla06]) between a robot and a workstation, which is used because robots can only move in strictly horizontal or vertical steps. The predicate `unassigned` is simply a check that `next_task` contains `none` for a given robot, and `moving` checks whether the absolute value of a robot's velocity is higher than 0. Additionally, the predicate ready is used as a shortcut to check whether the robot is ready for the next step, which requires it to be functional and not moving. Another shortcut is `undelivered`, which will be used by the coordinator's control procedure to select items that neither have already been delivered nor are currently scheduled to be delivered to a workstation. Finally, the two derived fluents `task_item` and `task_workstation` extract the item and the workstation from an assignment term. As will be shown in Section 3.3, this is helpful for the definition of control procedures because it avoids having to deal with Prolog terms within Python.

3.2.5 Action Precondition and Schedulability Axioms

Besides the successor state axioms, the other essential axiom type of the situation calculus is the *action precondition axiom*, usually denoted as **poss**. This axiom specifies whether a specific action instance is executable in a given situation. In SALMA, action preconditions are directly formalized as Prolog

```
dist_from_station(Rob, Station, Dist, S) :-
        xpos(Rob, X, S),
        ypos(Rob, Y, S),
        stationX(Station, Sx), stationY(Station, Sy),
        Dist is abs(X - Sx) + abs(Y - Sy).

unassigned(Rob, S) :-
        next_task(Rob, none, S).

moving(Rob, S) :-
        vx(Rob, Vx, S), abs(Vx) > 0, !
        ;
        vy(Rob, Vy, S), abs(Vy) > 0.

ready(Rob, S) :-
        not moving(Rob, S), not broken(Rob, S).

undelivered(Item, S) :-
        delivered_to(Item, none, S),
        domain(robot, Robots, S),
        not (member(R, Robots), next_task(R, d(Item, _), S)).

task_item(Rob, Item, S) :-
        next_task(Rob, Task, S),
        Task = d(Item, _), !
        ;
        Item = none.

task_workstation(Rob, Ws, S) :-
        next_task(Rob, Task, S),
        Task = d(_, Ws), !
        ;
        Ws = none.
```

Figure 3.6: Derived fluents in the multi-robot example.

clauses.

Definition 3.8 (Action Precondition Axioms)**.** The syntax of an action precondition axiom is given by the following EBNF-rule:

```
PossAxiom = "poss" "(" ActionTerm "," Variable ")" [ PossAxiomBody ] "." ;
ActionTerm = Variable | ActionName ["(" Term {"," Term } ")"] ;
PossAxiomBody = ":" "-" ? ARBITRARY VALID PROLOG SOURCE CODE ? ;
```

Here, the action term before the comma will be unified with the action term that is about to be applied in progression. In fact, SALMA's progression algorithm will only perform the progression of an action term if a **poss** axiom clause exists where the action term is unifiable and the axiom clause evaluates to true. The second parameter of the **poss** axiom contains a variable that will be bound to the current situation when the axiom is evaluated.

One subtle question about action precondition axioms is what should actually happen when an agent attempts to perform an action that is not possible in this situation. For the original use of the situation calculus, the fact whether an action is executable plays an important role in proofs or in planning. In simulation, however, the system *commits* itself to performing an action, i.e. there is no possibility of backtracking as in planning. One possible rather intuitive reaction to the attempt of performing an impossible action would be to *block* the execution of the performing agent process until the action becomes possible again. However, this would actually establish a kind of implicit synchronization mechanism that might not be adequate for every model. Therefore, the SALMA simulation engine by default treats the attempt of performing an impossible action as an error and cancels the simulation run. This forces the modeler to include explicit tests into the agent control procedures that make sure an action is only performed when its precondition is satisfied. Although this makes the behavioral model more verbose than in other approaches like GoLog, it actually fits well to SALMA's understanding that aspects like the possibility of a violated precondition are exactly what should be represented within the simulated model.

Yet another perspective on preconditions exists in the context of exogenous action (events). There, the **poss**-axioms are used by the simulation engine to select event instances that may occur in the current simulation step. As indicated in Section 2.3, this corresponds to what is usually called *activity scanning*, which is one possible strategy for *discrete event simulations*. However, in many cases, this mechanism is inefficient because aspects of the world state that are relevant for the precondition can only change at certain distinct times. For these cases, it is better to use an *event scheduling* mechanism where the time when an event instance should occur is determined in advance. For this purpose, SALMA introduces *schedulability axioms* that determine whether it is possible in the current simulation step to decide *at which point in the future* an event instance should occur. The syntax of such a statement is similar to the classical action precondition.

Definition 3.9 (Schedulability Axioms)**.** The syntax of a schedulability axiom is given by the following EBNF-rule:

```
SchedAxiom = "schedulable" "(" ActionTerm "," Variable ")"
                                     [ PossAxiomBody ] "." ;

ActionTerm = Variable | ActionName ["(" Term {"," Term } ")"] ;
PossAxiomBody = ":" "-" ? ARBITRARY VALID PROLOG SOURCE CODE ? ;
```

Like before, the action term in the axiom will be unified with an action term, in this case that of an exogenous action instance or with an instance of an exogenous action choice. Only if there is a schedulability axiom clause that evaluates to true, an event instance may be scheduled at a time that is determined by the occurrence probability distribution of the event (see Section 3.4.2). For exogenous action choices, a `schedulable` axiom may only be specified for the choice group itself but not for the events declared as mutually exclusive outcomes.

The action precondition and schedulability axioms for the robotics example are shown in Figure 3.7. At first, the movement actions are set to be only possible if the robot is not broken. A slightly more complex condition is needed for `pickUp`: first of all, the robot has to be at the same grid position as the item. Besides that, it can only grab an item that is not currently being carried by any other robot. The outcomes for `pickUp` add no further requirement, so they are declared to be possible anytime by setting the body of the corresponding **poss**-axioms to `true`. The same is done for the actions `request` and `assign_task`, which effectively implies the tolerable simplifications that the coordinator's request message queue is unbounded and assignments could be changed at any time. On the other hand, in order to deliver an item, a robot must not be broken, must be carrying the item, and has to be at the same grid cell as the target workstation.

The *preconditions* for these actions that are actively performed by robots are followed by two further **poss**-axioms for the events `collision` and `accidental_drop`. While for `accidental_drop`, the condition is again simply that the item is actually carried by the robot, formulating the precondition for the collision event requires more thought. One obvious condition is that two robots are close enough to each other, which comes down to being at the same grid location in the model of the ongoing example. This constraint is added as a simple coordinate comparison. However, this constraint would always be satisfied when both variables `R1` and `R2` point to the same robot, and therefore this case is excluded explicitly by the constraint `R1 \= R2` (where `\=` is Prolog's

```
poss(move_right(R), S) :- not broken(R,S).
. . . same for other move-actions

poss(pickUp(R, I), S) :-
    not broken(R, S),
    xpos(R, Xr, S), xpos(I, Xi, S), Xr =:= Xi,
    ypos(R, Yr, S), ypos(I, Yi, S), Yr =:= Yi,
    domain(robot, Robots),
    not (member(R2, Robots), carrying(R2, I, S)).

poss(grab(_, _, _), _) :- true.
poss(drop(_, _), _) :- true.

poss(request(_, _), _) :- true.
poss(assign_task(_, _, _, _), _) :- true.

poss(deliver(R, I, Station), S) :-
    not broken(R, S), carrying(R, I, S),
    xpos(R, Xr, S), stationX(Station, Xs), Xr =:= Xs,
    ypos(R, Yr, S), stationY(Station, Ys), Yr =:= Ys.

poss(accidental_drop(R,I), S) :- carrying(R,I,S).

poss(collision(R1, R2, _), S) :-
    R1 \= R2,
    xpos(R1, X1, S), xpos(R2, X2, S), X1 =:= X2,
    ypos(R1, Y1, S), ypos(R2, Y2, S), Y1 =:= Y2,
    (moving(R2, S), ! ; moving(R2, S)).

schedulable(step_finished(Rob), S) :-
    moving(Rob, S).
schedulable(accidental_drop(R,I), S) :-
    action_occurred(grab(R,I), S).

schedulable(request(_, _), _) :- true.
```

Figure 3.7: Action precondition and schedulability axioms in the multi-robot example.

syntax for $\neq$). As a last condition, the axiom also states that a collision occurs only if at least one of the robots is moving.

On closer inspection, there is an important difference between the event `collision` and the other events in the model with respect to the way the simulation determines their occurrence times. Actually, in order to decide if a collision between two robots can occur within a certain time interval, the simulation system has to evaluate the **poss**-axiom for each step because the coordinates of the robots may be different each time. In contrast to that, the occurrence time for an accidental drop event can be chosen *in advance* as soon as the item is grabbed. In other words, the event can be *scheduled* for a future point in time that is, for instance, determined by sampling from a geometric probability distribution. For cases like that, a schedulability axiom is required that specifies under which conditions it is possible to decide whether an event should occur and, if necessary, calculate the occurrence time. Therefore, the schedulability axiom for `accidental_drop` uses the predefined predicate `action_occurred` to assert that a `grab` action has occurred in the current step. Similarly, the axiom for the event choice group `step_finished` states that the end of a movement, i.e. one of the events `step_succeeded` and `step_failed`, can be scheduled right from its start, which is detected by the derived fluent `moving`. This makes it possible to model various forms of uncertainty with respect to the robots' motion by choosing adequate probability distributions for the delay of the `step_succeeded` and `step_failed` events. Finally, the last axiom in Figure 3.7 makes the `request` event schedulable at any time, which means that the arrival times of new requests will be determined solely by the probability distribution that is assigned for the event.

3.2.6 Representation of Time in SALMA

As mentioned in Section 2.2.1, several alternative approaches for modeling time in the situation calculus have been proposed. Since SALMA uses a discrete time structure and the models are used in simulation rather than history-based analysis, time can simply be represented by one global integer fluent, which is called *time*. The following axioms defines how time can progress:

```
primitive_action(tick, [steps:integer]).

fluent(time,[], integer).

effect(time, tick(Steps), TOld, T, _) :- T is TOld + Steps.

poss(tick(_), _) :- true.
```

Time *advances* only through a special action *tick* that has a parameter *steps* which represents how many *time units* the global simulation world time should be advanced. Although the action *tick* is defined as a regular primitive action like the ones in the previous examples, the simulation system does not allow it to be executed from any agent process (see Section 3.3). Instead it is executed directly by the simulation engine at the end of each simulation step. This basic model is further extended by relative time measurements called *clocks* that mark the times of the most recent occurrences of actions, events, and fluent changes. In particular, this can be used to define precondition axioms and probability distributions that model the time between actions or events. For instance, it would be possible to define a minimum duration of 10 time units for every step of a robot from the running example in this chapter:

```
poss(step_finished(Rob), S) :-
    moving(Rob, S),
    get_last_change_time(moving, [Rob], TLast),
    time(TNow, S),
    TNow >= TLast + 10.
```

With the means mentioned above, the situation calculus version used in SALMA is at least theoretically expressive enough to describe timing aspects in a similar way as with *timed automata* [AD94] or *probabilistic timed automata* [Jen96], which are widely used modeling paradigms for real-time systems (see also Section 2.4).

3.3 The SALMA Agent Process Definition Language

In the previous section it was shown how the SALMA domain description language (SALMA-DDL), a variant of the situation calculus, can be used to create a model of the simulated *domain*, i.e. the entities, features, and relations of the simulated system. In particular, this domain model defines *the actions* that an agent can perform together with the effects of these actions. What remains to be specified is the actual *behavior* of the agents. In the context of the situation calculus, this is commonly done using GoLog or one of its many variants (see Section 2.2.4). Usually, a GoLog interpreter is implemented in Prolog and thus very closely connected to the situation calculus model. Besides facilitating formal analysis, the choice of Prolog as platform has also practical advantages because it allows creating interpreters that are very concise and well-structured.

Nevertheless, SALMA's simulation engine is implemented in Python and instead of supporting GoLog, it offers an internal domain-specific language (DSL) in Python for the definition of agent processes. The main motivation behind this decision is to allow the modeler to integrate custom Python code into the agent control procedures. By this, it is possible to leverage the full potential of Python and its rich set of mathematical and scientific libraries for performing tasks that drive the control decisions of the agent, like searching, planning, or learning. At the same time, SALMA's agent process definition language (SALMA-APDL) is designed in a way that ensures a clear separation

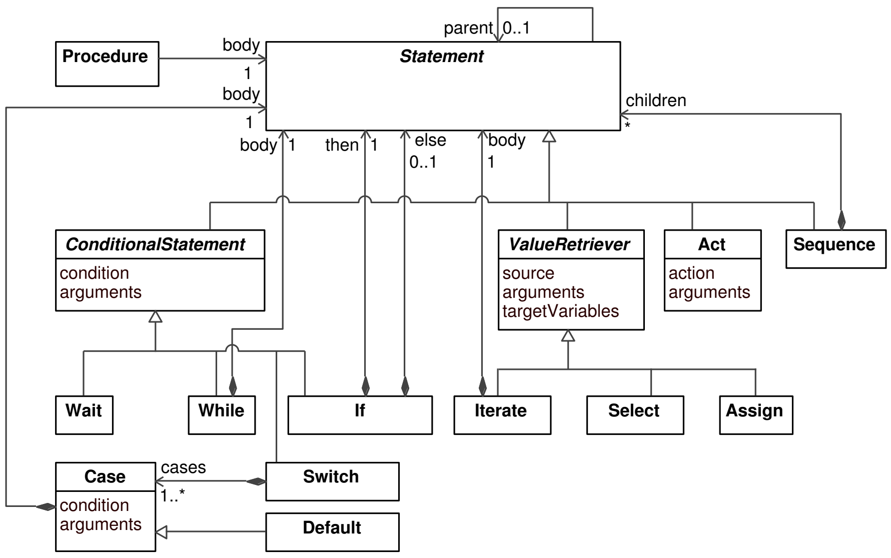


Figure 3.8: Class hierarchy of the SALMA-APDL.

between these potentially complex calculations and the actual agent control logic. Altogether, the chosen architecture provides the facilities that are required for the realization of complex modeling and simulation projects. This section introduces SALMA's agent process definition language and demonstrates its use by extending the multi-robotics example from the last section with behavior for the robots and the controlling station. As for the SALMA-DDL before, this section concentrates on the syntax and practical usage of the SALMA-APDL. A precise discussion of the semantics is postponed until Section 3.6.

In order to fully understand the usage of the SALMA-APDL, it is important to keep in mind that it is an *internal domain specific language*, i.e. an API that is realized as a set of regular Python classes that represent the statements of the SALMA-APDL. These classes are instantiated and assembled together to an object graph which stores all information that is needed by the simulation engine to execute the control logic. The class hierarchy of the SALMA-APDL is shown in Figure 3.8. Since there is no explicit instantiation keyword like `new` in Python, the constructors of the statement classes can be used like functions, which makes procedure definitions in the SALMA-APDL look similar to procedural code although they are in fact sequences of nested objects. However, it is desirable to describe the language in a way that captures the *effective syntax* that is actually used to represent model behavior. For this purpose, a specialized notation is introduced next.

3.3.1 Notation

Since the SALMA-APDL is a regular Python API and as such adheres to the general syntax of Python, it would not be very informative to discuss the "low-level" syntax by means of a classical EBNF like it was done in the last section. Details of the Python language are not covered here but can be found in the Python Language Reference [Pyt15a]. Instead, the following presentation regards Python expressions as basic building blocks of the language and focuses on which expression types are admissible for the individual arguments. This is denoted by a colon that is followed by a type name. The types that are used in the following include basic predefined types like `string` or `int` and classes defined within the SALMA-APDL. Furthermore, lists element types can be specified with list < type >, similar to generics in Java or templates in C++. Apart from these specialties, the notation borrows the essential elements from the EBNF, namely `[...]` for an optional occurrence, `\{...\}` for 0, 1 or multiple repetitions, and | for a choice.

For each of the concrete classes defined in the SALMA-APDL, there is a *constructor* that is used to instantiate them. To distinguish the definition of a constructor, the class name is enclosed in angular brackets, e.g. `<While>`. Besides these types that can actually be instantiated, there are abstract classes and "virtual" generalized types that are used as *nonterminal* symbols just like in the classical EBNF.

3.3.2 Agent Control Procedures

A *control procedure* is the top level element of the SALMA-APDL. As such, it wraps a sequence of *statements* to define a control flow that can then be assigned to agent processes.

Definition 3.10 (Structure of an Agent Control Procedure)**.** An agent control procedure is defined according to the following grammar:

<*Procedure*> = **Procedure**(procName: string, body: *StatementOrSequence*) ;
StatementOrSequence = *Statement* | list<*Statement*> ;
Statement = *Sequence* | *Act* | *Assign* | *If* | *Switch* | *While* |
    Select | *Iterate* | *Wait* ;
<*Sequence*> = **Sequence**(list<*Statement*>) ;

Essentially, Definition 3.10 resembles the class hierarchy of Figure 3.8. However, it can be seen in the definition of *StatementOrSequence* that a Python *list* can be used for the body. I this case, a *Sequence* instance will be created implicitly. Since Python allows the use of square brackets to build *list literals*, it is possible to write procedures in the following format:

$$
\begin{aligned}
proc = Procedure([\\
&\quad \text{statement}_1,\\
&\quad \ldots\\
&\quad \text{statement}_n])
\end{aligned}
$$

This shortcut for statement sequences will be used in several places and is one of the key factors that make the SALMA-APDL usable like a specialized external agent programming language. In the remainder of this section, each statement type that may appear within procedures will be introduced. It starts with the one that is most directly connected to the situation calculus domain model, namely the execution of an action.

Definition 3.11 (Action Execution)**.** Within an agent control procedure, an action execution is triggered by the Act-statement that is defined as follows:

<*Act*> = **Act**(actionName: string, arguments: *ArgumentList*) ;
ArgumentList = list<*Argument*> ;
Argument = string | number | boolean | *Variable* ;
<*Variable*> = **Variable**(varName: string [, varType: string]) ;

When an `Act` statement is reached in the control flow, an action term will be constructed for the given action name and the given arguments, where strings are translated to entity identifiers. When the simulation engine reaches an `Act` statement, it attempts to execute the given action in the current simulation step. If the created action instance is possible, i.e. the corresponding `poss`-axiom is satisfied, it will eventually be executed by means of the *progression* operator. However, if the precondition axiom is not satisfied, the simulation is canceled and an error is risen.

The Python expressions in the arguments of the action are not evaluated when the action term is created but when the statement object is instantiated during the simulation's setup. Therefore, they will act as constant values with respect to the action execution. Alternatively, instances of the class `Variable` can be used to mark variables that will be evaluated within the procedure's context when the statement that contains the variable is executed. For example, in the multi-robots scenario, the robot's control procedure contains the statements visible in Figure 3.9 where the action `deliver` is executed using the previously initialized `Variable` objects, `targetItem` and `targetWs` as arguments.

Naturally, variables do not only appear in action executions but in many other places. As the grammar above reveals, it is also optionally possible to specified a type in the constructor of the `Variable` object. However, this is

```
targetItem = Variable("targetItem")
targetWs = Variable("targetWs")
...
Act("deliver", [SELF, targetItem, targetWs])
...
```

Figure 3.9: Example for an action execution.

only required for variables that are used in the statements `Select` and `Iterate` that are introduced later. The more urgent question for now is how values are actually assigned to variables.

Definition 3.12 (Variable Assignment)**.** An `Assign`-statement retrieves a value from a *value source* and assigns it to a variable in the context of the current procedure. Its syntax is defined as follows:

```
<Assign> = Assign( target: TargetDef, source: ValueSource
                    [, arguments: ArgumentList] ) ;
TargetDef = string | Variable | list< string | Variable > ;
ValueSource = pythonFunction | FluentName | PrologFunctionName |
                    PythonExpression  ;
FluentName = string ; PrologFunctionName = string ;
PythonExpression = string ;
```

Here, the *target* is either a single variable or a list of variables to which the calculated value or values will be assigned (for multiple variables, the value retrieved from the source must be a tuple of appropriate length). Each variable is either identified by its name or by a `Variable` object. The *source*, from which the value is retrieved, can be one of the following:

a) A fluent whose name is given in `source` and whose instance is specified by the given arguments.

b) A situation-independent Prolog function (i.e. a predicate that returns a value by binding it to a variable in its last argument), whose name is given in `source` and to which the given arguments will be passed.

c) A Python function to which a pointer is passed in `source` and to which the given arguments will be passed. Within the function, it is also possible to access the world's state, i.e. fluent instances, entity domains, etc. (see Section 3.4.5).

d) A Python expression, given as string in `source` that will be evaluated with Python's built-in function `eval()`. Within the expression, there will be an extensive set of name-value bindings that provide access to variables and fluents (see Section 3.4.5).

In the delivery robots scenario, examples for uses of fluents, Python expressions and a Python function can be found within the agent control procedures, for instance in the fragment shown in Figure 3.10 that is reached when a robot has just been assigned to a new task:.

```
...
Assign(targetItem, "task_item", [SELF]),
Assign(targetWs, "task_workstation", [SELF]),
Assign(tx, "targetItem.xpos"),
Assign(ty, "targetItem.ypos"),
...
```

Figure 3.10: Examples for variable assignments within the robot control procedure.

Here, the robot directly accesses the derived fluents `task_item` and `task_workstation` to access the assigned item and the target workstation. After that, it uses two simple Python expression to access the position of the item. In fact, the SALMA framework provides a dynamic function and attribute mapping system that allows the modeler to access fluents and constants in an object-oriented manner, which significantly increases conciseness and readability (see Section 3.4.5). However, sometimes the calculation of the assigned value is too complex to be expressed in a single expression. In these cases, a Python function may be used as a source. For example, Figure 3.11 shows how the control procedure of the coordinator agent uses the function `select_item` to select the closest item to a given robot that has not been delivered and also is not yet assigned to be delivered, which is checked by `item.undelivered`. The first argument of this function is set in the `Assign`-statement with the variable `r` to which an idle robot has been assigned before. Eventually, the chosen pair of robot and item is used in an `assign_task` action to allocate the delivery task.

Even this rather simple example shows how valuable the tight integration with Python as a general purpose language is. Although in this situation it would also be possible to perform the calculation within a derived fluent clause in Prolog, it is easy to imagine more sophisticated selection strategies that might involve complex algorithms from fields like Graph theory, optimization, or machine learning. For all of these purposes there exist mature Python libraries that can easily be integrated in functions like the one above. At

```
def select_item(rob: Agent, ctx: EvaluationContext=None, **kwargs):
    if rob is None:
        return None
    closest_item = None
    min_dist = None
    for item in ctx.getDomain("item"):
        if item.undelivered:
            dist = np.abs(rob.xpos - item.xpos) + np.abs(rob.ypos - item.ypos)
            if min_dist is None or dist < min_dist:
                closest_item = item
                min_dist = dist
    return closest_item

        . . .
        Assign(i, select_item, [r]),
        . . .
         Act("assign_task", [SELF, r, i, ws]))])
```

Figure 3.11: Example use of a Python function as a value source for variable assignment.

the same time, the `Assign` statements act as *integration points* into the agent control procedures that make the *data flow* visible immediately.

For making decisions within agent control procedures, the SALMA-APDL defines two of the most common conditional statements, namely **If** and **Switch**.

Definition 3.13 (Conditional Statements)**.** Conditional control blocks can be created with the statements `If` and `Switch`.

```
<If> = If( condition: Condition [, arguments: ArgumentList],
              then: StatementOrSequence [, else: StatementOrSequence] ) ;
Condition = pythonFunction | FluentName | PrologFunctionName |
              PythonExpression  ;
<Switch> = Switch( CaseOrDefault {, CaseOrDefault } ) ;
CaseOrDefault = Case | Default ;
<Case> = Case( condition: Condition [, arguments: ArgumentList],
                  then: StatementOrSequence) ;
<Default> = Default(body: StatementOrSequence) ;
```

As usual, the `If`-statement tests a condition and executes the contained statement in `then` (which could also be a sequence) only if the condition is true. Optionally, another statement can be specified in `else` to be executed when the test fails. For the condition, the same options are available as for the value source of the `Assign` statement from Definition 3.12, i.e. a fluent, a Prolog function, a Python function, or a Python expression. In any case, the given condition source is evaluated in the current context and is expected to return a boolean value.

The `Switch` statement is inspired by the well-known construct from general purpose programming languages like Java or C/C++. The cases are specified by means of `Case` objects that contain a condition together with a statement. Optionally, one `Default` case can be specified that is executed when no condition is satisfied. It is worth noticing that these objects are directly given as arguments parameters for `Switch` instead of as a list, which helps distinguishing the case collection from a *sequence*. When the `Switch` statement is executed, the conditions of every `Case` object are evaluated in the same order they were specified and the first case for which the condition is satisfied is chosen for execution. If no test succeeds and a default case exists, then the statement specified in the `Default` block is executed. Otherwise, the control flow continues with the next statement after the `Switch`.

It is clear that conditional statements are indispensable for any agent control language. Often, an `If` is necessary to test a condition before an action is executed. In other cases, the agent decides between multiple alternatives how to proceed. Although the `If` statement could obviously also be used for this purpose, a `Switch` construct can often be useful for avoiding a deep nesting of statements that would obfuscate the real logic behind the decision.

Examples for the use of both `If` and `Switch` can be seen in Figure 3.12. This figure also contains a loop realized by the `While`-statement, which is defined next.

Definition 3.14 (While Loops)**.** The SALMA-APDL supports while loops in their usual form:

<*While*> = **While**(condition: *Condition* [, arguments: *ArgumentList*],
body: *StatementOrSequence*) ;

The condition used for `While`-loops is built in the same way as that of the `If`-statement. The statement in the loop body is consequently repeated until the condition fails, and unlike most procedural programming languages, the SALMA-APDL intentionally does not support a `break` statement that cancels loop execution. Although having such an option would be convenient in some cases, it could also make the control flow harder to trace and reason about.

Another essential control flow element appears in Figure 3.12 which is required to realize almost any nontrivial agent behavior: the ability to *block execution* until a condition is fulfilled. This is achieved by the `Wait` statement.

Definition 3.15 (Wait Statement)**.** In a SALMA-APLD procedure, a `Wait` statement can be used to block execution until a condition is true. Its syntax is defined as follows:

```
PRECONDITION TEST BEFORE DELIVERY:
. . .
If("not self.broken and "
  "dist_from_station(self, targetWs) == 0 and carrying(self, targetItem)",
        Act("deliver", [SELF, targetItem, targetWs]))

MAIN MOVEMENT CONTROL LOOP:
. . .
While("not self.broken and self.next_task != None and "
            "(self.xpos != tx or self.ypos != ty)", [
   Switch(
      Case("self.xpos < tx", Act("move_right", [SELF])),
      Case("self.xpos > tx", Act("move_left", [SELF]))
   ),
   Wait("self.ready"),
   Switch(
      Case("self.ypos < ty", Act("move_down", [SELF])),
      Case("self.ypos > ty", Act("move_up", [SELF]))
   ),
   Wait("self.ready")])
```

Figure 3.12: Example use of the control flow statements in the SALMA-APDL.

<*Wait*> = Wait(condition: *Condition* [, arguments: *ArgumentList*]) ;

When a `Wait`-statement is executed within an agent process whose condition is true, the control flow simply proceeds with the following statement. However, when the condition is false, the control flow leaves the current process and continues the execution of other processes. The process is *blocked* and only continued when the condition becomes true. This can only happen due to the change of a fluent value caused by an action performed in another process or an event.

Although the precise simulation semantics will not be discussed until Section 3.6, it is important to realize that action executions and blocked `Wait` statements are the only points in an agent control procedure where *context switches* occur in the simulation. The parts in between these points are not interleaved with other actions, i.e. they form atomic blocks. This means, for example, that an infinite loop in one process would actually block not only the process that contains it but the whole simulation.

In the robot's movement control loop shown in Figure 3.12, the `Wait` statements are used right after a movement is started to wait until the step has ended, which is indicated by the derived fluent `idle`. Without waiting for the idle state, the next move action would be executed before its precondition is fulfilled, which would cause the simulation to be canceled. Although this requirement for explicitly awaiting might seem cumbersome at first, this pattern represents the actual behavior of the modeled agent more accurately.

The set of statements defined above would, in principle, be sufficient to define any intended agent behavior. However, the SALMA-APDL offers two additional constructs that are very useful in many situations and actually establish a bridge to logic programming.

Definition 3.16 (Select Statement)**.** The `Select` statement chooses the first combination of argument values that make a Prolog predicate true.

> <*Select*> = **Select**(predicateName: string, arguments: *ArgumentList*) ;

Different from the `Assign` statement, the source for `Select` cannot be defined in Python but has to be either a relational (boolean) fluent, a boolean derived fluent, or a situation-dependent Prolog predicate. When the `Select` statement is executed, this predicate is called similarly as during the `Assign` statement. However, every variable that is not bound to a value at that time, and for which one of the entity types is specified as type, is treated as a *free variable.* For these, the corresponding argument positions are filled with fresh Prolog variables and the type information is used to set up membership constraints with respect the sort domains. Due to Prolog's evaluation scheme, the system will now use backtracking search to find a combination of values for which the called goal succeeds. If at least one such combination exists, the first one is chosen and each free variable is bound to its corresponding value. Otherwise, all free variables are set to `None`.

In many cases, not only the first but all possible solution for a predicate or a fluent should be considered. In this case, a construct is needed that iterates over these solutions.

Definition 3.17 (Iterate Statement)**.** The statement `Iterate` repeats its nested body for each possible solution of a relational fluent or Prolog predicate and binds free variables according to each iteration.

> <*Iterate*> = Iterate(source: *ValueSource*, arguments: *ArgumentList*,
> body: StatementOrSequence) ;
>
> *ValueSource* = pythonFunction | *FluentName* | *PrologFunctionName* |
> *PythonExpression* ;

The source for the iteration can be any of those that are also used for variable assignment. However, their meaning is different: for fluents and Prolog predicates, the iteration is performed over all possible solutions in the sense

```
r, i, ws = makevars(("r", "robot"), ("i", "item"), ("ws", "workstation"))
Procedure([
  Iterate("self.request_queue", [ws], [
    Select("unassigned", [r]),
    Assign(i, select_item, [r]),
    If("i != None and r != None",
      Act("assign_task", [SELF, r, i, ws]))])])
```

Figure 3.13: Example for the use of `Select` and `Iterate`.

of Prolog's `findall` operator (see [DEDC12, chap. 5.3]). In contrast to that, a Python function or expression is expected to return a *list*. This list can either contain one individual value for each cell, if only one free variable was specified, or tuples with one values for each free variable.

For both the `Select` and the `Iterate` statement, examples can be found within the control procedure of the coordinator agent, which is shown in Figure 3.13. Here, the iteration is performed over the list of requests that the coordinator has received until the loop is entered. In each iteration, the variable `ws` is set to the next workstation from the request queue. Within the loop's body, a `Select` is used to retrieve the first robot that is `unassigned` according to the corresponding derived fluent. As described above, this robot is stored in the variable `r` and passed as an argument to the function `select_item` that was shown in Figure 3.11 whose return value is assigned to the variable `i`. Finally, if a value could be found for both `r` and `i`, the task is assigned using the `assign_task` action. The variables `r`, `i` and `ws` are setup beforehand using the utility function `makevars` that simply returns one `Variable` object for each tuple in the argument list, where the tuples contain both the name of the variable and its type. In fact, in this case the type information is only necessary for `r`, which is used in the `Select` statement. However, since it adds valuable information to the model, it is certainly a good practice to add type information in all variable declarations.

3.3.3 Agent Process Types

After the agent control procedures have been defined as described above, they have to be attached to *agent processes*, which in turn are assigned to agent instances during simulation setup. In fact, each agent can be equipped with multiple processes that are scheduled and executed independently from each other. SALMA distinguishes three types of processes:

1. A **one-shot process** is executed once at the start of the simulation. After the procedure of the process has been finished, the process is *terminated* and will not be executed again for the current simulation run.

2. A **periodic process** is executed repeatedly with a specified period. In fact, each time the procedure of a periodic process is exited, the process becomes *idle* and is restarted as soon as the next time-slot has started. If the runtime of a periodic process is longer than its period, it is restarted immediately.

3. A **triggered process** is executed as soon as its *trigger condition* becomes true. When the condition is found to be true while the process is already running, this fact is just ignored.

Corresponding to these process types, the SALMA framework offers three Python classes that are tightly connected to the SALMA-APDL and allow the creation of process instances that can be installed onto agents.

Definition 3.18 (Agent Process Definition)**.** An instance of a one-shot, periodic, or triggered process can be created with one of the following class constructors.

```
<OneShotProcess> = OneShotProcess( procedure: Procedure ) ;

PeriodicProcess = PeriodicProcess( procedure: Procedure, period: integer ) ;

TriggeredProcess = TriggeredProcess( procedure: Procedure,
          condition: Condition ) ;
```

Here, the condition of the `TriggeredProcess` statement is defined in the same way as it was for the conditional statements in Definition 3.13. In each simulation step, it will be evaluated within the evaluation context of the agent that owns the process.

Once the desired process instances have been created, they can be used to create *agent instances* that are finally added to the *world instance.* At this point, the domain of the SALMA-APDL is left and the responsibility is passed to the core part of the SALMA simulation framework, which is described next.

3.4 The SALMA Simulation Framework

Figure 3.14 shows an overview of the core of SALMA's simulation framework. It comprises a class structure that is able to capture both the static and the dynamic aspects of the simulation model. The core of the model is constituted by the singleton class `World` that stores collections of objects representing all

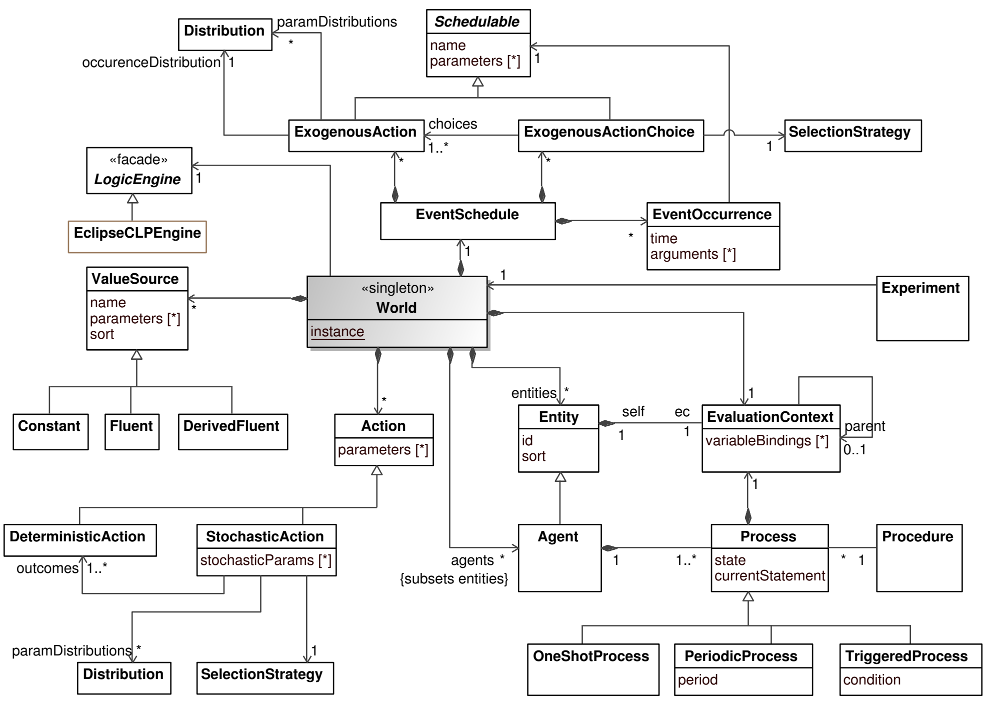


Figure 3.14: Overview of the core SALMA simulation framework.

ingredients of the SALMA domain model, namely fluents, actions, events, entities, and agents. Additionally, through indirect links, this also includes the agent's processes as well as the probability distributions for stochastic and exogenous actions. Besides this representation of the model and its configuration, the world instance has an *event schedule* that manages a queue of future event occurrences together with the occurrence times they are appointed to. On the other hand, the world instance is connected to an implementation of the `LogicEngine` interface which establishes the bridge to the concrete Prolog system in use[1].

The following section describes how the core classes of the SALMA framework are used to assemble the different parts of the simulation model that were discussed in the previous sections. It is also discussed how to fill the remaining gaps, namely the probability distributions and the initial situation at which the simulation should start. After that, Section 3.4.4 will explain how to set up a concrete *simulation experiment* based on this model, which includes making decision about simulation execution strategies and what information should be recorded for later analysis.

[1]At the moment, only the ECLiPSe Constraint Programming System [SS10, ecl06] is supported but since Python interfaces for other common implementations like SWI-Prolog [WSTL12] also exist, a port to these platforms would be possible.

3.4.1 Initial Setup

The first step in creating a simulation model in the SALMA approach is to define the *system domain model* as discussed in Section 3.2. In practice, this means that one or more Prolog source files are written that contain the declarations and axioms for fluents, actions, etc. In order to work with these sources, they have to be loaded and compiled via the Prolog interface. This happens when the path to the domain model source file is passed at the instantiation of a `LogicEngine` instance. At the same time, several global data structures within the Prolog runtime system are initialized that will later store all fluent instance and manage them during progression. The created engine is registered at the `World` class and a world instance is created. After that, the declarations from the domain model can be loaded, which automatically creates the corresponding instances of the classes `Fluent`, `DerivedFluent`, `Constant`, `DeterministicAction`, `StochasticAction`, `ExogenousAction`, and `ExogenousActionChoice`. These objects are registered at the world instance and the parameter lists of fluents, constants, actions, and events, as well as the sets of outcomes for stochastic actions and exogenous action choices, are initialized according to the declaration statements of Section 3.2. At the same time, the declared sorts are registered together with the subsort-relations associated with them. Altogether, these steps are performed with the following lines of Python code:

```
World.set_logic_engine( EclipseCLPEngine(<DOMAIN MODEL PATH>) )
world = World.create_new_world()
world.load_declarations()
```

3.4.2 Configuration of Actions and Events

After the domain model has been loaded, all of its elements can be accessed via the world instance. In particular, it is necessary to configure probability distributions and selection strategies for the actions and events:

For **exogenous actions**, an *occurrence distribution* has to be specified whose meaning depends on the declaration of the event. If a `schedulable` axiom is defined for the exogenous action, the occurrence distribution has to be numeric. When an event instance is detected to be schedulable in a simulation step, then the occurrence distribution is used to sample a *delay time* after which the occurrence of the event instance will be scheduled. On the other hand, if only a precondition for the event is specified by means of a poss `axiom`, then the occurrence distribution has to be boolean. If the precondition is satisfied for a specific event instance in the current simulation step, the occurrence distribution is used to decide whether the event instance should occur in the same step. Additionally, as mentioned in Section 3.2.3, an exogenous action can also have probabilistic parameters, for which the modeler has to set up adequate probability distributions, too.

For **stochastic actions**, a *selection strategy* has to be defined which is used to choose one of the possible *outcomes* when the stochastic action is performed by an agent. Often such a distribution simply assigns a fixed probability to each outcome, although any type of categorical distribution with the right value range can be used. Additionally, the modeler has to specify how the parameters of the chosen outcome action are going to be filled. This can either be a direct mapping to a controlled parameter that is set when the agent executes the action, or it could itself be another probability distribution that is able to yield values of the correct type.

Exogenous action choices, i.e. sets of mutually exclusive events, require a selection strategy similar to stochastic actions, with the difference that the choices are exogenous actions. Each of these options have to be configured like any independent event as described above.

As a concrete example for the configuration of probability distributions and selection strategies, Figure 3.15 shows an excerpt of one version of the delivery robots example. First, a *categorical selection strategy* is chosen for the exogenous action choice `step_finished`, where the probabilities for the events `step_succeeded` and `step_failed` are set to 0.8 and 0.2, respectively. For both of these step options, a constant occurrence distribution with the parameter 1 is defined, which means that each movement will either succeed or fail after exactly one time step. For the stochastic action `pickUp`, the probability of a successful grab is set to 70% vs a 30% chance of dropping the item. The parameters `r` and `i` of both outcomes, which represent the robot and the item, are *mapped* to the correspondent parameters of `pickUp`. Furthermore, the model specifies a uniform distribution for the `grip` parameter that selects a value between 1 and 10 with the same probability. For both the two following exogenous actions `accidental_drop` and `request`, geometric distributions are defined, with a success probability of 0.001 for the accidental drop and 0.01 for a request by a workstation. Finally, for the collision event, for which a `poss` but no `schedulable` axiom exists in the model (cf. Figure 3.7), the occurrence distribution decides directly about the occurrence within the current simulation step.

All probability distributions used in the current example have in common that they are defined using parameters that are fixed when the distribution object is created during simulation setup. Although this is sufficient in many cases, often a distribution depends on the current situation. Therefore, the SALMA framework makes it easy to define custom probability distributions by inheriting from the class `Distribution` and overriding the method `generateSample` or by using a special class `CustomDistribution` that delegates generation of a sample to a Python function. In both cases, the custom method or function receives the arguments that define the action or event instance and can access the full world state in order to derive a value. As an example, it could be reasonable to assume that the rate with which a workstation sends out item requests is related to the current number of “open orders”,

```
def setup_distributions(self):
    step_finished = world.get_exogenous_action_choice("step_finished")
    step_finished.selection_strategy = Categorical(step_succeeded=0.8,
                                                   step_failed=0.2)
    stepdelay = ConstantDistribution("integer", 1)
    world.get_exogenous_action(
        "step_succeeded").config.occurrence_distribution = stepdelay
    world.get_exogenous_action(
        "step_failed").config.occurrence_distribution = stepdelay

    pickup = world.get_stochastic_action("pickUp")
    pickup.selection_strategy = Categorical(grab=0.7, drop=0.3)
    grab = pickup.outcome("grab")
    grab.map_param("r", "r"), grab.map_param("i", "i")
    grab.uniform_param("grip", value_range=(1, 10))
    drop = pickup.outcome("drop")
    drop.map_param("r", "r"), drop.map_param("i", "i")

    accidental_drop = world.get_exogenous_action("accidental_drop")
    accidental_drop.config.occurrence_distribution =
        GeometricDistribution(0.001)

    request_event = world.get_exogenous_action("request")
    request_event.config.occurrence_distribution =
        GeometricDistribution(0.01)

    collision_event = world.get_exogenous_action("collision")
    collision_event.config.occurrence_distribution = BernoulliDistribution(1.0)
    collision_event.config.uniform_param("severity", value_range=(5, 10))
```

Figure 3.15: Definition of probability distributions for the delivery robots example.

i.e. requests for which no item has been delivered yet. If an item is interpreted as a resource that is needed at a workstation to perform some *task*, then it could be imagined that a workstation has a number N_{slots} of *slots* at which tasks can be performed in parallel. When the processes that happen at each slot are not modeled in more detail, it is a common choice to describe the times between requests at each slot by the same geometric distribution. However, if a request has been sent due to a current demand of a slot, this means that the slot will be waiting for the item to arrive and not issue any further requests in the meantime. Therefore, the total inter-arrival times for a workstation ws can be modeled by a geometric distribution that depends on the number of free slots $N_{free}(ws)$:

$$\begin{aligned} &P(T = t) \sim Geom(p_{tot}(ws)) \\ &\quad \text{where} \\ &p_{tot}(ws) = 1 - (1 - p_{slot})^{N_{free}(ws)} \end{aligned} \tag{3.1}$$

Here, p_{slot} is the parameter for each slot and $N_{free}(ws)$ is the number of slots of ws that are not currently waiting for the delivery of a requested item. This can be seen when $p_{tot}(ws)$ is understood as the probability that at least one free slot of ws issues a request at a given step. Since the model guarantees that every request will remain in the coordinator's queue until it is assigned to a robot, the number of free slots can be derived as follows:

$$\begin{aligned} N_{free}(ws) = N_{slots} &- card(\{\text{robots delivering to } ws\}) \\ &- card(\{\text{entries in request queue from } ws\}) \end{aligned} \tag{3.2}$$

Figure 3.16 shows how this probability distribution is realized by means of a short Python function `request_distrib` that is installed for the request event via a `CustomDistribution` object. This function is called each time the simulation algorithm has chosen an instance of the event `request` for which the corresponding `schedulable` axiom was true. The workstation, the coordinator, and the world's root *evaluation context* are passed to the function via the parameters `ws`, `c`, and `ctx`, respectively. Using these arguments, the function first uses the method count of Python's `list` type to count the number of times the workstation appears in the request queue of the coordinator. Then it iterates over all existing robots, i.e. the domain of the sort `robot` and counts the number of robots that are currently assigned to deliver an item to the workstation. With these numbers and the constants `N_SLOTS` and `P_SLOT`, the parameter `p_tot`is calculated as explained above and a sample from the corresponding geometric distribution is returned, which will be used to schedule the next occurrence for the event instance `request(ws, c)`.

```
def request_distrib(ws, c, ctx: EvaluationContext=None, **kwargs):
    ws_in_queue = c.request_queue.count(ws)
    assigned_robots = 0
    for r in ctx.getDomain("robot"):
        if r.task_workstation == ws:
            assigned_robots += 1
    n_free = N_SLOTS - assigned_robots - ws_in_queue
    p_tot = 1 - (1 - P_SLOT)**n_free
    return None if p_tot == 0 else np.random.geometric(p_tot)
...
request_event.config.occurrence_distribution = CustomDistribution(
    "integer", request_distrib)
```

Figure 3.16: Alternative custom distribution for the occurrence of item requests.

Altogether, it is obvious that the choice of probability distributions and outcome selection strategies plays an important role in the model. By the ability to integrate custom Python code, SALMA achieves a high level of flexibility that allows tackling even complex scenarios with much detail. Due to the important role of probability distribution definitions within the model, SALMA puts considerable effort into achieving a tight integration of the SALMA domain and behavior models into the Python environment in order to achieve conciseness and high readability. One of the facilities that serve this purpose is the event and action configuration API used in Figure 3.15. However, the most important facet of language integration is the fact that SALMA makes fluent and constant instances accessible in an object-oriented manner. This could already be seen, e.g., in the function `request_distrib` in Figure 3.16 and Section 3.4.5 will explore this topic a bit further. Before that, however, the remaining steps in configuring the simulation are taken, namely the population of the world's state at the simulation start and the definition of the concrete experiment.

3.4.3 Creating Entities, Agents, and the Initial Situation

In the classical situation calculus, the *initial situation* is the point of reference to which all formulas are projected through the regression operator. Similar, from the simulation perspective in SALMA, the initial situation represents the world state at the start, from which all future states evolve during simulation due to the repeated application of a progression step (cf. Section 2.2). Therefore, a value for every fluent and constant instance has to be defined for the initial situation. In fact, SALMA does not fill in undefined instances with default values but raises an exception when any uninitialized values are found at the start of the simulation. This is meant to force the modeler to consider every facet of the world's state at the beginning. Usually, significant parts of the initial values for constant and fluent instances are initialized by samples from probability distributions that have to be chosen carefully according to the concrete requirements of the simulation experiment. First, however, the model has to be populated with entities and agents that make up the domains of the declared sorts. For the delivery robots example, this means two sorts of agents are created, namely several robots and a coordinator. Additionally, two sorts of passive entities are added for items and workstations. How these system elements are created and registered at the world instance is shown in the method `create_entities` in Figure 3.17. For each robot and the coordinator, instances of the class `Agent` are created and assigned with a SALMA-APDL `Procedure` as described in Section 3.3. The content of these procedures are omitted here since the most important parts have already been shown above. However, the full code can be found on the SALMA website at [Kro16]. Besides the agents, `Entity` objects are created to represent the items and workstations in the system. All entity and agent objects are registered at the world instance using the method `World.add()`, which creates preliminary *domain maps* for the sorts of the registered entities. However, before the agents and entities can actually be used in the model, the world instance has to be *initialized* using the method `World.initialize()`. This step in particular sets up the sort domain hierarchy according to the `subsort` relations defined in the domain model, i.e. it makes sure that the domain of a super-sort contains the entities of all its subsorts.

After the entities for the system have been registered and the world instance has been initialized, the `Entity` and `Agent` objects can be accessed either individually by a lookup via their identifiers using the method `World.get_entities_by_id()`, or as part of a set returned by `World.getDomain()` that retrieves all entities of a given sort and its subsorts.

The next step at this point is to set values for constant and fluent instances at the initial situation. One way to do this is to use the methods `World.set_fluent_value(fname, params, value)` and `World.set_constant_value(fname, params, value)` that expect the name of the fluent or constant and the parameters that identify the instances and assigns the given value. However, a much more elegant way is leverage the object-oriented facade that is offered by `Entity` and `Agent` classes (see also Section 3.4.5). This approach has been used in Figure 3.18 where the initial situation is set up for the delivery robots example. There, within a loop over all `Agent` objects from the domain of the sort `robot`, the position of each robot is set to coordinate value sampled from uniform distributions over the world's *grid*, whose dimensions are defined by the constants `GRID_WIDTH` and `GRID_HEIGHT`. The same random initialization is done below for the positions of the items and the stations. In fact, as will be explained in Section 3.4.4, the method `create_initial_situation()` will actually be executed at the begin-

```
def create_robot(num):
   p = Procedure([ ... ])
   proc = OneShotProcess(p)
   return Agent("rob" + str(num), "robot", [proc])

def create_coordinator():
   p = Procedure([ ... ])
   proc = PeriodicProcess(p, 50)
   return Agent("coordinator1", "coordinator", [proc])
      ...

def create_entities(self):
      coordinator1 = create_coordinator()
      self.world.add(coordinator1)
      for r in range(1, NUM_ROBOTS + 1):
         self.world.add(create_robot(r))
      for i in range(1, NUM_ITEMS + 1):
         self.world.add(Entity("item" + str(i), "item"))
      for i in range(1, NUM_STATIONS + 1):
         self.world.add(Entity("ws" + str(i), "workstation"))
```

Figure 3.17: Creation of entities for the delivery robots example.

ning of each *simulation run*, hence creating an experiment where the entity positions are *randomized*. Of course, this is only one possible choice for an *initialization strategy* and others can be realized just as well using the facilities of the SALMA framework. In fact, this is already one important aspect of designing the simulation experiment, which is discussed next.

3.4.4 Defining and Performing a Simulation Experiment

The previous sections of this chapter explained how a simulation model can be defined using the modeling languages of the SALMA approach and how the model can be turned into a concrete scenario by initializing values and choosing probability distributions. In order to use this model within a *simulation experiment*, some additional decisions have to be made, for instance:

1. How often should the simulation be repeated, i.e. how many *simulation runs* are performed?
2. How long should each simulation run be followed? What criteria for cancellation exist?
3. How should parameters of the model be varied between the simulation runs.

```
def create_initial_situation(self):
    coordinator1, = self.world.get_entities_by_id("coordinator1")
    robots = self.world.getDomain("robot")
    items = self.world.getDomain("item")
    workstations = self.world.getDomain("workstation")

    for r in robots:
        r.xpos = np.random.randint(1, GRID_WIDTH)
        r.ypos = np.random.randint(1, GRID_HEIGHT)
        r.vx = 0
        r.vy = 0
        r.broken = False
        r.next_task = None
        r.robot_radius = 1
        for i in items:
            r.set_carrying(i, False)

    coordinator1.request_queue = []

    for item in items:
        item.xpos = np.random.randint(1, GRID_WIDTH)
        item.ypos = np.random.randint(1, GRID_HEIGHT)
        item.delivered_to = None

    for ws in workstations:
        ws.stationX = np.random.randint(1, GRID_WIDTH)
        ws.stationY = np.random.randint(1, GRID_HEIGHT)
        ws.delivered_item_count = 0
```

Figure 3.18: Creation of the initial situation for the delivery robots experiment.

4. Which information should be recorded in order to create an adequate data base for later analysis?

In general, making these choices is part of a process that is often referred to as *simulation experiment design*, which is a wide field by itself and can therefore only be touched very superficially in this thesis. Often cited introductions to this topic can be found in [San05] or [Law14, Chap. 12].

The first aspect to realize with respect to the issues mentioned above is that these decisions have to be made not only for one individual simulation run but for a series of simulation runs. This means that the framework needs a way to control the entire life cycle of a simulation run, including initialization, reset, cleanup, and data logging. The part of the SALMA simulation framework that is responsible for this is shown in Figure 3.19. A simulation experiment is defined by creating a subclass of `Experiment`. There, a protocol for the initialization of a simulation run is established by means of several *template methods* that can be implemented within the user-defined subclass (see [GHJV94]). Primarily, this includes the methods `create_entities`, `setup_distributions`, and `create_initial_situation`, for which examples were shown throughout this section. The structure of the initialization procedure itself can be seen in Figure 3.22a.

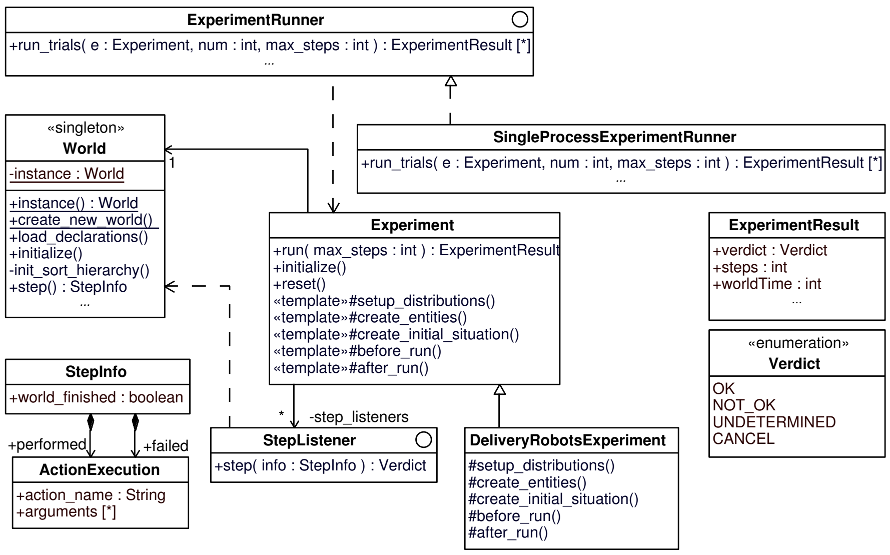


Figure 3.19: Structure of the SALMA simulation experiment framework.

After the initialization sequence, a simulation experiment can be executed using the method `Experiment.run()`. The simulation would then proceed until either the *world has finished*, i.e. there is no process left that might be executed, or when a *time limit* is reached. Additionally, an experiment can be equipped with one or more `StepListeners`, which, although represented as interfaces in Figure 3.19, are really *callback functions* that are executed after each simulation step. Every step listener function receives arguments that include a reference to the `World` instance and a collection of details about the current step. In particular, it receives two lists that contain the actions (including events) that were performed in this step and those that failed because their preconditions were not satisfied. One typical use case for a step listener is to write part of the state and action information to a log-file or a database that can later be used for analysis of the experiment results. An example for such a logging handler that is used in the delivery robots example can be found in Figure 3.20. There, a step listener is created as a *closure* that is bound to a *file object* [Pyt15a, 16.2] that references a CSV file to which the positions of all robots are written (together with other data that is omitted here).

Besides for data logging, step listeners can be used to define stop condi-

```
def create_step_logger(f: TextIOBase):
   def __l(world: World, step=None, **kwargs):
      positions = []
         . . .
      robots = sorted(world.getDomain("robot"))
      for rob in robots:
         positions.append((rob.xpos, rob.ypos))
         . . .
      columns = [step, world.time]
      for p in positions:
         columns.extend(p)
      . . .
      f.write(";".join(list(map(str, columns))) + "\ n")
      f.flush()
   return __l

   . . .
   experiment = DeliveryRobotsExample(experiment_path)
   experiment.initialize()
   with experiment_path.joinpath("experiment.csv").open("w") as f:
      f.write(create_csv_header() + "\ n")
      f.flush()
      experiment.step_listeners.append(create_step_logger(f))
      experiment.run(max_steps=3000)
```

Figure 3.20: Use of a step listener for data logging in the delivery robots experiment.

tions for simulation runs. In fact, a simulation run is stopped when a step listener returns one of the *verdicts* `OK` or `NOT_OK`, declaring the run either as success or as failure. This can be an important measure to avoid simulations from getting stuck in a state in which no further valuable progress is possible. For instance, in the delivery robots example, it might happen that all robots are broken and thus unable to move, or that all items have been delivered. In both cases, it obviously does not make sense to continue the simulation. Therefore, the two additional step listeners shown in Figure 3.21 are installed. The first, `break_when_all_delivered`, returns a positive verdict, when the fluent `delivered_to` is set for all items. On the other hand, `break_when_all_broken` returns a negative verdict when the fluent `broken` is true for all robots. Both return `None` when their conditions are not met to indicate that the simulation should continue.

When a simulation experiment involves any kind of statistical analysis, a single simulation run is not enough for any meaningful analysis. Instead, a *batch* of simulation runs has to be performed to gather a sufficient amount of data. For this purpose, the SALMA framework defines the interface

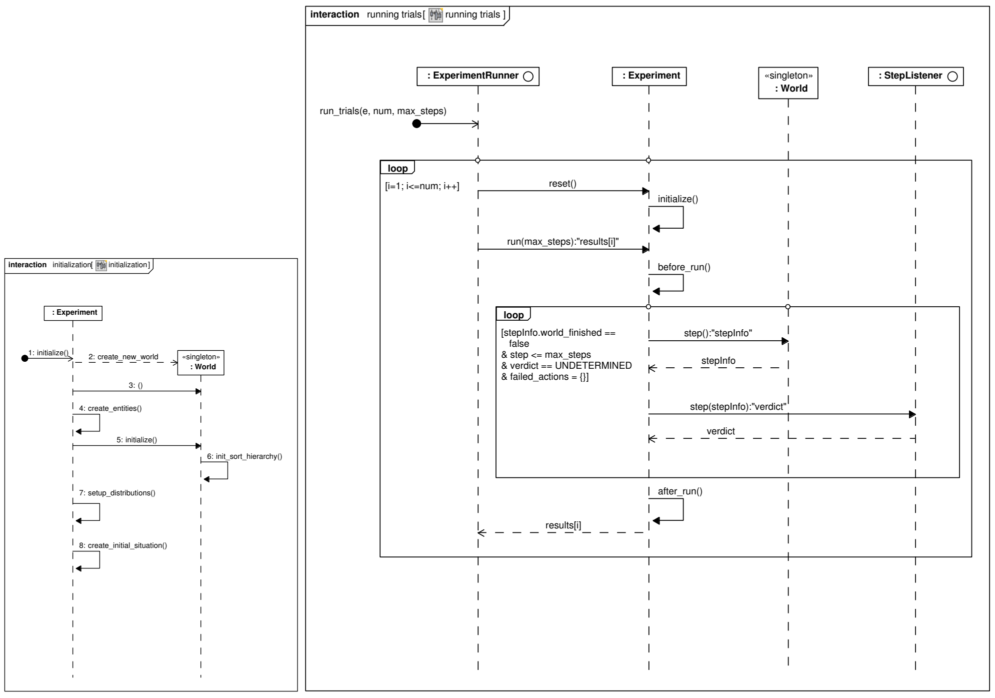
```
def break_when_all_delivered(world: World, **kwargs):
    for i in world.getDomain("item"):
        if i.delivered_to is None:
            return None
    return OK

def break_when_all_broken(world: World, **kwargs):
    for r in world.getDomain("robot"):
        if r.broken is False:
            return None
    return NOT_OK
```

Figure 3.21: Use of step listeners to establish simulation stop conditions in the delivery robots experiment.

`ExperimentRunner` with the method `run_trials()` that can be called with a number of simulation runs (trials) that should be performed. The execution then enters a nested loop that is sketched in Figure 3.22b. It can be seen that the initialization procedure from Figure 3.22a is executed at the beginning of every simulation run. As explained above, this resets the world state by recreating all entities, restoring the event and action configurations, and constructing an initial situation for the next run. Then, the `Experiment` is executed via its `run()` method, which triggers the hook function `before_run()` that can, for example, be used to initialize auxiliary data structures or resources like log-files. Then, the inner loop is entered, which keeps executing the main step function of the simulation algorithm, `World.step()` until a) the simulation indicates a finish of the world's processes (represented by the flag `stepInfo.world_finished` being true); b) the maximum number of steps has been reached; c) the simulation run was stopped because a verdict has been found by a step listener, or d) an action has failed, i.e. it has been performed by an agent although its precondition was not satisfied. Finally, when the inner loop is left, the method `after_run()` is called to perform any necessary post-processing, e.g. saving files to disc, and the results of the simulation run are appended to the overall result collection which will eventually be returned when `run_trials()` is exited.

Although the abstracted interaction in Figure 3.22 refers to the general interface `ExperimentRunner`, it is actually a representation of the execution schema realized in the class `SingleProcessExperimentRunner`. At the moment, this class that performs all simulation runs sequentially within one Python process is the only implementation included in the SALMA framework. However, with the recent development in the fields of parallel and distributed computing, cloud computing, and various emerging "Big Data" technologies,



(a) Simulation run initialization sequence.

(b) Execution of a batch of simulation runs.

Figure 3.22: Interactions during simulation initialization and execution.

it has become a relatively straightforward task to realize experiment runners that are able to simultaneously perform many simulation runs in a cluster and aggregate the results. Since this kind of *horizontal scalability* is a key factor for the success of statistical model checking and simulation approaches in general, this topic will be revisited in the outlook that is given in Section 7.3 in the conclusion of this thesis.

3.4.5 Mechanisms for Language Integration

One of the main goals that guided the design of the SALMA modeling languages and the underlying framework is to create a tight integration between the between the model parts where the situation calculus model can be accessed in a natural way but the modeler can still leverage the Python ecosystem to full extent. As Fowler points out in [Fow10, Chap. 6], this is one of the strongest arguments for the use of an internal *domain specific language (DSL)* like the SALMA-APDL instead of an *external* one that is parsed and interpreted as a closed unit.

In Section 3.3, several points were discussed where the demand for integration of the situation calculus model with Python code arises, namely the definition of conditions and the retrieval of values for variable assignment and iteration. The first obvious option in these cases is to specify the name of the fluent and a list of arguments that are then passed directly to the `LogicEngine`, i.e. the *bridge* to the Prolog interpreter (see Figure 3.14). This works fine when the fluent name itself conveys enough information to explain the meaning of its concrete use on first sight. However, often the condition or value source is more complex and involves calculations or combinations of multiple fluents or constants. For these cases, the SALMA framework allows using Python expressions in which all elements of the situation calculus model are accessible in a natural way. In fact, Section 3.3 already presented many examples for expressions like that, for instance in Figure 3.12 where the precondition of the `deliver` action is tested by an `If` statement with the following condition:

```
If("not self.broken and "
  "dist_from_station(self, targetWs) == 0 and carrying(self, targetItem)", . . .
```

It is obvious that this condition can be read like a regular object-oriented expression and captures both the decision logic and the origin of the involved data in a very concise way. Expressions like this are possible because the SALMA framework populates the *namespace* that is used when the expression is evaluated by Python's built-in function `eval()` [Pyt15b, Chap. 2]. During the initialization phase, it adds *accessor functions* for fluents and constants and *entity objects* that are bound to their identifiers, e.g. `targetWs` and `targetItem` above. Accessor functions, like `carrying()` in the example above, act as direct wrappers around calls to the logic engine where the name of the fluent or constant is given implicitly by the function name. This already allows creating concise expressions that hide much unnecessary clutter. However, the SALMA framework offers another option for accessing fluent and constant values that is even much more concise and clearer to read. A good example can again be taken from Figure 3.12, this time being the condition of the robots' main movement control loop:

```
While("not self.broken and self.next_task != None and "
        "(self.xpos != tx or self.ypos != ty)", [ . . .
```

As demonstrated in this expression, the SALMA framework provides an *object-oriented view* on the domain model. In fact, many fluents and constants are declared with a single parameter whose type is an entity sort, e.g. the fluents `xpos` and `ypos` or the constants `stationX` and `stationY` from the delivery robots example (see Figure 3.3). These fluents and constants can be understood as *attributes* of the entities that are admissible for the qualifying argument. To achieve this, the SALMA framework leverages a mechanism in Python that allows customizing the attribute access of objects (see [Pyt15a, Sec. 3.3.2.]). In particular, the special methods `__get_attr__()` and `__set_attr__()` that are declared in Python's top-level class `object`, are overwritten in the class `Entity`. These methods are called by the Python runtime when undefined attributes are accessed. When this happens, the method implementations in `Entity` delegate the access to the Python-Prolog bridge `LogicEngine` that retrieves or updates the value of the fluent or constant instance as intended. Besides being usable in expressions within SALMA-APDL statements as above, the instances of the class `Entity` are used also in other circumstances, e.g. within custom probability distribution functions (Figure 3.16) or for the creation of the initial situation (Figure 3.18). Altogether, it can be seen from these examples that this style of attribute fluent and constant access effectively achieves a seamless integration of the situation calculus model into the Python environment. On top of that, the object-oriented view provides a perspective on the system model that is likely to be more intuitive for users who are nowadays almost certainly familiar with object-oriented programming languages.

3.5 The Delivery Robots Experiment Revisited

All components of the SALMA simulation approach have now been discussed. This means that the parts of the delivery robots example shown in the previous sections can be assembled to create a fully functional simulation model and define concrete simulation experiments. As a first step, it is important to validate the model, i.e. to confirm that the simulated system behaves in a way that is consistent with the assumptions and expectations of the modeler. In the case of the delivery robots experiment, this can best be done by visualizing the behavior of the robots. The easiest way this can be achieved is by logging the positions of the robots to a CSV file with a step listener like that in Figure 3.20. After the simulation experiment, this file can be read and analyzed with any mathematical software package like Matlab, Octave, or R, or with libraries provided by the scientific Python stack (SciPy) [dev15], which was used for all calculations and diagrams in this thesis. As a first step, the path of the robots can be *plotted* in a two-dimensional diagram together with the locations of items and workstations. Figure 3.23 show such a plot with all robot paths for a simulation with 3 robots, 20 items, and 5 workstations with a grid size of 200×200. All positions were initialized randomly by sampling from uniform distributions on the grid's dimensions as shown in Figure 3.18. Even though it is hard to trace the exact movements of each individual robot, it can clearly be seen that the robots in fact adhere to the movement strategy defined in Figure 3.12 and that all items were visited during simulation. This already can be seen as a strong confirmation for the simulation's validity. For further investigation, it is also possible to animate the robots' movement, which shows that each robot exactly follows the two phase of its control procedure, i.e. picking up the item and then delivering to a workstation.

After the validity of the simulation model has been confirmed visually,

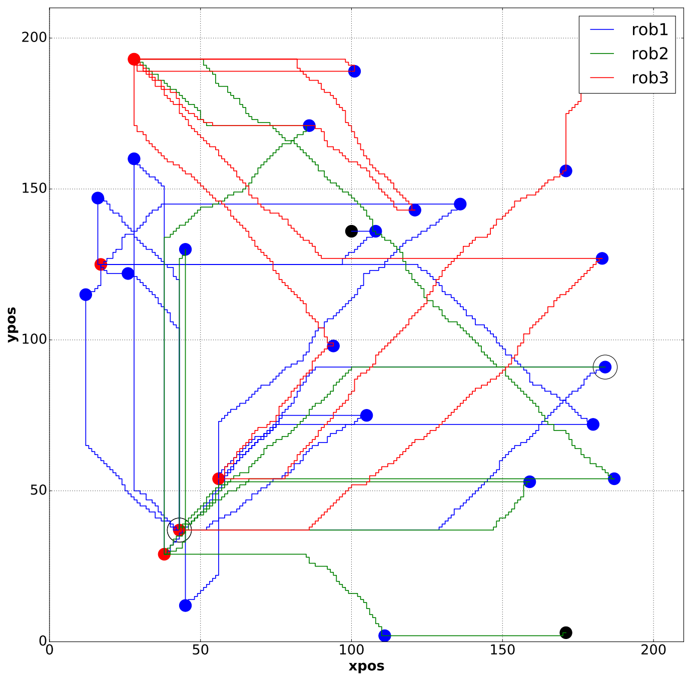


Figure 3.23: Robot paths from a simulation with 3 robots, 20 items, and 5 workstations in a 200 × 200 grid.

experiments can be set up to perform deeper analyses of aspects in which the modeler is interested. One typical question that might arise in a scenario like the delivery robot example is how the number of robots influences the rate of items delivered to workstations. On the one hand, it makes sense to expect that deploying more robots means that items can be delivered faster. This is partly due to higher parallelism and also because the distances that have to be covered between robots, items, and workstation tend to be shorter. However, it should also be expected that there are limiting factors that counteract the positive effect of higher robot numbers from a certain point on. For example, depending on the grid size, very high numbers of robots probably lead to more collisions and hence broken robots that cannot contribute anymore. Therefore, one important question that could be investigated by means of a simulation experiment is to see whether there is in fact a limit for the number of robots at which a congestion appears.

Another significant factor for the efficiency of the delivery robots system is the strategy used by the coordinator to assign delivery tasks. So far, this has been a rather simple one that was already shown in Figure 3.11 and Figure 3.13. With this strategy, the coordinator selects any robot that is not currently assigned to a delivery task and then chooses the closest item to that robot that has not been assigned to another robot, yet. Clearly, the assignments that are created could be rather inefficient when a robot is selected that

```
def select_item2(station: Entity, ctx: EvaluationContext=None, **kwargs):
    dist = (lambda r: np.abs(r.xpos - station.stationX) +
                      np.abs(r.ypos - station.stationY))
    robot_distances = [(dist(r), r) for r in ctx.getDomain("robot") if r.unassigned]
    if len(robot_distances) == 0:
        return None, None
    closest_robot = min(robot_distances)[1]

    dist = (lambda i: np.abs(i.xpos - closest_robot.xpos) +
                      np.abs(i.ypos - closest_robot.ypos))
    item_distances = [(dist(i), i) for i in ctx.getDomain("item") if i.undelivered]
    if len(item_distances) == 0:
        return None, None
    closest_item = min(item_distances)[1]
    return closest_robot, closest_item

...
p = Procedure([
        Iterate("self.request_queue", [ws], [
            Assign([r, i], select_item2, [ws]),
            If("i != None and r != None",
               Act("assign_task", [SELF, r, i, ws]))])
    ])
```

Figure 3.24: Improved strategy for delivery task assignment.

is far away from the requesting workstation. A first possible improvement of the strategy might therefore be to select the unassigned robot that is closest to the workstation. This is shown in Figure 3.24 where the selection is done inside a Python function that uses a very compact style leveraging Python's list comprehensions and *lambda functions* for calculating the Manhattan distances (see [Bla06]) between robots and stations and between robots and items, respectively. Since the new selection function returns robot and item together as a pair, the `Select` statement used before in the coordinator's control procedure (Figure 3.13) is removed and the result of the select function is assigned to the two variables `r` and `i` at once.

In order to compare the performance of the two proposed strategies and to analyze the effects of the robot number, the experiment can be modified so that the number of robots and the strategy are used as controlled variables and the mean number of items delivered within a certain period is treated as the measured result. This is realized by the three nested loops shown in Figure 3.25. It can be seen that the number of robots is increased in steps of 5 from 5 to 100 (since the second argument for Python's range is exclusive),

```
for num_robots in range(5, 105, 5):
  for strategy in [1, 2]:
    for i in range(10):
      if experiment_path.joinpath("stop.txt").exists():
        break
      else:
        experiment = Experiment02(num_robots, strategy)
        experiment.initialize()
        experiment.run(max_steps=500)
```

Figure 3.25: Experiment control loop for comparing delivery task selection strategies.

and for each number, 10 independent simulations are performed for the first strategy, and 10 for the second. What is not shown here is that instead of recording data for each simulation step as in the first experiment, this time a summary of each simulation run is logged, containing most importantly the number of robots and the number of items that could be delivered within the given period of 500 simulation steps. This is done by overwriting the life-cycle handler function `after_run` that the class `Experiment02` inherits from the `Experiment` base class.

The loop in Figure 3.25 conducts $20 \times 2 \times 10 = 400$ independent simulations and writes the summary for each of them as one line to a comma separated values (CSV) file. Due to the relatively high complexity of the logical progression performed in each step, the runtime of the simulations is significantly higher than it would be for an approach with a lower abstraction level. In fact, the time complexity of the progression step obviously grows approximately linearly in the number of fluents and entities in the system. For a configuration with 50 robots, 10 workstations, and 100 items, this can mean durations of about two minutes for each simulation on a modern average desktop computer. However, since the simulations are independent, it is possible to run them in parallel, either on multiple cores on the same system of on a cluster of multiple machines. The recorded results from all simulations can then simply be merged in order to achieve more precise analysis results. In fact, for the results shown below, the inner loop from Figure 3.25 was actually split and the simulations were distributed among 10 virtual machines on a cloud platform each of which conducted 2 simulations for each number of robots (one for each strategy). For each simulation, a grid of 500×500 units was used that was populated with 10 workstations and 100 items with randomized initial positions. The probability p_{slot} that any workstation slot generates a request in the current time step was set to 0.01 (see 3.1) in Section 3.4.2) and the number of slots per workstation was set to 100. This guarantee that there are always enough free

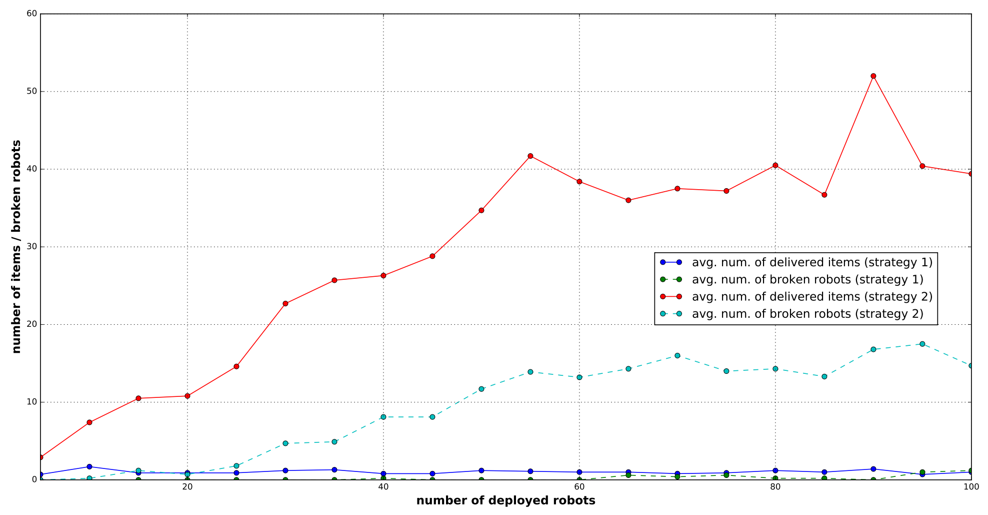


Figure 3.26: Mean count of delivered items and collisions by number of robots.

slots to generate tasks for all robots. Figure 3.26 shows the mean numbers of delivered items and the mean number of robots that were broken due to collisions, within 500 simulation steps for each combination of a number of robots and the chosen strategy. Each data point in the plot is a mean calculated from the results of 10 simulation runs. The plot clearly shows the expected advantage of the second strategy. It also confirms the anticipated congestion effect since a number of robots higher than 55 does not appear to improve the overall delivery performance significantly.

3.6 SALMA Simulation Semantics

So far, the SALMA simulation approach has been described from a detailed yet practical perspective. All important elements of the SALMA modeling languages have been introduced with respect to their syntax and their role in the simulation model. Together with the examples within this chapter, it should already be possible to get a clear picture about how simulation in SALMA works. To render this first understanding more precisely, this section provides a formal operational semantics of SALMA's simulation mechanism. The structure of the semantics is chosen so that it is on the one hand very close to the actual simulation algorithm but on the other hand abstract enough to allow focusing on the most important aspects.

3.6.1 Basic Definitions

As the top-level element of the simulation semantics described here, the *system simulation model* is a combination of all declarations together with the world state.

Definition 3.19 (System Simulation Model)**.** The system simulation model is defined by the following tuple:

$$Sys = \langle Decl, \mathcal{D}^{Dom}, Sched, Agents, (Procs_a)_{a \in Agents}, Prob, \mathcal{F}^{S_0} \rangle \tag{3.3}$$

Here, *Decl* is the set of all declaration statements for sort, sort hierarchies, fluents, and primitive, stochastic and exogenous actions. In conjunction with this, $\mathcal{D}^{Dom}$ denotes the *basic action theory* [Rei01], which is defined as the complete set of successor state and precondition axioms that define when and how the system can progress in response to actions and events. In addition to this classical part, the domain model in SALMA also contains schedulability axioms, represented by the set *Sched*, that define preconditions that have to be met so that events can be scheduled. Furthermore, $(Procs_a)_{a \in Agents}$ is an indexed family of process definitions that define the agents' behavior, and *Prob* stands for the set of probability distributions that are used by the simulation to schedule events and to choose probabilistic action outcomes. Finally, $\mathcal{F}_{S_0}$ is the set of fluent instance values defined for the *initial situation*, i.e. the set of values for fluent instances that is used at the start of the simulation run.

Based on the system model, it is possible to define the *system state*, which is a combination of several structures that are manipulated during simulation.

Definition 3.20 (System state)**.** The state of the simulated system is defined as follows:

$$St = \langle P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F} \rangle \tag{3.4}$$

Here, P_{run}, P_{act}, P_{wait}, and P_{idle} are sets of *process states* (see below) that describe the processes which are currently being executed, performing actions, waiting, or idle, respectively. *Act* is the set of pending actions that are yet to be executed in the current simulation step, *Evt* is the event schedule, and $\mathcal{F}$ represents the current situation, i.e. the current set of values for all fluent instances of the system.

The life-cycle status of a process is represented by its membership to one of the sets P_{run}, P_{act}, P_{wait}, and P_{idle}. Clearly, a process can at any point in time either be running, acting waiting, or idle. Therefore, the following constraint holds:

$$\begin{aligned} P_{run} \cap P_{act} = \emptyset \wedge P_{run} \cap P_{wait} = \emptyset \wedge P_{run} \cap P_{idle} = \emptyset \\ \wedge P_{act} \cap P_{wait} = \emptyset \wedge P_{act} \cap P_{idle} = \emptyset \wedge P_{wait} \cap P_{idle} = \emptyset \end{aligned} \tag{3.5}$$

The process state descriptions mentioned above combine all information about the current state of each process.

Definition 3.21 (Process state)**.** The state of a process p in state St, denoted as p^{St}, is defined by a tuple of the following form:

$$p^{St} = (pid, a, n_{cur} \circ \sigma, \eta) \tag{3.6}$$

Here, *pid* is a process identifier, a is the agent that executes the process and n_{cur} is the current *process control node*, i.e. the position in the procedure of the process that was reached last. Besides that, the suffix of the process, i.e. the sequence of remaining statements of the procedure that will be executed after n_{cur}, is denoted by σ. The operator $\circ$ is used in this definition and below to represent either sequence composition or the addition of an element at the start or end of a sequence. Finally, η is the process-local evaluation context that defines the mappings of variables to values.

At the *initial state* of a simulation, the *initial situation* S_0 holds, i.e. all fluent instances are set to their initial values. Additionally, all processes are in the *idle* state and neither actions nor events are scheduled.

Definition 3.22 (Initial system state)**.** Let S_0 be the initial situation of the system in the sense of the situation calculus. Furthermore, let PBody_{pid} be the full control node sequence of the procedure declaration of the process with the process identifier *pid*. Then the initial state of the simulation is given by

$$St^{S_0} = \langle P^{S_0}_{run}, P^{S_0}_{act}, P^{S_0}_{wait}, P^{S_0}_{idle}, Act^{S_0}, Evt^{S_0}, \mathcal{F}^{S_0} \rangle \tag{3.7}$$

where

$$\begin{aligned} P^{S_0}_{run} =& P^{S_0}_{act} = P^{S_0}_{wait} = Act^{S_0} = Evt^{S_0} = \emptyset \\ P^{S_0}_{idle} =& \{(pid, a, \mathsf{PBody}_{pid}, \emptyset) \mid a \in Agents \\ & \wedge (pid, \mathsf{PBody}_{pid}) \in Procs_a\} \end{aligned} \tag{3.8}$$

3.6.2 Core Simulation Semantics

In the following, the simulation semantics of SALMA will be described by means of transition rules that define how the system state can evolve during simulation. These rules are written in a style that is inspired by structural operational semantics (see [Plo04]). However, the premises for the applicability of each rule are mostly not stated as explicit preconditions but implicitly by patterns in the respective data structures that have to be matched when the rule “fires”.

At first, there is the *Act* statement with which an agent can execute *actions*. In fact, this is the only option for an agent to influence its environment – namely through the effect of the executed actions in the sense of the situation calculus. However, the progression is not performed directly for each *Act* call. Instead, the current interpretation of the action term is added to the set of *pending actions*. At the same time, the process is suspended temporarily until the action has been handled.

Definition 3.23 (Action execution)**.** Let α be an action term that possibly contains variables. Then, an action execution is interpreted as follows:

$$\begin{aligned}&\langle\{(pid, a_s, \mathsf{Act}(\alpha)\circ\sigma, \eta)\}\cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle \longrightarrow\\ &\langle P_{run}, P_{act}\cup\{(pid, a_s, \sigma, \eta)\}, P_{wait}, P_{idle}, Act\cup\{[\![\alpha]\!]_{\mathcal{F},\eta}\}, Evt, \mathcal{F}\rangle\end{aligned}\tag{3.9}$$

Concordant with the postponed execution mentioned in the last definition, the system performs *progression* steps for the executed actions only when all active processes are currently blocked, i.e. they are either waiting or performing actions. In this case, both pending actions and events that are due for the current time step are performed in random order.

Definition 3.24 (Action progression)**.** Let α and ϵ be ground terms that denote a valid concrete action or event, respectively. Furthermore, let $t = time(S)$ be the current time and $\mathsf{progress}(\mathcal{F}, \alpha)$ the fluent database that results from performing a progression step for action α to the fluent database in the current simulation step, $\mathcal{F}$ (see [Rei01, Chap. 9]). Then, the following rule describes the premise and effect of a progression step:

a) *Actions*:

$$\begin{aligned}&\langle\emptyset, P_{act}, P_{wait}, P_{idle}, \{\alpha\}\cup Act, Evt, \mathcal{F}\rangle \longrightarrow\\ &\langle\emptyset, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}'\rangle\end{aligned}\tag{3.10}$$

$$\text{where } \mathcal{F}' = \begin{cases}\mathsf{progress}(\mathcal{F}, \alpha) & \text{if } [\![poss(\alpha)]\!]_{\mathcal{F}}\\ \mathcal{F} & \text{otherwise}\end{cases}$$

b) *Events*:

$$\begin{aligned}&\langle \emptyset, P_{act}, P_{wait}, P_{idle}, Act, \{(\epsilon, t)\} \cup Evt, \mathcal{F}\rangle \longrightarrow\\ &\langle \emptyset, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}'\rangle\end{aligned} \tag{3.11}$$

$$\text{where } \mathcal{F}' = \begin{cases}\mathsf{progress}(\mathcal{F}, \epsilon) & \text{if } [\![poss(\epsilon)]\!]_{\mathcal{F}}\\ \mathcal{F} & \text{otherwise}\end{cases}$$

After all currently scheduled agent actions have been performed, idle due processes are activated and blocked processes are reactivated. This first includes all processes that are currently executing actions. Additionally, waiting and idle processes can be scheduled in this phase when the corresponding conditions are satisfied.

Definition 3.25 (Process activation)**.** Let *cond*(p) denote a condition on which the process p is waiting after executing a `Wait` statement, and let *trigger*(p) be a condition that, if true, causes an idle *triggered process* to be executed in the current step. Similarly, let *nextScheduleTime*(p) be a function that yields the next time point when an idle periodic process is scheduled to be launched. Then, the following rule describes the activation of processes:

$$\langle \emptyset, P_{act}, P_{wait}, P_{idle}, \emptyset, Evt, \mathcal{F}\rangle \longrightarrow \langle P_{run}, \emptyset, P'_{wait}, P'_{idle}, \emptyset, Evt, \mathcal{F}\rangle \tag{3.12}$$

$$\begin{aligned}\text{where}\quad P_w^+ &= \{p_w \mid p_w \in P_{wait} \wedge [\![cond(p_w)]\!]_{\mathcal{F}} = \top\}\\ P_i^+ &= \{p_i \mid p_i \in P_{idle} \wedge ([\![trigger(p_i)]\!]_{\mathcal{F}} = \top \vee\\ &\qquad [\![time]\!]_{\mathcal{F}} = nextScheduleTime(p_i))\}\\ P_{run} &= P_{act} \cup P_w^+ \cup P_i^+\\ P'_{wait} &= P_{wait} \setminus P_w^+\\ P'_{idle} &= P_{idle} \setminus P_i^+\end{aligned}$$

In the definition above it can be seen that both processes that are *idle* and those that are *waiting* can be reactivated. The waiting state is reached as an explicit consequence of a `Wait` statement, which will be explained later in Definition 3.33. In contrast, a process becomes *idle* simply when all statements of the process body have completed. This is expressed by the following rule.

Definition 3.26 (Process completion)**.** Let *pid* be the id of a running process that has completed its statement sequence in the current simulation step and let PBody_{pid} be the body that is defined for the process in the agent behavior model. A process becomes idle when all statements of its body have completed, i.e.

$$\begin{aligned}&\langle\{(pid, a_s, \emptyset, \eta)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle \longrightarrow \\ &\quad\langle P_{run}, P_{act}, P_{wait}, \{(pid, a_s, \mathsf{PBody}_{pid}, \emptyset)\} \cup P_{idle}, Act, Evt, \mathcal{F}\rangle\end{aligned} \tag{3.13}$$

It was already mentioned before, e.g. in Section 3.4.2, that an event can be added to the event schedule either a) *instantaneously*, i.e. within the current simulation step, or b) *anticipatory*, i.e. at a time point in the future that is chosen according to a specific probability distribution. Instantaneous scheduling is used when a `poss` axiom but no `schedulable` axiom exists for the event type. Then, that poss axiom is tested for every time step and if it is found to be true, the occurrence distribution assigned to the event is used to decide whether or not the event should occur. For anticipatory scheduling, a `schedulable` axiom has to exist, which is also evaluated at each time step. When this yields a positive result, it means that at this point it is possible to determine a point in the future at which the event should occur. This is done by sampling the delay until the occurrence of the event instance from the assigned occurrence distribution.

In general, scheduling is performed iteratively until no further events can be scheduled, although every concrete event instance can only be scheduled once. Additionally, there may be cases where there can be only one of a set of several events - i.e. they form an *exogenous action choice* (see Section 3.2.3).

Definition 3.27 (Event scheduling)**.** Let ϵ be an event term, ΔT_ϵ a random variable that models a delay for the event ϵ, and $Occur_\epsilon$ a random variable that models whether ϵ should occur ($Occur = 1$) or not ($Occur = 0$) in the current simulation step. Furthermore, let the predicates $schedulable(\epsilon)$ and $poss(\epsilon)$ represent the tests of the schedulability and possibility axioms with the concrete event instance ϵ in the current simulation step. Whether an event is scheduled instantaneously or anticipatory in the sense explained above is determined by the function $type(\epsilon)$. Furthermore, the predicate $exclusive(\epsilon_1, \epsilon_2)$ indicates that two events are mutually exclusive as part of an *exogenous action choice*. Finally, the notation $\xrightarrow{p}$ is used to express that the given state transition occurs with probability p. Finally, as above, $\mathcal{F}$ is used to represent the fluent database in the current simulation step.

Given the definitions above, assume that Φ_0 and one of the two conditions Φ_1 or Φ_2 are satisfied:

$$\begin{aligned}
&\Phi_0 \equiv (\nexists t'.\, (\epsilon, t') \in \mathit{Evt}) \wedge (\nexists t'' \nexists \epsilon'.\, \mathit{exclusive}(\epsilon, \epsilon') \wedge (\epsilon', t'') \in \mathit{Evt}) \\
&\Phi_1 \equiv \mathit{type}(\epsilon) = \mathit{immediate} \wedge [\![\mathit{poss}(\epsilon)]\!]_{\mathcal{F}} \\
&\Phi_2 \equiv \mathit{type}(\epsilon) = \mathit{scheduled} \wedge [\![\mathit{sched}(\epsilon)]\!]_{\mathcal{F}}
\end{aligned} \tag{3.14}$$

Then,

$$\begin{aligned}
&\langle \emptyset, P_{act}, P_{wait}, P_{idle}, \mathit{Act}, \mathit{Evt}, \mathcal{F} \rangle \xrightarrow{p} \\
&\langle \emptyset, P_{act}, P_{wait}, P_{idle}, \mathit{Act}, \{(\epsilon, t)\} \cup \mathit{Evt}, \mathcal{F} \rangle
\end{aligned} \tag{3.15}$$

and

$$\begin{aligned}
&\langle \emptyset, P_{act}, P_{wait}, P_{idle}, \mathit{Act}, \mathit{Evt}, \mathcal{F} \rangle \xrightarrow{1-p} \\
&\langle \emptyset, P_{act}, P_{wait}, P_{idle}, \mathit{Act}, \mathit{Evt}, \mathcal{F} \rangle
\end{aligned} \tag{3.16}$$

where

$$p = \begin{cases} Pr(\mathit{Occur}_\epsilon = 1) & \text{if } t = [\![\mathit{time}]\!]_{\mathcal{F}} \wedge \Phi_1 \\ Pr(\Delta T_\epsilon = \delta t) & \text{if } t = [\![\mathit{time}]\!]_{\mathcal{F}} + \delta t \wedge \Phi_2 \\ 0 & \text{otherwise} \end{cases} \tag{3.17}$$

When no agent processes can proceed any further and no more actions or events can be executed in the current simulation step, the system progresses to the next time point that is relevant either a) because an event is scheduled for this time, b) because a periodic process is scheduled to be executed, or c) because an event will become possible or schedulable at this time.

Definition 3.28 (Time advance)**.** Let $\mathcal{F}$ be the fluent database in the current simulation step and S_{cur} the associated current *situation.* As usual in the situation calculus literature, $S_1 \subset S_2$ is used to express that situation S_2 results from performing a sequence of actions in S_1. Also, let *EligibleEvent* be a *derived fluent* that is defined as below:

$$\begin{aligned}
\mathit{EligibleEvent}(\epsilon, S) \equiv\ & (\mathit{type}(\epsilon) = \mathit{immediate} \wedge \mathit{poss}(\epsilon, S)) \\
\vee\ & (\mathit{type}(\epsilon) = \mathit{scheduled} \wedge \mathit{sched}(\epsilon, S))
\end{aligned} \tag{3.18}$$

EligibleEvent determines whether an event instance can happen in a given situation or be scheduled for later occurrence. Based on that, *StepEnd* is true if no further processing is possible in the current simulation step.

$$\begin{aligned} \mathit{StepEnd} \equiv & (\nexists p_w \in P_{wait}.\ [\![\mathit{cond}(p_w)]\!]_S = \top) \\ & \wedge\ (\nexists p_i \in P_{idle}.\ [\![\mathit{trigger}(p_i)]\!]_S = \top) \\ & \wedge\ \nexists \epsilon.\ \mathit{EligibleEvent}(\epsilon) \end{aligned} \tag{3.19}$$

Furthermore, let $t_{cur} = [\![\mathit{time}]\!]_{\mathcal{F}}$ be the time at the current time step and $\mathit{nextScheduleTime}(p)$ the next time at which a periodic process p is scheduled to be launched.

If *StepEnd* holds in the current system state, the following rule is applicable:

$$\begin{aligned} & \langle \emptyset, \emptyset, P_{wait}, P_{idle}, \emptyset, Evt, \mathcal{F} \rangle \longrightarrow && \text{if } \mathit{StepEnd} = \top \\ & \quad \langle \emptyset, \emptyset, P_{wait}, P_{idle}, \emptyset, Evt, \mathcal{F}' \rangle && \end{aligned} \tag{3.20}$$

$$\text{where } \mathcal{F}' = \mathsf{progress}(\mathcal{F}, \mathit{tick}(t_{next} - t_{cur})) \tag{3.21}$$

$$t_{next} = \mathit{min}\{t_{ev}, t_{wait}, t_{period}, t_{scan}\} \tag{3.22}$$

$$t_{ev} = \mathit{min}\{t \mid (\epsilon, t) \in Evt\} \tag{3.23}$$

$$t_{wait} = \begin{cases} \infty & \text{if } P_{wait} = \emptyset \\ t_{cur} + 1 & \text{if } P_{wait} \neq \emptyset \end{cases} \tag{3.24}$$

$$t_{period} = \mathit{min}\{t \mid \exists p \in P_{idle}.\ t = \mathit{nextScheduleTime}(p)\} \tag{3.25}$$

$$\begin{aligned} t_{scan} = \mathit{min}\{t \mid \exists S, \epsilon.\ & \mathit{EligibleEvent}(\epsilon, S) \wedge \\ & t = \mathit{time}(S) \wedge S_{cur} \subset S\} \end{aligned} \tag{3.26}$$

In the definition above, a situation calculus perspective was chosen to describe the interpretation of fluents, preconditions, and schedulability predicates. In particular, this allows evaluating `EligibleEvent` for future steps that follow the current situation. In the calculation for t_{scan} this is used to search for future situations in which any event becomes possible or schedulable. In fact, as mentioned earlier in Section 3.2, the use of effect axioms in the domain model allows the simulation algorithm to detect if a fluent instance is affected by the *tick* event. This can also be used to determine whether the formula in a *poss* or *schedulable* axiom is time-dependent. By excluding time-independent event instances from consideration in the scanning process, the algorithm can avoid unnecessary computation.

The definition of t_{wait} in (3.24) effectively says that as soon as at least one process is in the waiting state, the next time step has to be visited, i.e. the time cannot be advanced further that $t_{cur}+1$. This is due to the fact that in general, the conditions of waiting processes could depend on events other than *tick*, whose occurrences cannot be predicted. However, as a future extension, it would be worthwhile to integrate a mechanism that recognizes conditions which only depend on time – similar to the mechanism that already exists for exogenous actions (events).

The most obvious effect of the progression step $\mathsf{progress}(\mathcal{F}, tick(t_{next} - t_{cur}))$ is that the world *time* is *advanced* to the next "interesting" point for which an event or a process is scheduled. In fact, the time steps in between the current simulation step and this next one are not simulated explicitly. Depending on the model, this can be much more efficient than approaches that are fixed to equidistant time steps. However, it is possible that the model contains effect axioms that make the event $tick(\Delta T)$ affect not only *time* but also other fluents. In this case, it has to be guaranteed that progressing the model in one step to t_{next} actually has the same effect as advancing every time step separately or as any other time partitioning scheme. This is expressed as a property that will be called *time-advance stability* in this thesis.

Definition 3.29 (Time-Advance-Stability)**.** Let $F(x_1,\ldots,x_n,S)$ be a functional fluent and $G(x_1,\ldots,x_n,S)$ a relational fluent defined in the system domain model. Additionally, let S denote a situation term and $\hat{A}_S = (a_{i,S})_{i=1}^{l}$ the action sequence that leads from the initial situation S_0 to S. Furthermore, let $\mathcal{S}_{ta}$ be defined as the set of possible situation terms that consist only of *tick* actions, i.e.

$$\mathcal{S}_{ta} = \{S \mid \forall i \in [1,l].\, \exists t.\, a_{i,S} = tick(t)\} \tag{3.27}$$

Then F is called *time-advance-stable* if the following property holds:

$$\begin{aligned}&\forall S \in \mathcal{S}_{ta}.\,\forall \Delta t \in \mathbb{N}_0.\forall x_1,\ldots,\forall x_n.\\ &\quad time(S) = time(S_0) + \Delta t \implies\\ &\qquad F(x_1,\ldots,x_n,S) = F(x_1,\ldots,x_n,do(tick(\Delta t),S_0))\end{aligned}$$

Similarly, G is *time-advance-stable iff*

$$\begin{aligned}&\forall S \in \mathcal{S}_{ta}.\,\forall \Delta t \in \mathbb{N}_0.\forall x_1,\ldots,\forall x_n.\\ &\quad time(S) = time(S_0) + \Delta t \implies\\ &\qquad G(x_1,\ldots,x_n,S) \equiv G(x_1,\ldots,x_n,do(tick(\Delta t),S_0))\end{aligned}$$

Finally, a system model *Sys* is called time-advance-stable *iff* all fluents defined in *Sys* are time-advance-stable.

Time-advance-stability requires that all fluents are defined in a way so that it does not matter whether the time is advanced by multiple *tick*-steps that add up to the intended delay, or whether this happens with a single *tick*-action. This directly leads to the following theorem:

Theorem 3.1. *Let Sys be simulation model and let $\mathcal{F}$ be the state of the fluent instance database in a simulation step of Sys. Let furthermore t_{next} and t_{cur} be the times of the current and the next scheduled simulation step, respectively. Furthermore, let $(\delta t_i)_{i=1}^{n}$ denote a sequence of time differences and let $\mathcal{F}_1 = \mathcal{F}_2$ be an abbreviation for the fact that the value of each fluent instance in $\mathcal{F}_1$ is equal to the value of the corresponding fluent instance in $\mathcal{F}_2$. Finally, the progression operator is recursively lifted to the application of a sequence of time steps:*

$$\begin{aligned}\mathsf{progress}(\mathcal{F}, (\delta t_i)_{i=1}^{1}) &= \mathsf{progress}(\mathcal{F}, \mathit{tick}(\delta t_i))\\ \mathsf{progress}(\mathcal{F}, (\delta t_i)_{i=1}^{n}) &= \mathsf{progress}(\mathsf{progress}(\mathcal{F}, (\delta t_i)_{i=1}^{n-1}), \mathit{tick}(\delta t_n))\end{aligned}$$

Given the definitions above, it holds that if Sys is time-advance stable, then all possible sequences of tick actions that in sum lead to the same time advance have the same effect on the model, i.e.

$$\begin{aligned}&\forall \Delta T \forall\, (\delta t_i)_{i=1}^{n} \,.\, \sum_{1}^{n} \delta t_i = \Delta T \implies\\ &\qquad \mathsf{progress}(\mathcal{F}, (\delta t_i)_{i=1}^{n}) = \mathsf{progress}(\mathcal{F}, \mathit{tick}(\Delta T))\end{aligned}$$

Proof. It is shown in [Rei01, Chap. 9.2.1] that for SALMA's fluent database, which is *logically complete* in the sense that it stores concrete values for all fluent instances, a progression step can be realized by substitution with the right sides of the successor state axioms, i.e. by a one-step regression. In fact, this is exactly how progression is realized in the SALMA simulation engine. This implies that if S_0 denotes the current situation, then the sequence $(S_i)_{1\leq i\leq n}$ can be defined recursively as follows:

$$S_i = do(tick(\delta t_i), S_{i-1}) \tag{3.28}$$

This means that the situation S_n, which corresponds to the situation that results from performing $\mathsf{progress}(\mathcal{F}, (\delta t_i)_{i=1}^{n})$, will have the following structure:

$$S_n = do(tick(\delta t_n), do(tick(\delta t_{n-1}), \dots do(tick(\delta t_1), S_0) \dots) \tag{3.29}$$

Since S_n contains only *tick* actions, it belongs to the set $\mathcal{S}_{ta}$ that was defined in Definition 3.28. Therefore, since the system model is time-advance stable, it holds for any functional fluent F and any relational fluent G that

$$F(x_1, \ldots, x_n, S_n) = F(x_1, \ldots, x_n, do(tick(\Delta T), S_0)) \tag{3.30}$$
$$G(x_1, \ldots, x_n, S_n) \equiv G(x_1, \ldots, x_n, do(tick(\Delta T), S_0)) \tag{3.31}$$

By the definition of the progression operator from above, this implies

$$\mathsf{progress}(\mathcal{F}, (\delta t_i)_{i=1}^{n}) = \mathsf{progress}(\mathcal{F}, tick(\Delta T)) \tag{3.32}$$

Therefore, the theorem holds. □

3.6.3 Semantics of Other SALMA-APDL Elements

The rules presented so far described the core simulation mechanism. In the following, this will be augmented by rules for the remaining elements of the SALMA-APDL. First, and arguably one of the most basic ingredient of any non-trivial procedure, is the assignment of a value to a variable, which modifies the evaluation context η.

Definition 3.30 (Variable Assignment)**.** Let x be a variable name and θ be a value source expression as described in Definition 3.12 that can be evaluated within the current evaluation context η in the current situation S. Then the assignment of the value of θ to x is interpreted as follows:

$$\begin{aligned}&\langle\{(pid, a_s, \mathsf{Assign}(x, \theta) \circ \sigma, \eta)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle \longrightarrow\\&\langle\{(pid, a_s, \sigma, \eta')\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle\\&\text{where } \eta' = \eta[x \mapsto [\![\theta]\!]_{\mathcal{F},\eta}]\end{aligned} \tag{3.33}$$

Here, the interpretation of the value source $[\![\theta]\!]_{S,\eta]}$ depends on the source type of θ:

a) If $\theta = f(v_1, \ldots, v_n)$ where f is a fluent, constant, or situation-independent Prolog function, then θ is evaluated by the logic engine with respect to the current situation.

b) If $\theta = g(v_1, \ldots, v_n)$ where g is a Python function, or θ is a string that contains a Python expression, then θ is evaluated within Python using the bindings described in Section 3.4.5.

The conditional statements are interpreted as expected.

Definition 3.31 (Conditional Statements)**.** Let θ be a conditional expression as described in Definition 3.13 and let ς_{then} and ς_{else} be statements. Then, an `If` statement is interpreted as follows:

$$\langle\{(pid, a_s, \mathsf{If}(\theta, \varsigma_{then}, \varsigma_{else}) \circ \sigma, \eta)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle$$

$$\rightarrow \begin{cases} \langle\{(pid, a_s, \varsigma_{then} \circ \sigma, \eta)\} \cup P_{run}, & \text{if } [\![\theta]\!]_{\mathcal{F},\eta} = \top \\ \quad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle & \\ \langle\{(pid, a_s, \varsigma_{else} \circ \sigma, \eta)\} \cup P_{run}, & \text{otherwise} \\ \quad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle & \end{cases}$$

Here, it is assumed for the sake of brevity that ς_{else} can also be empty if the `If` statement does not state an alternative branch.

Similarly, let $\theta_1, \dots, \theta_n$ be conditional expressions and $\varsigma_1, \dots, \varsigma_n$ be statements. Then, the interpretation of the `Switch` statement can be defined as follows:

$$\langle\{(pid, a_s, \mathsf{Switch}(\mathsf{Case}(\theta_1, \varsigma_1), \dots, \mathsf{Case}(\theta_n, \varsigma_n), \mathsf{Default}(\varsigma_{def})) \circ \sigma, \eta)\} \\ \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle$$

$$\rightarrow \begin{cases} \langle\{(pid, a_s, \varsigma_i \circ \sigma, \eta)\} \cup P_{run}, & \text{if } [\![\theta_i]\!]_{\mathcal{F},\eta} = \top \wedge \\ \quad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle & \quad \forall 1 \leq j < i. [\![\theta_j]\!]_{\mathcal{F},\eta} = \bot \\ & \\ \langle\{(pid, a_s, \varsigma_{def} \circ \sigma, \eta)\} \cup P_{run}, & \text{if } \nexists i. [\![\theta_i]\!]_{\mathcal{F},\eta} = \top \\ \quad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle & \end{cases}$$

As before, the omission of the default case is seen as an abbreviation for a `Default` clause with empty body.

As usual in operational semantics, while loops are interpreted by unfolding, i.e. by inserting the body of the `While` block before the loop if the condition evaluates to true.

Definition 3.32 (While loops)**.** Let θ be a conditional expression as described in Definition 3.13 and let ς be a statement. Then a `While` loop with the body ς and the condition θ is interpreted as follows:

$$
\begin{aligned}
&\langle\{(pid, a_s, \mathsf{While}(\theta,\varsigma)\circ\sigma,\eta)\}\\
&\qquad\qquad \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle\\
&\to \begin{cases}
\langle\{(pid, a_s, \varsigma\circ\mathsf{While}(\theta,\varsigma)\circ\sigma,\eta)\}\cup P_{run}, & \text{if } [\![\theta]\!]_{\mathcal{F},\eta} = \top\\
\qquad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle & \\
\langle\{(pid, a_s, \sigma,\eta)\}\cup P_{run}, & \text{if } [\![\theta]\!]_{\mathcal{F},\eta} = \bot\\
\qquad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle &
\end{cases}
\end{aligned}
$$

Another essential element that already appeared implicitly in Definition 3.25 is the `Wait` statement. It moves a running process to the set of waiting processes P_{wait} if the condition is false.

Definition 3.33 (Wait statements)**.** Let θ be a conditional expression. A `Wait` statement that watches the condition θ is interpreted by the following rule:

$$
\begin{aligned}
&\langle\{(pid, a_s, \mathsf{Wait}(\theta)\circ\sigma,\eta)\}\cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle\\
&\to \begin{cases}
\langle\{(pid, a_s, \sigma,\eta)\}\cup P_{run}, & \text{if } [\![\theta]\!]_{\mathcal{F},\eta} = \top\\
\qquad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle & \\
\langle P_{run}, P_{act}, \{(pid, a_s, \mathsf{Wait}(\theta)\circ\sigma,\eta)\}\cup P_{wait}, & \text{if } [\![\theta]\!]_{\mathcal{F},\eta} = \bot\\
\qquad P_{idle}, Act, Evt, \mathcal{F}\rangle &
\end{cases}
\end{aligned}
$$

The semantics for the final two statements of the SALMA-APDL that have not been covered yet in this section, namely `Select` and `Iterate`, would be quite complex to formalize completely. Therefore, the details of the actual result selection are abstracted away and the following rules focus on the consequences to the simulation state.

Definition 3.34 (Selection and iteration)**.** Let P be a Prolog predicate or a relational fluent of arity n and let $x_1, \ldots, x_n$ be a sequence of terms that are either literal values, bound variables, or free variables, i.e. variables that are not yet bound. In this respect, let $I_{FV}(x_1, \ldots, x_n)$ denote the set of indexes between $1, \ldots, n$ that belong to the free variables among $x_1, \ldots, x_n$. Then a `Select` statement for P with the arguments $x_1, \ldots, x_n$ is interpreted as follows:

$$
\begin{aligned}
&\langle\{(pid, a_s, \mathsf{Select}(P, [x_1, \ldots, x_n])\circ\sigma,\eta)\}\cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle\\
&\longrightarrow \langle\{(pid, a_s, \sigma,\eta')\}\cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle
\end{aligned}
$$

where

$$\eta' = \begin{cases} \eta[\forall i \in I_{FV}(x_1, \dots, x_n).\ x_i \mapsto v_i] & \text{if } \exists v_1 \exists v_2 \dots \exists v_n. \\ & \qquad [\![P(v_1, \dots, v_n)]\!]_{\mathcal{F}} = \top \\ \eta[\forall i \in I_{FV}(x_1, \dots, x_n).\ x_i \mapsto \texttt{None}] & \text{otherwise} \end{cases}$$

For defining the semantics of the `Iterate` statement, it is practical to abstract away the details of how the data over which the simulation iterates is generated. As described in Section 3.3.2, there are several different options for data sources at this point, namely fluents, Prolog predicates and Python functions or expressions. However, regardless of which option is chosen, the source is evaluated only once at the beginning and the iteration works on a snapshot of the data. Formally, let θ be an expression that is admissible for the use as a data source in an `Iterate` statement as defined in Definition 3.17. The result of the evaluation of θ at the time when the `Iterate` statement is reached, is denoted by $[\![\theta]\!]_{\mathcal{F},\eta} = [(v_{1,1}, \dots, v_{1,m}), \dots, (v_{n,1}, \dots, v_{n,m})]$, i.e. a sequence of tuples that contain the values that will be bound to the free variables in the `Iterate` block. With all that, the semantics of an `Iterate` statement can be defined recursively by the following rules:

$$\begin{aligned} &\langle \{(pid, a_s, Iterate(\theta, [x_1, \dots, x_m], \varsigma) \circ \sigma, \eta)\} \cup P_{run}, \\ &\qquad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F} \rangle \\ &\longrightarrow \langle \{(pid, a_s, Iterate([\![\theta]\!]_{\mathcal{F},\eta}, [x_1, \dots, x_m], \varsigma) \circ \sigma, \eta)\} \cup P_{run}, \\ &\qquad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F} \rangle \end{aligned}$$

$$\begin{aligned} &\langle \{(pid, a_s, Iterate([(v_{1,1}, \dots, v_{1,m}), \dots, (v_{n,1}, \dots, v_{n,m})], [x_1, \dots, x_m], \varsigma) \circ \sigma, \eta)\} \\ &\quad \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F} \rangle \\ &\longrightarrow \langle \{(pid, a_s, \\ &\quad \varsigma \circ Iterate([(v_{2,1}, \dots, v_{2,m}), \dots, (v_{n,1}, \dots, v_{n,m})], [x_1, \dots, x_m], \varsigma) \circ \sigma, \eta')\} \\ &\quad \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F} \rangle \\ &\qquad \text{where } \eta' = \eta[x_1 \mapsto v_1, \dots, x_m \mapsto v_m] \end{aligned}$$

$$\begin{aligned} &\langle \{(pid, a_s, Iterate(\emptyset, [x_1, \dots, x_m], \varsigma) \circ \sigma, \eta)\} \cup P_{run}, \\ &\qquad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F} \rangle \\ &\longrightarrow \langle \{(pid, a_s, \sigma, \eta)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F} \rangle \end{aligned}$$

3.6.4 Remarks

The presented semantics of SALMA's simulation mechanism has some important consequences that might not be apparent on first sight:

1. *Actions and events do not have a duration.* The system does not enforce a limit for the number of actions and events performed at the same time step. This means that the modeler is responsible for assuring that the simulation does not exhibit *Zeno behavior* (see [BK08]), which basically describes a system that performs an infinite number of actions within a finite time span. Every activity that blocks a process for a certain duration has to be modeled by an action that is followed by a `Wait`-statement (e.g. in Figure 3.12 on page 54).

2. *Actions and events are interleaved nondeterministically.* Each time a process executes an action, it is only performed after all other processes had the chance to schedule an action execution. All actions that have been collected in the action schedule *Act* and all due events are then performed in *random order* before any process can continue its execution. There is no built-in prioritization mechanism, neither with respect to processes nor actions or events. If such a mechanism is intended, it has to be modeled explicitly.

3.7 Summary

This chapter has introduced SALMA as an approach for discrete event simulation. It presented two modeling languages provided by SALMA, namely its Domain Description Language (SALMA-DDL) and its Agent Process Definition Language (SALMA-APDL), that can be used to define the *domain model* of the simulated system and the behavioral model of the involved agents. It was shown how the SALMA-DDL extends the basic situation calculus with additional concepts and constructs specifically geared towards discrete event simulation. At first, this includes an hierarchical sort system that is used to declare *finite entity types*. By restricting all fluent parameters to finite types, SALMA's simulation engine is able to use the progression mechanism of the simulation calculus to calculate the world state for the next simulation step. Another notable extension that is introduced by the SALMA-DDL is an axiom that defines under which conditions an event instance can be scheduled, i.e. at which point it is possible to determine the next occurrence time of an event instance. This extension allows the use of the event scheduling paradigm of discrete event simulation, which is in many scenarios much more efficient than a simulation strategy based on fixed equidistant time steps. While the domain description language is based on Prolog in the fashion of the original implementations of the simulation calculus, SALMA's agent modeling language is implemented as an *internal domain specific language* [Fow10] in Python. This API provides a similar structure as traditional agent programming languages based on the situation calculus like GoLog [L^+97]. The chosen architecture achieves a tight integration of the languages and allows leveraging the full potential of the Python platform. It was demonstrated along the running example of this chapter that the flexibility gained from this integration is very important for models in which agents have complex behavior. In particular, this is true when *adaptive* systems are modeled, since the adaptation mechanisms typically involve specialized and potentially complex computations for tasks like optimization, planning, or learning.

Besides providing a practical introduction to the SALMA modeling and simulation approach, this chapter presented a formal operational semantics of the simulation algorithm. This is intended to provide a thorough understanding of SALMA's concepts and mechanisms, which is a necessary for being able to fully comprehend the next chapter that describes SALMA's support for *statistical model checking.*

Since a technical presentation of SALMA's simulation algorithm would not have added much essential information to the discussion of the semantics, it was left out altogether. Instead, this chapter did provide strong evidence for the correctness of the implementation by means of a simple yet versatile example simulation experiment whose results could be visualized in a way that allowed a very exact comparison between the actual and the expected behavior of the model.

3.8 Related Work

One aspect of discrete event simulation that seems to be especially relevant in the context of the SALMA approach is the distinction between different *world views* that was already mentioned in Section 2.3, namely the *activity-oriented view*, the *event-oriented view*, and the *process-oriented view* (cf. [BCINN04, Chap. 3]). It was also mentioned in Section 2.3 that often mixtures of these world-views are used, in particular combinations of event scheduling and activity scanning, which is often labeled as *the three-phase approach* [Pid95]. As described in this chapter, a similar combination is also used within the SALMA simulation semantics that supports both *schedulable* and *immediate* events.

Many tools or libraries exist that support either one of the mentioned world-views or combinations of them. Among them, there are simulation libraries for general purpose programming languages that provide abstractions, runtime infrastructure and utilities that facilitate building simulations. Examples for freely available software packages of this kind are the process-oriented SimPy [MV03] library for Python or the DESMO-J[LP99] framework for Java that supports both the event-scheduling and the process-interaction world view. Clearly, SALMA is comparable to solutions like that with respect to the fact that it also presents itself as a framework with a Python API for building and running the simulation experiment. However, through the combination with the situation calculus, SALMA adds a declarative layer that often notably increases conciseness of the model.

Although it is not a technical restriction, SALMA is focused on modeling and simulating multi-agent systems. Multi-agent simulation has been adapted in many different fields, which has resulted in a broad spectrum of more or less specialized approaches. Widely used examples for domain-independent frameworks in that area are MASON (Java) [LCRP$^+$05] and RePast (Java, C++)[Col03]. Software packages like that offer highly flexible APIs at a relatively low level of abstraction. On the other hand, there are modeling and simulation approaches that are specialized on particular applications, e.g. the MatSim framework for multi-agent transport simulations [HNA16]. SALMA tries to provide as much flexibility as possible with regard to fitting simulations to the characteristics of the modeled domain. Most of all, this includes SALMA's ability to vary the level of detail and abstraction of the system model within a broad spectrum, which was discussed thoroughly in this chapter. However, the logic-based generic representation in SALMA is inherently much more computationally expensive than optimized specialized approaches. In particular, the application of SALMA might not be practical for models with very large numbers of agents, which is typical, for instance, in more realistic traffic simulation experiments. In such cases, it could still be beneficial to use SALMA as a supporting approach for analyzing certain parts or mechanisms of the model from a *microscopic perspective*. This will become even more apparent when SALMA's abilities for statistical model checking are discussed in the next chapter.

On the theoretical side, the DEVS (Discrete Event System Specification) [ZPK00] formalism, which was originally developed in the 1970s by Bernard P. Zeigler, has become very popular and there has been a lot of research directed to it. DEVS provides a system-oriented unified view on both continuous and discrete simulation models. Additionally, DEVS establishes mechanisms for hierarchical subsystem composition. In particular because of this compositionality, it would be an interesting branch of future work to develop a mapping of the SALMA semantics onto the DEVS formalism.

Chapter 4

Statistical Model Checking in SALMA

The previous chapter has introduced SALMA as a versatile approach for modeling and simulation of multi-agent systems. However, the real value of SALMA's consequent logical representation becomes visible when it is extended towards statistical model checking (see Section 2.5). This chapter first introduces the SALMA Property Specification Language (SALMA-PSL) that is used to formalize invariants and goals that are checked against each simulation run. Following that, a practical overview of SALMA's usage in a statistical model checking setting is given. In particular, it is shown how an experiment is set up and how results are analyzed by means of hypothesis tests and interval estimation. Based on this introduction, Chapter 5 will later discuss the mathematical and algorithmic details behind the evaluation of SALMA-PSL properties.

Remark: *SALMA's property specification language has already been described in [Kro14a] and [Kro14b]. Also, a simplified denotational semantics for most language elements was presented in [Kro14b]. However, some corrections and extensions have been made for this chapter and the discussion is much more thorough.*

4.1 SALMA's Property Specification Language

The predominant general method of reasoning about *execution traces* in computer science is to use *temporal logics*. In agreement with that, the SALMA Property Specification Language (SALMA-PSL) is based upon a first-order version of *bounded linear temporal logic (BLTL)* (see [Pnu77b]), a variant of LTL that adds an upper time bound for the temporal operators. This time bound guarantees that every formula can be confirmed or falsified by a simulation run with finite length. On top of the LTL-based foundation, the SALMA PSL adds several capabilities that facilitate reasoning about characteristic aspects of the situation-calculus-based model. This section first presents the syntax of SALMA-PSL formulas and then describes their semantics in detail.

4.1.1 Syntax and Language Structure

In the following, the syntax and structure of the SALMA property specification language is described throughout several separate definitions that specify the grammar using a simplified version of the EBNF. In particular, when an argument list is given as $(X, \ldots, X)$, this is an abbreviation for the EBNF expression $"("X\{","X\}")"$, i.e. it allows one or multiple arguments. Additionally, a “hat” decoration like $\hat{p}$ is used to indicate that a symbol has arity 0. The presentation starts with *formulas* as the top-level elements of the language and then discusses their ingredients one by one.

Definition 4.1 (Formulas)**.** The syntax of a SALMA-PSL formula Φ is specified by the following grammar:

$$\begin{aligned}\Phi ::= &\; \mathit{true} \mid \mathit{false} \mid \\ &\; \Theta \sim \Theta \mid \hat{p} \mid p(\Theta, \ldots, \Theta) \mid \hat{F_{\mathbb{B}}} \mid F_{\mathbb{B}}(\Theta, \ldots, \Theta) \mid \mathit{not}(\Phi) \mid \mathit{and}(\Phi, \ldots, \Phi) \mid \\ &\; \mathit{or}(\Phi, \ldots, \Phi) \mid \mathit{implies}(\Phi, \Phi) \mid \mathit{forall}(x : T, \Phi) \mid \mathit{exists}(x : T, \Phi) \mid \\ &\; \mathit{occur}(\alpha) \mid \mathit{always}(\tau, \Phi) \mid \mathit{eventually}(\tau, \Phi) \mid \mathit{until}(\tau, \Phi, \Phi) \mid \\ &\; \mathit{let}(z : \Theta, \Phi)\end{aligned}$$

Here, Θ represents a term as defined in Definition 4.2, $\sim$ is a comparison operator (like $<$, $>$, $=$, or $\neq$), $\hat{p}$ and p are situation-independent predicates, $\hat{F_{\mathbb{B}}}$ and $F_{\mathbb{B}}$ are relational fluents, x and z are variable names, T is the name of a *finite* sort, α is an action instance as in Definition 4.5, and τ is an admissible time bound expression (see Definition 4.4). Besides that, *not*, *and*, *or*, and *implies* are the usual logical connectives. Furthermore, *forall* and *exists* denote the universal and existential quantifiers that are restricted to finite domains, which have to be specified by means of an entity sort name after the colon in the quantifier's variable declaration. The special predicate *occur* tests whether an action or event was performed or happened in the current time step. As mentioned in the beginning, the temporal operators *always*, *eventually*, and *until* are restricted with a finite time bound that is specified as the first argument. Finally, the keyword *let* provides a means for assigning expression values to variables that can be reused within nested subformulas.

One point that immediately strikes out in the definition above is that fluents are used without a *situation argument.* Indeed, situation terms are suppressed entirely in SALMA-PSL formulas and restored during evaluation according, either by inserting the current situation or a situation constructed in the context of lookahead evaluation (see Section 5.6.3). The basic ingredients of a formula are *terms*, which can be used as arguments of predicates or relational fluents or in comparisons.

Definition 4.2 (Terms)**.** The syntax of SALMA-PSL terms, represented by Θ, is specified by the following grammar:

$$\Theta ::= \Theta_{\mathbb{R}} \mid x \mid e \mid \hat{f_T} \mid f_T(\Theta, \ldots, \Theta) \mid \hat{F_T} \mid F_T(\Theta, \ldots, \Theta)$$

Here $\Theta_{\mathbb{R}}$ is a numeric term (see Definition 4.3 below), x is a variable whose type is a finite sort, e is an entity value, T is a finite sort from the model, $\hat{f_T}$ and f_T are situation-independent functions of type T, and $\hat{F_T}$ as well as F_T are functional fluents of type T.

In the general definition of terms above, a distinction is made between *numeric* and non-numeric terms, i.e. terms that represent a value from a finite entity sort. The reason for this is that only the former may be used within arithmetic expressions.

Definition 4.3 (Numeric terms)**.** *Numeric terms* have the following syntax:

$$\begin{aligned}\Theta_{\mathbb{R}} ::= {} & x \mid c \mid \Theta_{\mathbb{R}} \circledast \Theta_{\mathbb{R}} \mid f_{\mathbb{R}}(\Theta, \ldots, \Theta) \mid F_{\mathbb{R}}(\Theta, \ldots, \Theta) \mid \\ & lastTime(\alpha)\end{aligned}$$

Here x is a numeric variable, c is a numeric literal, $\circledast$ is an arithmetic operator (like $+$, $-$, $*$, $/$), $\hat{f_{\mathbb{R}}}$ and $f_{\mathbb{R}}$ are numeric situation-independent functions, and both $\hat{F_{\mathbb{R}}}$ and $F_{\mathbb{R}}$ are numeric fluents. Furthermore, α is action instance and *lastTime* is a special function that returns the last time at which the action instance α occurred.

It is actually necessary to constrain numeric terms further when they are used for specifying time bounds in temporal operators. In fact, SALMA only allows time bounds specified by either a natural number literal or a variable that is bound in a *let*-expression that encapsulates the temporal operator. The restriction to a plain natural number instead of a more complex expression is necessary because of the way such expressions are evaluated, which will be explained in Chapter 5. Without going into details, it can be said that a compound expression is translated into a conjunction of chained evaluations and variable assignments. However, because the time bound would have to be calculated before the temporal operator expression, it would not be possible to treat this temporal expression separately. Since that is necessary for the evaluation goal scheduling mechanism (see Section 5.5), allowing arbitrary expressions time bounds would require a separate evaluation strategy. A solution to this problem is to move the desired time bound expression to the definition of a variable in a *let*-expressions. During evaluation, the variable is replaced by the expression's value and the *let*-block is eliminated. For the evaluation goal schedule, this effectively has the same effect as using a numeric literal. Since it would be rather hard to guarantee that an expression always evaluates to a natural number, the SALMA-PSL interpreter allows any real number as time bound and rounds it appropriately (see Section 4.1.3). The restrictions defined above are summarized in the next definition.

Definition 4.4 (Admissible time bound expression)**.** An expression τ is an admissible time bound expression if it is either a numeric literal, i.e. $\tau \in \mathbb{N}$, or the temporal operator that uses τ is nested in the body ϕ within a *let*-expression of the form $let(\tau : \Theta_{\mathbb{R}}, \phi)$ where $\Theta_{\mathbb{R}}$ is a numeric term as defined above.

As seen above, it is sometimes necessary to refer to an *action instance*, i.e. an action symbol together with a combination of arguments for all parameters according to the action's signature (see Section 3.2.3).

Definition 4.5 (Action instance)**.** An *action instance* is a special term constrained by the following grammar:

$$\alpha ::= \hat{a} \mid a(\Theta, \ldots, \Theta) \mid \hat{ev} \mid ev(\Theta, \ldots, \Theta)$$

Here Θ represents an arbitrary term, $\hat{a}$ and $\hat{ev}$ are 0-ary action and event symbols, and a and ev are n-ary action and event symbols from the model signature.

Similarly, a fluent instance refers to a concrete selection of values for a fluent.

Definition 4.6 (Fluent instance)**.** The syntax of a fluent instance is defined as follows:

$$\gamma ::= \hat{F_T} \mid F_T(\Theta, \ldots, \Theta)$$

Here, Θ represents an arbitrary term, and $\hat{F_T}$ and F_T are 0-ary and n-ary fluent symbols of type T.

The definitions above have introduced all constituents that are used to build formulas and terms in SALMA-PSL. The last step that is necessary in order to make a SALMA-PSL formula usable as a property that is checked during statistical model checking is to mark it as either an *invariant* or a *goal*. This information is added by wrapping the formula in one of the *pseudo-operators invariant* or *goal*.

Definition 4.7 (SALMA-PSL property)**.** The syntax of a SALMA-PSL property Prop is specified by the grammar

$$\mathsf{Prop} ::= \ \mathit{invariant}(\Phi) \mid \mathit{goal}(\Phi)$$

where Φ can be any formula as defined in Definition 4.1.

Before the formal semantics of SALMA-PSL formulas is presented in Section 4.1.3, it is helpful to examine some examples that demonstrate the typical usage of the language constructs.

4.1.2 Examples

Since the SALMA property specification language is based on LTL, much of the typical structure of linear time temporal logic formulas can be found in SALMA-PSL properties. Typically, a formula refers to a start point that is marked either by a *state constraint*, i.e. a condition built out of fluent values, or by the occurrence of an action or event, which can be expressed using the special predicate *occur*. By using nested temporal operators, it is possible to describe complex sequences of expected behavior. For instance, the following formula could be used to express some time constraints for robots from the example that was introduced in Section 3.1:

```
forall(r:robot, implies(occur(activate(r)),
    eventually(10, until(100, moving(r),
        and(occur(grab(r, ?)), eventually(200, atBase(r)))))))
```

This property requires that the behavior of every robot must fulfill the following constraints:

1. It starts moving within 10 time units after it is activated.
2. Then, after continuously moving for at most 100 time units, it grabs some item that is not specified further.
3. Finally, it returns to the base station within 200 time units.

One of the most important aspects of the SALMA-PSL is its ability to refer to properties of entities and relation between entities in a very detailed way. Often, it is very useful to store the value of more complex expressions in variables that can be re-used later. This can be seen in Figure 4.1.

Here, the variable `critDist` is a typical example for the re-use of an expression, in this case meant to be a critical distance to the designated destination

```
forall(r:robot, forall(i:item,
  implies(occur(grab(r, i)),
    let(critDist: min(25, speed(r) * 10), let(tlim: deadline(i) - time,
      until(tlim,
        forall(r2:robot, forall(ws:workstation,
          and(
            implies(r2 =/= r, distance(r, r2) > crtitDist),
            implies(ws =/= destination(i), distance(r, ws) > crtitDist)))),
        delivered(i)))))))
```

Figure 4.1: Example for the use of variables and **let** blocks.

of an item. In the `let`-expression, this variable is set to the distance the robot would cover within 10 time units, or at least 25 length units. Within the body of a let expression, the defined variable can be re-used at any position. In this case, `critDist` is used twice to express safety distances to all other robots and all workstations except the destination of the delivered item. In contrast, the other variable `tlim` is required because the SALMA-PSL does not allow arbitrary expressions for time bounds of temporal operators (see Definition 4.4). Here, `tlim` is used to set a time bound that respects a deadline for the delivery of an item.

The examples above already contain essentially all core constructs that are currently defined in the SALMA-PSL. Many more sophisticated elements that are found in other expression languages, such as aggregation or filter operators, do not exist in a generic, model-independent way. However, one important strength of the SALMA approach is that it is possible to integrate any Prolog (ECLiPSe) predicate or function, including both those defined by the user and those shipped with standard or 3rd-party libraries. For example, it might be useful to define a function that calculates the maximum horizontal position that any robot has in the current situation. This can easily be expressed in plain Prolog:

```
maxxpos(XMax) :-
    domain(robot, Robots, s0),
    member(R, Robots), xpos(R, XMax, s0),
    not (member(R2, Robots), R \= R2,
        xpos(R2, XR2, s0), XR2 > XMax), !.
```

With this user-defined function, it is now possible to define a SALMA-PSL property which expects any robot that picks up an item to move past the current maximum X position:

```
forall(r:robot,
    implies(occur(grab(r,?)),
        let(mx:maxxpos, eventually(50, xpos(r) > mx))))
```

Here, the use of the `let`-expression it is actually necessary. In fact, when the property above is evaluated, the current value of `maxxpos` is syntactically substituted for the variable `mx` and the resulting rewritten formula is scheduled for further inspection (see Section 5.5). This means that the *current* maximum position at the moment the robot grabs the item is *frozen* and set as the goal. If the function `maxxpos` was used directly in `eventually` expression, the current position of the robot would in any step be compared to the maximum X position at that time, which would make the constraint unsatisfiable.

4.1.3 Semantics of SALMA-PSL Properties

In order to properly describe the semantics of the language, it is first necessary to introduce the concept of a *simulation trace.*

Definition 4.8 (Simulation trace)**.** A simulation trace of a model $\mathcal{M}$, denoted by $\sigma_{\mathcal{M}}$ is a finite sequence of *situations* $S_0, \ldots, S_n$, where S_0 is the initial situation at the simulation start, S_i is the situation after i actions have been performed, and S_n is the situation that is present in the latest simulation step. Furthermore, σ_k^n is the *segment* of the simulation trace that starts at situation S_k and ends in situation S_n.

This notion of a *trace segment* is important because in general, the interpretation of a SALMA-PSL formula depends on both a reference (or start) point and the time span from that point up to the end of the observed simulation trace.

Definition 4.9 (Interpretation with respect to a simulation trace segment)**.** Given a term Θ and a simulation trace segment σ_k^n, the interpretation of Θ with respect to σ_k^n is denoted as $[\![\Theta]\!]_k^n$. This can be understood as the value that is assigned to Θ for the situation S_k when the future of S_k is known up to S_n.

Another concept that is necessary to express the semantics of formulas in terms of the situation calculus is *regression*, which was introduced in Section 2.2. The application of the regression operator to the term Θ, written as $\mathcal{R}(\Theta)$, transforms Θ to a term that still expresses the same value but refers only to the initial situation S_0. This allows using the abbreviated statement $\mathcal{M}, S_0 \models \mathcal{R}(\Phi)$ to expresses that Φ is entailed by the basic action theory that is formed by the currently simulated model with the initial situation S_0. Similarly, when a formula is entailed for all possible initial situations, this is written as $\mathcal{M} \models \Phi$.

In the chosen semantics, the interpretation of a formula at a given situation S_k is understood in the context of a simulation trace with a fixed last situation S_n. However, for formulas that contain temporal operators, it is not always possible to decide whether they are true or false based only on the available simulation trace segment σ_k^n. Therefore, it is necessary to use a *three-valued logic* that allows expressing that in a given situation, the result of a formula cannot be decided yet. Throughout the remainder of this thesis, the three possible options will be represented by the symbols $\top$, $\bot$, and ?.

With the concepts and definitions from above, the semantics of SALMA-PSL formulas can be defined in the context of the situation calculus.

Definition 4.10 (Semantics of a SALMA-PSL expression)**.** Let v be a numeric or boolean literal or a constant of an entity sort, $\theta_{\mathbb{R},1}, \ldots, \theta_{\mathbb{R},n}$ numeric terms, $\odot$ an arithmetic operator, $\sim$ a comparison operator (i.e. one of $<$, $\leq$, $=$, $\geq$, $>$, or $\neq$), $\theta, \theta_1, \ldots, \theta_n$ arbitrary terms, and $\Phi, \Phi_1, \ldots, \Phi_n$ formulas. Furthermore, let $i, j, k, n \in \mathbb{N}_0$ be situation indexes, and let $T \in \mathbb{N}_0$ represent a time limit. Finally, let α and γ be an action and a fluent instance as defined in Definition 4.5 and Definition 4.6, respectively. Then the semantics of a SALMA-PSL expression Θ with respect to the trace segment σ_k^n is recursively defined as follows:

1. **constants:**

$$[\![v]\!]_k^n = \begin{cases} v & \text{if } v \in \mathbb{R} \text{ or } v \in T \text{ where } T \text{ is an entity sort} \\ \top & \text{if } v = \mathit{true} \\ \bot & \text{if } v = \mathit{false} \end{cases}$$

2. **arithmetic expressions:** $[\![\theta_{\mathbb{R},1} \odot \theta_{\mathbb{R},2}]\!]_k^n = [\![\theta_{\mathbb{R},1}]\!]_k^n \odot [\![\theta_{\mathbb{R},2}]\!]_k^n$.

3. **functional expressions:** $[\![g(\theta_1, \ldots, \theta_n)]\!]_k^n = v$
   if (a) g is a situation-independent function with arity n and the evaluation of $g([\![\theta_1]\!]_k^n, \ldots, [\![\theta_n]\!]_k^n)$ yields the result v, or (b) g is a functional fluent and $\mathcal{M}, S_0 \models \mathcal{R}(g([\![\theta_1]\!]_k^n, \ldots, [\![t_n]\!]_k^n, S_k)) = v$.

4. **relational expressions:**

   a) $[\![f(\theta_1, \ldots, \theta_n)]\!]_k^n = \top$ if (a) f is a situation-independent predicate and $\mathcal{M} \models f([\![\theta_1]\!]_k^n, \ldots, [\![\theta_n]\!]_k^n)$, or (b) f is a relational fluent and $\mathcal{M}, S_0 \models \mathcal{R}(f(\theta_1, \ldots, \theta_n, S_k))$.

   b) $$[\![\theta_1 \sim \theta_2]\!]_k^n = \begin{cases} \top & \text{if } [\![\theta_1]\!]_k^n \sim [\![\theta_2]\!]_k^n \\ \bot & \text{otherwise} \end{cases}$$

5. **temporal expressions:**

a) $$[\![\mathbf{eventually}(T,\Phi)]\!]_k^n = \begin{cases} \top & \text{if } \exists j \in [k, \min(k+\lfloor T \rfloor, n)].\ [\![\Phi]\!]_j^n = \top \\ \bot & \text{if } n \geq k + \lceil T \rceil \land \\ & \qquad \forall j \in [k, k+\lceil T \rceil].\ [\![\Phi]\!]_j^n = \bot \\ ? & \text{otherwise} \end{cases}$$

b) $$[\![\mathbf{always}(T,\Phi)]\!]_k^n = \begin{cases} \top & \text{if } n \geq k + \lceil T \rceil \land \\ & \qquad \forall j \in [k, k+\lceil T \rceil].[\![\Phi]\!]_j^n = \top \\ \bot & \text{if } \exists j \in [k, \min(k+\lfloor T \rfloor, n)].\ [\![\Phi]\!]_j^n = \bot \\ ? & \text{otherwise} \end{cases}$$

c) $$[\![\mathbf{until}(T,\Phi_1,\Phi_2)]\!]_k^n = \begin{cases} \top & \text{if } \exists j \in [k, \min(k+\lfloor T \rfloor, n)].\ [\![\Phi_2]\!]_j^n = \top \land \\ & \qquad \forall i \in [k, j[.\ [\![\Phi_1]\!]_i^n = \top \\ \bot & \text{if } \Big(\exists i \in [k,n].\ [\![\Phi_1]\!]_i^n = \bot \land \\ & \qquad \forall j \in [k,i].\ [\![\Phi_2]\!]_j^n = \bot\Big) \\ & \ \lor \Big(n \geq k + \lceil T \rceil \land \\ & \qquad \forall j \in [k, k+\lceil T \rceil].\ [\![\Phi_2]\!]_j^n = \bot\Big) \\ ? & \text{otherwise} \end{cases}$$

6. **logical connectives:**

a) $$[\![\mathbf{not}(\Phi)]\!]_k^n = \begin{cases} \top & \text{if } [\![\Phi]\!]_k^n = \bot \\ \bot & \text{if } [\![\Phi]\!]_k^n = \top \\ ? & \text{otherwise} \end{cases}$$

b) $$[\![\mathbf{and}(\Phi_1,\ldots,\Phi_n)]\!]_k^n = \begin{cases} \top & \text{if } \forall 1 \leq i \leq n.\ [\![\Phi_i]\!]_k^n = \top \\ \bot & \text{if } \exists 1 \leq i \leq n.\ [\![\Phi_i]\!]_k^n = \bot \\ ? & \text{otherwise} \end{cases}$$

c) $$[\![\mathbf{or}(\Phi_1,\ldots,\Phi_n)]\!]_k^n = \begin{cases} \top & \text{if } \exists 1 \leq i \leq n.\ [\![\Phi_i]\!]_k^n = \top \\ \bot & \text{if } \forall 1 \leq i \leq n.\ [\![\Phi_i]\!]_k^n = \bot \\ ? & \text{otherwise} \end{cases}$$

d) $$[\![\mathbf{implies}(\Phi_1,\Phi_2)]\!]_k^n = [\![\mathbf{or}(\mathbf{not}(\Phi_1),\Phi_2)]\!]_k^n$$

7. **quantifiers:**

$$\text{a)}\ [\![\mathbf{forall}(x:T,\Phi)]\!]_k^n = \begin{cases} \top & \text{if } \forall e \in [\![\mathit{domain}(T)]\!]_k^n. \\ & \quad [\![\Phi[e/x]]\!]_k^n = \top \\ \bot & \text{if } \exists e \in [\![\mathit{domain}(T)]\!]_k^n. \\ & \quad [\![\Phi[e/x]]\!]_k^n = \bot \\ ? & \text{otherwise} \end{cases}$$

$$\text{b)}\ [\![\mathbf{exists}(x:T,\Phi)]\!]_k^n = \begin{cases} \top & \text{if } \exists e \in [\![\mathit{domain}(T)]\!]_k^n. \\ & \quad [\![\Phi[e/x]]\!]_k^n = \top \\ \bot & \text{if } \forall e \in [\![\mathit{domain}(T)]\!]_k^n. \\ & \quad [\![\Phi[e/x]]\!]_k^n = \bot \\ ? & \text{otherwise} \end{cases}$$

$$8.\ [\![\mathbf{occur}(\alpha)]\!]_k^n = \begin{cases} \top & \text{if } S_k = do(\alpha, S_{k-1}) \\ \bot & \text{otherwise} \end{cases}$$

$$9.\ [\![\mathbf{lastTime}(\alpha)]\!]_k^n = \begin{cases} t & \text{if } \exists k.\, S_k = do(\alpha, S_{k-1}) \wedge \mathit{time}(S_k) = t \wedge \\ & \quad \nexists k'.k' > k \wedge S_{k'} = do(\alpha, S_{k'-1}) \\ -1 & \text{if } \nexists k.\, S_k = do(\alpha, S_{k-1}) \end{cases}$$

$$10.\ [\![\mathbf{let}(x:\theta,\Phi)]\!]_k^n = \begin{cases} \top & \text{if } [\![\Phi[[\![\theta]\!]_k^n/x]]\!]_k^n = \top \\ \bot & \text{if } [\![\Phi[[\![\theta]\!]_k^n/x]]\!]_k^n = \bot \\ ? & \text{otherwise} \end{cases}$$

The first interesting point in the definition above is how fluents are interpreted in the rules 3 and 4a. Here, the connection is made between the SALMA-PSL semantics and the situation calculus by reducing the interpretation with respect to a simulation trace segment (see Definition 4.9) to the application of the *regression operator* of the situation calculus that was shortly introduced in Section 2.2. There it was already explained that in SALMA, regression is actually never used in the traditional sense, in which the initial situation would refer to the start of the simulation. Instead, *progression* is used, which effectively means that the database is updated in each step so that the initial situation S_0 always directly represents the current simulation state. Thus, the regression-based viewpoint above should be understood as an abstraction that allows a more concise description.

The semantics of the temporal operators clarifies that the current valuation of a SALMA-PSL formula always has to be understood with respect to the current trace segment. A definite value ($\top$ or $\bot$) is assigned only if the requirements for the temporal operator are either clearly fulfilled or violated within the currently accessible time horizon. Otherwise, the marker ? is used to declare that no conclusive decision has been found yet. Naturally, this marker for indefiniteness dominates over other values in the logical operators **and**, **or**, and **not**, i.e. one indefinite part of such an expression is enough to prevent a definite decision for the whole expression. As mentioned in Section 4.1.1, the time bound for a temporal operator is not guaranteed to be a natural number. Since SALMA uses a discrete time base, definite decisions are only possible at time points that are whole numbers. This is reflected by the floor and ceiling functions in the definitions above.

Another crucial aspect of the interpretation of formulas is the way in which the *range* of quantifiers is determined in rule 7. In fact, the entities that are substituted for the quantifier's variable are taken from $[\![domain(T)]\!]_k^n$, which is the *current* content of the domain of sort T in situation S_k. This actually leads to an evaluation semantics that conforms to a natural understanding of the temporal operators. For instance, a model describing a task scheduling system could contain a sort that represents the set of all tasks that currently exist in the system. If the model is intended for long-running simulations, then it makes sense to integrate the possibility of new tasks being scheduled or finished tasks being deleted. This means that the domain of the sort *task* could be changing throughout a simulation run. In such a model, one could imagine a simple requirement like the following:

implies(**occur**(snapshot), **eventually**(10,
  forall(t : task, **occur**(writeStackTrace(t))))

This formula says that as soon as a `snapshot` event occurs, all active tasks in the system have to write out their stack traces within 10 time units. In this case it is clear that a task entity that is added to the domain *after* the `snapshot` event occurred should not be included in the quantifier inside the until block because the task did not receive the signal in the first place. Here, the static unfolding of domains actually achieves this since the formula that is registered in the evaluation goal schedule when the trigger event occurs automatically refers only to the entities that exist at that time.

The rules for the quantifiers are followed by the definition of predicates **occur** and the function **lastTime** that provide access to the last occurrence of a specific action instance. It can be seen that both constructs are interpreted based on the *simulation history.* However, like the use of the regression operator above, this is an abstraction that is meant to achieve a more concise presentation. In fact, the SALMA evaluation mechanism does not store a longer part of the history but uses a clock-mechanism that records time-stamps for actions and events (see Section 5.6.8).

In each step of the simulation, when the evaluation mechanism has calculated interpretations for the formulas of all properties that are registered to be checked during the experiment, the SALMA runtime combine these results and tries to find a verdict for the current simulation trace. At this point, the distinction between goals and invariants matters. Basically, a simulation will be canceled and declared as a failure if at least one invariant from the active *property collection* is violated. On the other hand, it will be declared as success as soon as all goals are fulfilled. However, sometimes it is not easy to formulate proper goals in order to define the end condition of simulation. Besides that, it could be possible that the formulated goals are not reached at all or only after an unbearable long simulation runtime. Therefore, it is also possible to specify a time limit at which the simulation is ended at the latest if no conclusive result has been found yet. In this case, the simulation run is found to be a success if no goals were specified and is left as inconclusive (?) if at least one goal is still unsatisfied when the time limit is reached. This behavior is summarized in the following definition.

Definition 4.11 (Interpretation of a property collection)**.** Let $\mathcal{PC}$ be a property collection, i.e. a set of invariants or goals that are specified as in Definition 4.7, and let σ_0^n be the simulation trace segment observed up to the current step (n). Furthermore, let T_{lim} be a time limit specified for the experiment. Then, the current interpretation of the property collection $\mathcal{PC}$ with respect to σ_0^n is defined as follows:

$$
[\![\mathcal{PC}]\!]_0^n = \begin{cases}
\top & \text{if } (\forall Inv \in \mathcal{PC}.\, Inv = invariant(\Phi_{inv}) \implies \\
& \qquad \forall i \in [0,n].\, [\![\Phi_{inv}]\!]_i^n = \top) \\
& \quad \wedge \Big(\big((\forall G \in \mathcal{PC}.\, G = goal(\Phi_G) \implies \exists j \in [0,n]\, [\![\Phi_G]\!]_j^n = \top) \\
& \qquad \wedge\, time(S_n) < T_{lim}\big) \\
& \qquad \vee \big((\nexists G \in \mathcal{PC}. G = goal(\Phi_G))\big)\Big) \\
\bot & \text{if } \big(\exists Inv \in \mathcal{PC} \exists i \in [0,n].\, Inv = invariant(\Phi_{inv}) \\
& \qquad \wedge [\![\Phi_{inv}]\!]_i^n = \bot\big) \\
& \quad \vee \big(time(S_n) \geq T_{lim} \wedge \\
& \qquad \exists G \in \mathcal{PC}. G = goal(\Phi_G) \wedge \forall j \in [0,n]. [\![\Phi_G]\!]_j^n = \bot\big) \\
? & \text{otherwise}
\end{cases}
$$

The definitions in this section define the semantics of SALMA-PSL properties in a concise but relatively abstract manner. How these rules are implemented by SALMA's property evaluation mechanism is the topic of Chapter 5. First, however, it is time to return to more practical issues, namely how statistical model checking experiments are actually performed with SALMA.

4.2 Framework Support for Statistical Model Checking

In Section 3.4.4, it was explained how the SALMA framework is used to set up an experiment that consists of a domain model, an agent behavior model, a set of probability distributions for actions and events, and a configuration for the initial situation. In order to use such an experiment for statistical model checking, the following additional steps have to be performed.

1. Invariants and goals have to be registered in the *property collection* that is associated with the experiment.

2. Optionally, a time limit can be specified to ensure that simulation trials are not executed unnecessarily long.

3. A hypothesis test can be set up for the experiment. The sampling strategy of the test determines how many repetitions are conducted. This can either happen a-priori due to some heuristic or dynamically, which means that after each evaluation step, the test strategy decides whether further trials are required or not.

4. The configured experiment is executed repeatedly either for a preconfigured number of trials or until the hypothesis test accepts an hypothesis.

5. The results gathered during the repeated simulations can be used for further statistical analysis, e.g. for estimating a confidence interval for the success probability (see Section 4.3).

The part of the SALMA framework that is responsible for realizing statistical model checking is shown in Figure 4.2. One central point is that through the interfaces `HypothesisTest` and `ExperimentRunner`, it is possible to specify different hypothesis tests and strategies for the execution of repeated simulation trials. A typical example for a hypothesis test that is very suitable for statistical model checking is Wald's sequential probability ratio test (SPRT, [W$^+$45]) that was introduced in Section 2.5.1. As described there, the SPRT is able to determine dynamically when enough trials have been conducted for a given set of parameters. An implementation of this test is provided by the SALMA framework.

For the execution strategy, the framework currently offers only a basic implementation that runs simulations sequentially on the same CPU core. However, it would be a straightforward task to create an implementation of the `ExperimentRunner` interface that performs multiple simulations in parallel and aggregates the results. In fact, due to the recent technical developments in the field of distributed computing and "Big Data", the prospect of moving

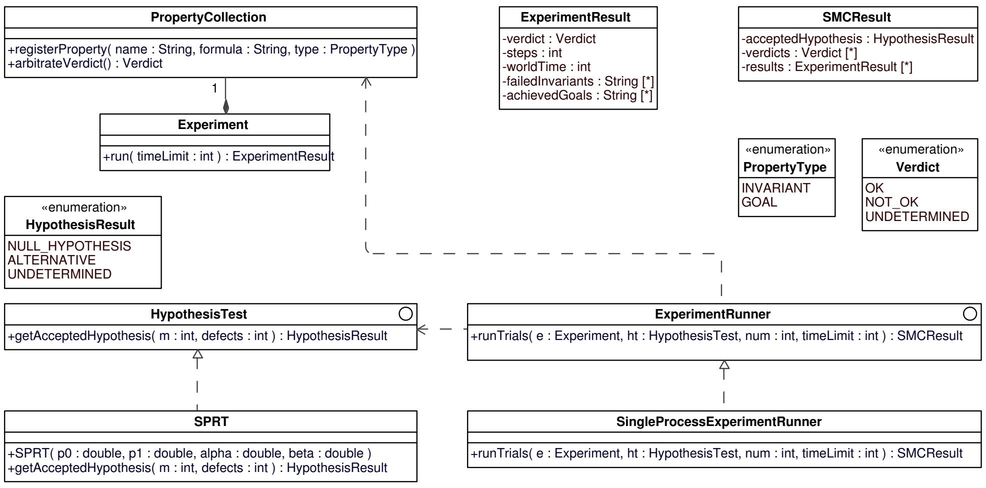

PropertyCollection
+registerProperty(name : String, formula : String, type : PropertyType)
+arbitrateVerdict() : Verdict
1
Experiment
+run(timeLimit : int) : ExperimentResult
ExperimentResult
-verdict : Verdict
-steps : int
-worldTime : int
-failedInvariants : String [*]
-achievedGoals : String [*]
SMCResult
-acceptedHypothesis : HypothesisResult
-verdicts : Verdict [*]
-results : ExperimentResult [*]
«enumeration» PropertyType
INVARIANT
GOAL
«enumeration» Verdict
OK
NOT_OK
UNDETERMINED
«enumeration» HypothesisResult
NULL_HYPOTHESIS
ALTERNATIVE
UNDETERMINED
HypothesisTest
+getAcceptedHypothesis(m : int, defects : int) : HypothesisResult
ExperimentRunner
+runTrials(e : Experiment, ht : HypothesisTest, num : int, timeLimit : int) : SMCResult
SPRT
+SPRT(p0 : double, p1 : double, alpha : double, beta : double)
+getAcceptedHypothesis(m : int, defects : int) : HypothesisResult
SingleProcessExperimentRunner
+runTrials(e : Experiment, ht : HypothesisTest, num : int, timeLimit : int) : SMCResult

Figure 4.2: SALMA framework support for statistical model checking.

statistical model checking approaches like SALMA to large distributed infrastructures has become realistic. The discussion will come back to this topic shortly in the outlook of this thesis in Section 7.3.

4.3 A Detailed Predictable Example

As a first demonstration of SALMA's usage for statistical model checking, this section presents an example that is small and simple enough to be understood instantly but still exercises the most important concepts. Another beneficial feature of this example is that it is possible to calculate an exact probability for the success of a simulation trial. This allows its use as a system test case that validates the core functionality of both the simulation engine and the property evaluation mechanism.

The scenario that is discussed here is based on the domain model that was introduced in Section 3.1. Just like there, the model used in this section describes a simple world where robots are moving around a two-dimensional discrete space and are able to carry items with them. However, unlike in Section 3.1, it is assumed here that each time a robot grabs an item, the grip it has on it can be of different quality. Depending on grip quality, there is a certain probability of the item being dropped accidentally. In the example, this is modeled as a *stochastic action* `pickUp` whose single outcome is a modified version of `grab` that now has an integer stochastic parameter `grip` which represents the grip quality by means of a grade from 1 (perfect grip) to 4 (very bad grip). This value is stored in a fluent `grip`, which serves as the conditioning state for the probability distribution that governs the occurrence of the accidental drop event. As in the original example in Chapter 3, the robot moves in discrete steps that are started intentionally by the agent

```
stochastic_action(pickUp, [r:robot, i:item], [grab]).
primitive_action(grab, [r:robot, i:item, grip:integer]).
fluent(grip, [r:robot], integer).
schedulable(accidental_drop(R,I), S) :-
      action_occurred(grab(R,I,_), S).
effect(carrying(Rob, Item), grab(Rob, Item, _), _, true, _).
effect(carrying(Rob, Item), accidental_drop(Rob, Item), _, false, _).
effect(grip(Rob), grab(Rob, _, NewGrip), _, NewGrip, _).
```

Figure 4.3: Excerpt of the robot domain model with grip quality.

with `move_?`-actions and that end when a `finish_step` event occurs after a stochastic delay. The relevant excerpt of the domain model that sets up the new rules for modeling grip quality is shown in Figure 4.3.

Besides the new grip quality aspect, the domain model is further extended with the parametrized constants named `destX` and `destY` that store the coordinates of a *destination* for an item, i.e. the location to which the item should be delivered by a robot. Although both the initial location and the destination of each item could be any two-dimensional position, it is assumed in this example that a) each item is initially located at the same position as the robot that is assigned to deliver it, and b) the destination for each item is set to a point strictly to the right of its initial position. This means that the movement of the robots can be limited to a series of steps to the right. It will become clear below why this greatly simplifies the validation of the simulation results.

Based on the presented domain model, the concrete example of this section equips robots with a simple agent control process that picks up a predetermined item and moves to the right until the item's destination is reached. Then, it drops the item intentionally, which is meant to represent delivery in the example. Figure 4.4 shows the Python code fragment that creates a robot agent with the described control procedure that is registered as a *one-shot process*, which means that it is only executed once.

One of the essential configuration steps for the simulation is the definition of probability distributions for the exogenous actions `finish_step` and `accidental_drop` and for the stochastic action `pickUp`. The code fragment that does this is shown in Figure 4.6. For the *occurrence distribution* of the event `finish_step`, the example actually uses the constant value 1, which means that each movement step is guaranteed to have a fixed duration. Although somewhat unrealistic, this abstraction is crucial for the calculation of an exact success probability (see below). The stochastic action `pickUp` has only one possible outcome, namely `grab`, so no selection distribution has to be specified. With `map_param`, the robot and item arguments of the `pickUp` ac-

```
def create_robot(self, num):
    myItem = Variable("myItem")
    proc = OneShotProcess([
        Act("pickUp", [SELF, myItem]),
        While("xpos(self) < destX(myItem)", [
            Act("move_right", [SELF]),
            Wait("not moving(self)")
        ]),
        If("carrying(self, myItem)",
          Act("drop", [SELF, myItem]))
    ])
    agent = Agent("rob" + str(num), "robot", [proc], myItem="item" + str(num))
    return agent
```

Figure 4.4: Agent process controlling robots in SMC example.

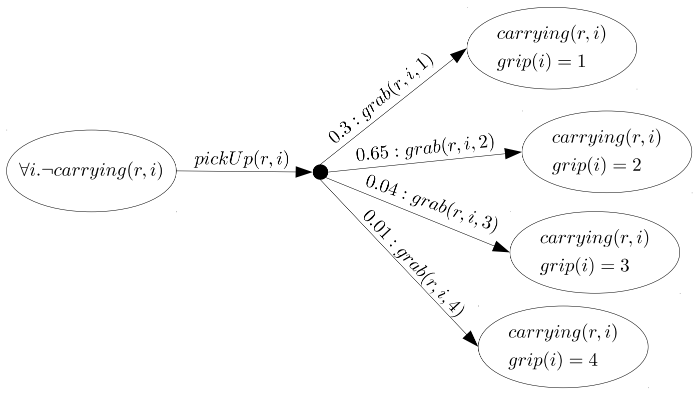

$\forall i.\neg carrying(r,i)$
$pickUp(r,i)$
$0.3 : grab(r,i,1)$
$0.65 : grab(r,i,2)$
$0.04 : grab(r,i,3)$
$0.01 : grab(r,i,4)$
$carrying(r,i)$ $grip(i)=1$
$carrying(r,i)$ $grip(i)=2$
$carrying(r,i)$ $grip(i)=3$
$carrying(r,i)$ $grip(i)=4$

Figure 4.5: Probabilistic transitions for the `pickUp` action.

tion, which is performed intentionally by the robot agent, are passed through to the action `grab`. However, in the updated domain model of this example, `grab` has the additional parameter `grip`. For this parameter, a categorical distribution is installed that assigns a distinct probability to each possible grip quality grade. In the concrete configuration used for this example, four different grip quality states are distinguished and encoded by the numbers 1 to 4, where 1 is meant to denote perfect grip and 4 very bad grip. The probability mapping used in this example was $\{1 \mapsto 0.3, 2 \mapsto 0.65, 3 \mapsto 0.04, 4 \mapsto 0.01)$. Given the effect axioms from Figure 4.3, this leads to the transition system fragment in Figure 4.5.

In order to model the effect of the grip quality, the occurrence delay distribution for the event `accidental_drop` is set to a conditional geometric distribution that chooses its parameter dependent on the value of the grip fluent. This is realized by a closure that is created over the custom Python function `accidental_drop_delay` together with an array of probabilities for each possible grip value. This closure is installed as a `CustomDistribution` and will hence be used to sample occurrence delays for `accidental_drop` event instances. The integration works similar to Python functions that are called in agent processes to calculate values for variable assignments (see Section 3.3.2): the first parameters r and i are the qualifying parameters of the event instance, i.e. a robot and the item it might drop. Besides these values, the framework automatically injects fluent accessors in declared keyword arguments, which is used here to access the current value of the grip quality. Based on this value, a parameter for a geometric distribution is chosen from the parameter vector and a random delay is sampled accordingly. Besides returning this delay, which will be used by the simulation engine to schedule the event, a distribution function can also choose to return `None` to signal that this particular event instance should not be scheduled at all at this point.

The concrete parameter vector in `drop_probabilities` that was used for the experiment discussed here was $(0.0, 0.001, 0.01, 0.2)$. This means that the grip value 1 (aka "perfect grip") is assigned to the parameter 0.0, in which case the distribution function returns `None`, corresponding to the idea that a robot would never accidentally drop an item if it has perfect grip. The other three parameter options yield a series of geometric distributions whose cumulative distribution functions (CDF) are shown in Figure 4.7.

The last component of the experiment's configuration is the definition of the initial situation for each simulation trial. As described in Section 3.4.4, this is done in the method `create_initial_situation` of the experiment class, which is called by the framework at the beginning of each simulation run. For this example, all robots are initially placed above each other in distinct rows at x-position 0. Additionally, for each robot, the item with the corresponding id is placed at the same position as the robot, i.e. `item1` at the location of `rob1`, etc. Then, each item is assigned a random destination that is constrained to be at the same y-coordinate as the item and on the right side of it with a distance that is sampled from a uniform distribution ranging from 10 to 50 steps. This setup of the destinations can be seen in Figure 4.8.

When the experiment is configured as described above, it is now almost ready to be used together with SALMA's statistical model checker. What remains to be done is the definition of invariants and goals that should be checked. The property that is going to be examined here is whether all robots are able to deliver the item they have been assigned to its destination within a certain time without dropping the item accidentally. The SALMA-PSL formula that expresses this requirement will be called F and can be found in the top of Figure 4.9.

In the definition of F, the occurrence of a `grab` marks the start of the delivery process that has to be completed within 100 time units. This limit is intentionally chosen high enough so that it cannot be a reason for a trial

```
def setup_distributions(self):
    . . .
    fstep = world.get_exogenous_action("finish_step")
    fstep.config.occurrence_distribution = ConstantDistribution("integer", 1)
    pickup = world.get_stochastic_action("pickUp")
    grab = pickup.outcome("grab")
    grab.map_param("r", "r"), grab.map_param("i", "i")
    grab.set_param_distribution("grip", CategoricalDistribution("integer",
                                   [(1, 0.3), (2, 0.65), (3, 0.04), (4, 0.01)]))

    drop_delay_fn = generate_drop_delay_distribution(
                                   [0.0, 0.001, 0.01, 0.2])

    acc_drop = world.get_exogenous_action("accidental_drop")
    acc_drop.config.occurrence_distribution = \
              CustomDistribution("integer", drop_delay_fn)
    . . .

def generate_drop_delay_distribution(drop_probabilities):
  def accidental_drop_delay(r, i, grip=None, **ctx):
    g = grip(r)
    if g < 1 or g > len(drop_probabilities):
      raise SALMAException(. . . )
    p = drop_probabilities[g - 1]
    if p == 0:
      return None
    else:
      return np.random.geometric(p)
  return accidental_drop_delay
```

Figure 4.6: Setup of probability distributions for the robot example.

failure, which allows concentrating on the possible occurrence of accidental drops. The wild-card symbol `?` is used in the *occur*-predicate to ignore the grip quality argument. Besides this invariant, it is also necessary to declare a stop condition for the simulation by means of an achieve-goal. The intuitive choice in this example is to declare the simulation trial as successful when the invariant has not been violated, all items have been carried to their destinations, and the items have been delivered, i.e. no item is still being carried by any robot. The corresponding goal, named G, can also be seen in Figure 4.9, which actually shows the content of a *property specification file*. Assuming this file is stored under the name `robots01.sspl`, the invariant F and the goal G can be loaded and registered in the experiment's property collection with the following command :

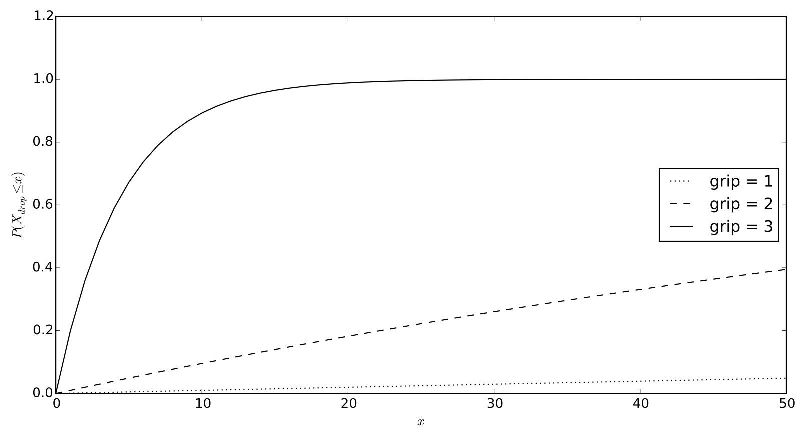


Figure 4.7: CDFs for schedule delays of `accidentalDrop` events, conditioned on the robot's current grip quality.

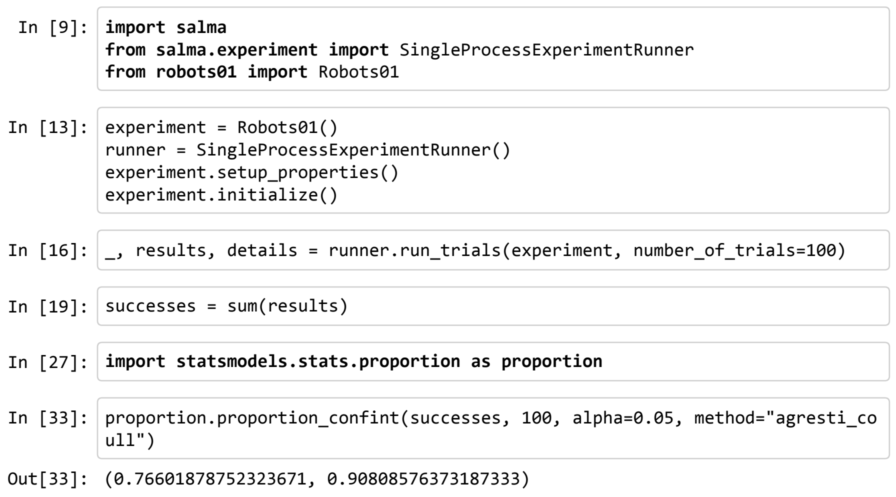
```
...
for i in self.world.getDomain("item"):
    dist = np.random.randint(10, 51)
    self.world.set_constant_value("destX", [i.id],
                                  self.world.get_fluent_value("xpos", [i.id]) + dist)
    self.world.set_constant_value("destY", [i.id],
                                  self.world.get_fluent_value("ypos", [i.id]))
```

Figure 4.8: Setup of the item destinations in the robot example.

```
INVARIANT F: forall(r:robot, forall(i:item,
     implies(occur(grab(r, i, ?)), until(100, carrying(r, i), xpos(i) = destX(i) ))))

GOAL G: forall(i:item,
               and( xpos(i) = destX(i), not(exists(r:robot, carrying(r, i)) ) ) )
```

Figure 4.9: Content of the property specification file `robots01.sspl` that defines the invariant and the goal used in the example.



```
In [9]:  import salma
         from salma.experiment import SingleProcessExperimentRunner
         from robots01 import Robots01

In [13]: experiment = Robots01()
         runner = SingleProcessExperimentRunner()
         experiment.setup_properties()
         experiment.initialize()

In [16]: _, results, details = runner.run_trials(experiment, number_of_trials=100)

In [19]: successes = sum(results)

In [27]: import statsmodels.stats.proportion as proportion

In [33]: proportion.proportion_confint(successes, 100, alpha=0.05, method="agresti_co
         ull")

Out[33]: (0.76601878752323671, 0.90808576373187333)
```

Figure 4.10: Performing a SALMA experiment in the IPython environment.

```
experiment.property_collection.load_from_file("robots01.sspl")
```

At this point, the configuration is complete and the experiment can be performed. This can be done by using an instance of a concrete `ExperimentRunner` class, e.g. `SingleProcessExperimentRunner` (see Section 4.2). With that, a predefined number of trial runs can be performed by a call to the method `run_trials`, which returns a list with boolean results for all trials for which a conclusive verdict could be found. Additionally, a list of dictionaries is passed back that contain detailed information about the trials like the number of performed simulation steps. Based on this data, the user can perform various kinds of statistical analyses. In particular, there is now a well-established collection of mathematical and scientific Python modules that form the so-called scientific Python stack [dev15], which contains implementations of various methods for calculating confidence intervals for proportions or performing a wide range of common hypothesis tests. Alternatively `run_trials` can also be called with an optional argument containing an instance of some subclass of the abstract class `HypothesisTest`. In the current example, SALMA's integrated simple implementation of Wald's sequential probability ration test (SPRT [W$^+$45]) is used, which was introduced in Section 2.5.1. The simulation is repeated only as long as necessary before the hypothesis can be accepted or rejected with the desired error bounds. Called in this way, `run_trials` either returns the number of the accepted hypothesis (0 or 1) or `None` if no definite choice could be made within a maximal number of simulation trials.

A typical example for the use of the SALMA framework together with common scientific Python modules can be seen in Figure 4.10, which shows an excerpt of an interactive session example in the IPython environment [PG07]. IPython provides a read–eval–print loop (REPL) that can be used similar to common mathematical or statistical software packages like Matlab or R. Here, after importing the required modules, the class `Robots01`, which contains the experiment described in this section, was instantiated and initialized. Then, 100 simulation trials were performed with a call to `run_trials`. This returned a tuple with three parts, the accepted hypothesis, which is ignored here since no test was specified, the trial results, and the execution details. The last lines in Figure 4.10 use a function from the StatsModel library [sta15] to calculate a confidence interval for the success probability using the Agresti-Coull method [AC98] that was recommended in a survey by Brown et al. in 2001 [BCD01] for samples larger than 40. In this case, 85 of the 100 performed trials were successful and the calculated confidence interval was $[0.766019, 0.908086]$ at level 0.05.

As mentioned in the beginning of this section, the example presented here allows an exact calculation of the success probability. What makes this possible is the fact that the event probability distributions are set up in a way so that it is known that each robot needs exactly X time steps to reach a target that has a distance of X length units. Since the occurrence times of accidental drop events are modeled by a geometric distribution, the cumulative distribution function (CDF) of the geometric distribution can be used to calculate the probability that a robot drops an item before it reaches its destination. As described above, the parameter of the geometric distribution for the drop events depends on the grip quality, which is reached as an outcome of the stochastic action `pickUp`. Given that the item distances are uniformly distributed between 10 and 50, the probability that one robot drops an item before reaching its destination can be expressed as follows:

$$\begin{aligned}
P(\text{fail-1-robot}) &= \sum_{d=10}^{50}\sum_{g=1}^{4}(P(Dist = d)P(Grip = g)\cdot P(X_{drop} \le d \mid g)) \\
&= \frac{1}{40}\sum_{d=10}^{50}(0.3\cdot(1-(1-0)^d) + 0.65\cdot(1-(1-0.001)^d) + \\
&\qquad 0.04\cdot(1-(1-0.01)^d) + 0.01\cdot(1-(1-0.2)^d)) \\
&= \frac{1}{40}\sum_{d=10}^{50}(0 + 0.65\cdot(1-0.999^d) + 0.04\cdot(1-0.99^d) + \\
&\qquad 0.01\cdot(1-0.8^d)) \\
&\approx 0.0392471
\end{aligned}$$

Given that there are three robots that act independently, the following expression yields the overall probability that a simulation run is a success:

$$P(\text{success}) = (1 - P(\text{fail-1-robot}))^3 \approx \mathbf{0.8868193}$$

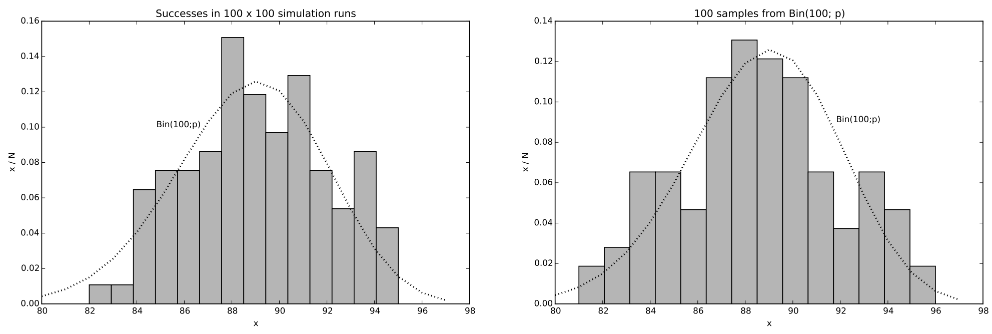
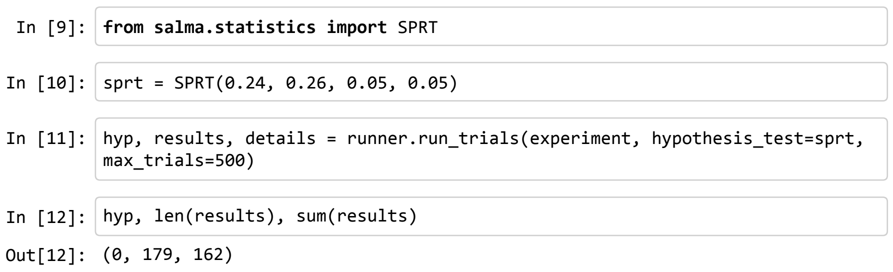


Figure 4.11: Left side: success rates of simulation runs for the Robot experiment, grouped into samples of size 100. Right side: histogram of s sample of size 100 from the binomial distribution with the theoretical probability.

Obviously, the confidence interval that was produced in Figure 4.10 contains the real probability. However, in order to validate the accuracy of the simulation, it is necessary to work with a much larger sample. In fact, the simulation was run for a total of $10,000$ times, grouped in 100 samples of 100 trials. Altogether, $8,916$ of the $10,000$ trials were successful, which, using the Agresti-Coull method, leads to a 0.05 level confidence interval of $[0.885354, 0.897545]$. This is so close to the actual probability of 0.8868193 that it intuitively suggests that both the simulation mechanism and the property evaluation worked correctly. For further comparison, the simulation results can be understood as a sample of size 100 from the binomial distribution $B(100; 0.8868193)$ that describes the theoretically expected number of successes. The left side of Figure 4.11 shows a normalized histogram of this sample, i.e. the success rates. It also contains a line plot that visualizes the probability mass function of the said theoretical distribution. The other histogram on the right side of Figure 4.11 shows a sample of size 100 that was drawn directly from that binomial distribution using the appropriate function from the SciPy package. It can be seen that the deviation from the expected results has a similar magnitude for both cases. This again underpins the impression that the differences between the simulation results and the theoretically expected results are not significant. Finally, this can be verified more formally with a χ^2 goodness of fit test (see e.g. [DS12, Chap. 10.1]) for the null-hypothesis that the batched simulation result is distributed according to $B(100; 0.8868193)$. This test yielded the test statistic 14.373, which corresponds to a p-value of 0.213. This means the null-hypothesis would be accepted for all significance levels up to 0.213.

As mentioned before, SALMA also allows using the sequential probability ratio test (SPRT) to test a composite null-hypothesis of the form $P(failure) \leq P_{max}$, i.e. to confirm that the probability of the registered properties being violated is at most P_{max}. Such a test can be performed with a command sequence like the one shown in Figure 4.12. The test configuration requires four parameters: the first two, named p_0 and p_1, define an *indifference region*



```
In [9]:  from salma.statistics import SPRT

In [10]: sprt = SPRT(0.24, 0.26, 0.05, 0.05)

In [11]: hyp, results, details = runner.run_trials(experiment, hypothesis_test=sprt,
         max_trials=500)

In [12]: hyp, len(results), sum(results)

Out[12]: (0, 179, 162)
```

Figure 4.12: Conducting an sequential probability ration test (SPRT) in the IPython environment.

in the sense that the null hypothesis is only accepted if $P(failure) < p_0$ and only rejected if $P(failure) > p_1$. The remaining parameters specify α and β, the maximal acceptable probabilities for errors of the first and second kind. Instead of being specified by a parameter as before, the number of necessary trials is determined automatically by the SPRT algorithm that continues until a decision can be found that respects the given error bounds. Only a maximal number of 500 trials is specified to avoid unacceptably long run-times. In the example of Figure 4.12, p_0 was set to 0.24, p_1 to 0.26, and both α and β to 0.05. The last line in Figure 4.12 shows the result returned by the call to `run_trials`. As expected, the null-hypothesis was accepted in this case (which is indicated by `hyp` being 0), since the actual failure probability is approximately $1 - 0.8868193 = 0.113181$. The number of trials that were performed before the hypothesis was accepted was 179 and 162 trails were successful.

In order to validate SALMA's implementation of the SPRT algorithm more thoroughly, the test was performed with the same levels for α and β but with different values for p_0 and p_1. In fact, the indifference region was set to a fixed interval of length 0.4 that was centered around a variable probability value p, which was moved up from 0.3 to 0.93 in steps of 0.2. The results of this experiment can be seen in Figure 4.13 where the number of performed trials is plotted against p and the markers indicate whether the null-hypothesis was accepted or rejected, or no definite result could be found within the given maximum number of trials. The dashed vertical line marks the theoretical probability for a failure, i.e. $1 - P(success) \approx 0.113181$ and the horizontal line in the top shows the trial number limit that was set to 500. It can be seen that the null-hypothesis $H_0 : P(failure) < p$ was in fact accepted for all tests where p was lower than 0.11 and accepted when it was higher. For $p = 0.11$, the number of trials exceeded the specified bound of 500, so no decision could be made. It is also clear that there is a strong peak for the number of necessary trials at the position of the actual failure probability. In fact, this coincides exactly with the findings in [W^{+}45] and therefore provides additional evidence for the correctness of the SALMA simulation and property evaluation algorithms.

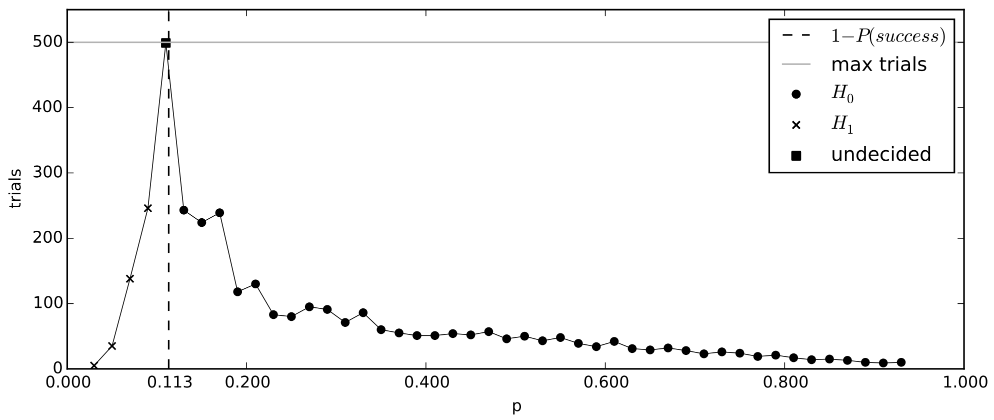


Figure 4.13: Results of iterative executions of the SPRT test for the delivery robots example with $H_0 : P(failure) < p$.

4.4 Summary

This chapter described how the SALMA approach can be used for *statistical model checking*. In doing so, it revealed one of the main contribution of SALMA, which is the seamless integration of a logical *property specification language* into a situation calculus based environment. This language can be seen as a variant of linear temporal logics (LTL), which is a common foundation for many existing statistical model checking approaches. In contrast to classical LTL, the property specification language of SALMA (SALMA-PSL) allows to define detailed expressions in which all elements of the system model can be accessed with first-order predicate logics. The syntax and semantics of the SALMA-PSL were described in this chapter and illustrated with several example formulas. After that, the use of SALMA for statistical model checking was demonstrated by means of an experiment based on the delivery robots example from Chapter 3. This experiment was adapted in a way so that the probability that a simulation run satisfies a given SALMA-PSL property can be calculated exactly. The expected outcome was compared to the results of 10.000 repetitions of the simulation which confirmed that the deviation was not statistically significant. This acts as evidence for the correctness of the approach and its implementation.

4.5 Related Work

A short overview of related work in the field of statistical model checking was already given in Section 2.5. A significant part of the research in SMC has been focused on *statistical issues*, such as the selection of adequate hypothesis tests or estimators, the handling of rare events (e.g. in [CZ11]), or the application of Bayesian method as in [JCL09] . These topics are mainly independent from the way in which the simulation is realized that produces the traces on which the statistical model checkers operate. Hence, they are as relevant for the approach presented in this thesis as they are for any other. On the other hand, this means that solutions which are developed for them in the context of other approaches can also be applied when SALMA is used. As this thesis focuses on the integration of SMC with logic-based modeling and simulation, such general statistical aspects are not discussed further.

What is more relevant from the perspective of this thesis is the design and capabilities of the different property specification languages that are used in statistical model checking. Although almost all SMC solutions adapt the languages they use for modeling and property specification to their specific needs, they are basically all based on the "classical" temporal logics that were shortly introduced in Section 2.4, namely linear temporal logics (LTL), computational tree logic (CTL), the probabilistic CTL variant PCTL and Continuous Stochastic Logic (CSL).

As explained in this chapter, the property specification language that is used by the SALMA framework can be seen as a *first-order predicate logic (FOL)* variant of *bounded LTL (BLTL)*. The term bounded LTL generally describes variants of LTL in which all temporal operators have upper bounds on the number of steps until which . BLTL was discussed very early, e.g. in (see [Pnu77b]), as an obvious variant of LTL. For statistical model checking, it is used, for instance by Clarke and Zuliani in their statistical model checking approach presented in [CZ11], which focuses on cyber-physical systems. To the author's knowledge, there is currently no other implemented SMC solution that supports FOL. However, there have been attempts to lift exact model checking to first-order temporal logics. For instance, in [BDG$^+$98], the authors use a symbolic model checking algorithm to verify a Binary Decision Diagram based representation of a *first-order Kripke structure*. Respectively, a model-checker for first-order LTL was described in [WTM04]. There, the system model is specified as an abstract state machine (ASM), and the requirements as formulas in a first-order variant of LTL. These formulas are then automatically transformed to propositional LTL formulas and checked against the system model using a method that is adapted from the classical automata-based LTL model checking approach by Vardi [Var96].

PCLT and CSL have been used originally in exact stochastic model checking tools like PRISM [HKNP06]. For statistical model checking, both have been adopted, for instance, by Ymer [You05b] and VESTA [SVA05b], and by PRISM itself, whose latest version includes an extension for statistical model checking. Since CSL is based on a continuous time model, it obviously differs from both PCTL and the property specification language that is used in SALMA. However, PCTL and CSL have in common that they provide *prob-*

abilistic operators that reason about the probability with which a subformula is fulfilled.

The SALMA property specification language does not include probabilistic operators and therefore only allows reasoning about the probability of top-level formulas by means of hypothesis tests or estimation. The reason why nestable probabilistic operators were not included in SALMA is mainly because of the immense simulation effort that could be necessary when formulas with such nested multi-level structures are tested. For example, suppose that the SALMA verification engine had to test a PCTL formula like this:

$$\Psi = (P_{\leq 0.05}\Phi_1)\, U_{\leq 100}\, \Phi_2$$

Without anticipating details of the actual evaluation algorithm that is used in SALMA, which will be explained elaborately in Chapter 5, it can be said that in order to determine whether the subformula $(P_{\leq 0.05}\Phi_1)$ holds in the current state, it would be necessary conduct a full nested hypothesis test for the formula Φ_1. This means that for every visited step during the *outer simulation* in which the top-level formula Ψ is tested, the engine would have to spawn a batch of independent simulations that start from the current state and test the property Φ_1. Hence, the total number of simulations that are necessary to determine a result for the top-level formula could in the worst case grow exponentially with the number of nesting levels.

In spite of this obvious problem, all of the established statistical model checkers that were mentioned in Section 2.5 support nested probabilistic operators. However, the authors of these tools mainly address another issue, namely how the statistical methods they use have to be adapted to take into account that the result of the evaluation of a subformula with probabilistic operators is a random variable itself (see [SVA04, sec. 3.2], [YS02, sec. 4.1]). In [YS02, sec. 4.1]), the authors shortly mention the problem of having to conduct full hypothesis tests for each nesting level. They also point out that the nesting level is not required in most practical scenarios ([YS02, sec. 5])). As an obvious solution for mitigating the growth of the number of required samples, memoization is suggested in [YS02] and [You05a], i.e. storing results of nested evaluations ad reusing them when the same start state is visited again. This idea is extended in [YKNP06] by combining statistical model checking for the outer formulas with numerical stochastic model checking for the nested subformulas. Both solutions would be difficult to integrate in SALMA since the state space is so large that exact numerical methods are not applicable and it is rather unlikely that the exact same state is ever reached again. Altogether, it has to be said that a proper solution for handling nested probabilistic operators in SALMA has yet to be found. In fact, the discussion will return to this issue shortly when an outlook for the thesis is presented in Chapter 7.3.

Another typical feature of temporal logics that is missing in the SALMA property specification language is the support for unbounded temporal operators. In fact, this is a topic that has been addressed time and again within the statistical model checking community. The obvious challenge for any simulation-based approach is that it may not be possible in acceptable (or even finite) time to find a trace prefix that invalidates or confirms the checked properties. There have been different approaches to deal with this problem. In “Monte Carlo Model Checking” [GS05], the structure of the model allows using automata-based techniques for finding *loops* in a very similar way as in classical model checkers. However, this is not possible for the more general cases of statistical model checking where structure of the model is not as accessible. In [SVA05a], the authors propose a method that involves the use of a modified model in which an additional outgoing failure transition with the same non-zero probability is added to each state, i.e. transitions that lead to an absorbing failure state. By this, every simulation run will eventually come to an end. The authors then describe how to use hypotheses tests on this modified model for testing within some given error bounds that an unbounded path formula is *not* fulfilled. However, the justification for their approach is partly based on a lower bound for the success probability of a simulation trace that is proportional to $(1-p_s)^N$ where N is the number of states of the system. For the type of FOL models used in SALMA, where the number of possible states is measured by the product of the sizes of all fluent domains. Therefore, the state space will effectively be so large that this lower bound converges towards 0 and thus becomes useless. Since this problem is so universal, it is not surprising that, for instance, PRISM uses a much more pragmatic solution by simply limiting the simulation length (cf. [pri16]). On the other hand, it is unclear how problematic this restriction actually is. Intuitively, it might be argued that for cases in which even a very high limit on the trace lengths is not acceptable, statistical model checking itself might be not exact enough.

Chapter 5

Efficient Evaluation of FO-BLTL Properties

Once invariants and goals are formulated using SALMA's property specification language, they are registered with the simulation engine and the experiment can be stared. All registered properties are then evaluated *"on the fly"* alongside simulation and each simulation run is stopped as soon as a conclusive verdict for the configured combination of invariants and goals is found. Since statistical model checking in general requires a large number of simulation runs, one of the key requirements for SALMA's property evaluation module is the ability to find verdicts as soon as possible in order to avoid running simulations longer than necessary. At the same time, the evaluation mechanism has to be scalable with respect to the complexity of the formulas, the number of entities in the system, and the length of the history. This chapter describes how this is achieved in SALMA through a combination of optimized data structures and algorithms.

5.1 Overview of the Evaluation Mechanism

Figure 5.1 summarizes SALMA's property evaluation mechanism. At first, the properties specified in SALMA-PSL are translated by the *formula compiler* into an internal representation that can be interpreted by the Prolog-based evaluation algorithm. Section 5.4 explains this step. The compiled formulas are added to a *property registry* from which they are retrieved and evaluated in each simulation step. As soon as the evaluation of a property returns a conclusive verdict (positive or negative), it is returned to the simulation engine which includes it in the evaluation statistics and in the arbitration of the overall result for the simulation run. However, if a property's formula contains temporal operators, it cannot always be determined immediately whether a property holds at a given time point or not. In this case, an entry in the

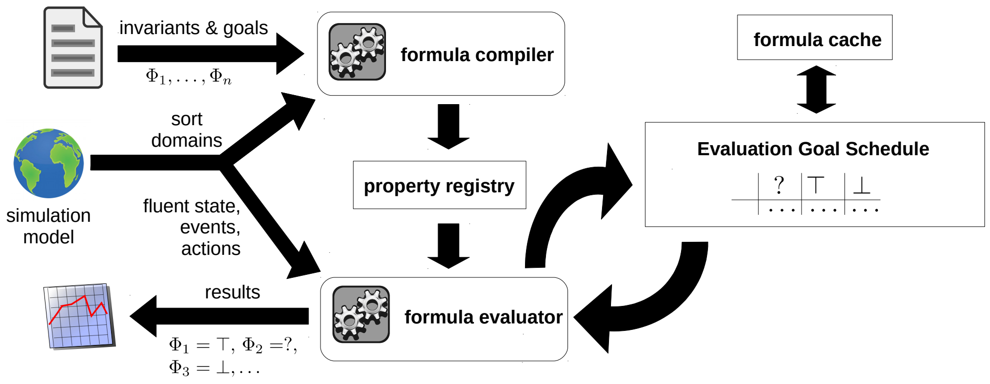


Figure 5.1: Property evaluation overview.

evaluation goal schedule is created that marks the property for being re-visited in the following steps. Section 5.5 explains this scheduling mechanism in detail and describes how it uses term rewriting, a *formula cache*, and a transitive reference model to support formulas with multiple levels of nested temporal operators. However, before these more technical details can be revealed, it is necessary to discuss several core concepts of SALMA's evaluation mechanism.

5.2 Formula Evaluation With Variable Time Advances

Besides the support for complex requirement specifications, one of the main aspects in the design of SALMA's property evaluation module is the support for simulations with *variable time advances*. As described in Section 3.6, the simulation engine uses a priority queue in which events are scheduled at time points that are basically chosen according to probability distributions that model their delays [1]. Since probabilistic effects can only occur through events, the simulated system follows a deterministic state trajectory during the time between two events. Therefore, from the perspective of the simulation, it is often not necessary to calculate the world state for every time step but only for those where events occur. In this case, the *time is advanced* and the time-dependent fluents are updated by a single progression step (cf. Definition 3.28 in Section 3.6.2). As explained in Theorem 3.1, this optimization is possible when the domain model of the simulated system fulfills the *time-advance-stability* property that was introduced in Definition 3.29. In contrast, when this property is not satisfied, then the simulation has to explicitly visit every step between the current and the next event, i.e. the simulation works with a *constant time advance* of 1.

[1] In fact there are also slightly different scheduling schemes that are based on ad-hoc event selection rather than delays (see Section 3.6)

However, even if the simulated model is time-advance-stable, the period between the current and the next event can generally not be ignored when the simulation traces are used as input for the evaluation of goals and invariants. In fact, it is not always possible without direct inspection to determine whether a property holds at a specific time. For instance, a simplified kinematic model of a multi-robot system could contain the fluents shown in Figure 5.2. Since the velocity fluents v_x and v_y can obviously only change through an intentional action, they have to remain constant during a time advance period. Therefore, it is easy to see that pos_x and pos_y are *time-advance-stable*:

Lemma 5.1. *The fluents pos_x and pos_y in Figure 5.2 are time-advance-stable.*

Proof. Let $\Delta t \in \mathbb{N}_0$ be the total delay of the time advance between the current event end the next scheduled event. Additionally, let $t_1, \ldots, t_n$ be a partition of Δt, i.e. $\Delta t = \sum_{i=1}^{n} t_i$. Since by definition no event or action other than *tick* can occur during a time advance period, the value of any instance of the fluents v_x and v_y have to remain constant and we can substitute them with a constant v_x or v_y, respectively. To prove that pos_x and pos_y are time-advance-stable, we have to show that the value for any of their instances is the same both for a situation term with the single action $tick(\Delta t)$ and for a situation term that consists of the action sequence of $(tick(t_1), \ldots, tick(t_n))$. Since the definitions of pos_x and pos_y are symmetrical, it is enough to show this for pos_x:

$$\begin{aligned}
&pos_x(r, do(tick(t_n), do(tick(t_{n-1}), do(\ldots, do(tick(t_1), S_0)))))\\
&\quad = pos_x(r, do(tick(t_{n-1}), do(\ldots, do(tick(t_1), S_0)))) + v_x t_n\\
&\quad = pos_x(r, do(tick(t_{n-2}), do(\ldots, do(tick(t_1), S_0)))) + v_x t_{n-1} + v_x t_n\\
&\quad = \ldots = pos_x(r, S_0) + v_x t_1 + \ldots + v_x t_n = pos_x(r, S_0) + v_x \sum_{i=1}^{n} t_i\\
&\quad = pos_x(r, S_0) + v_x \Delta t = pos_x(r, do(tick(\Delta t), S_0))
\end{aligned}$$

□

When the simulation advances from the current time t_{now} to the time of the next scheduled event t_{next}, it can simply perform a single *progression* with $tick(t_{next} - t_{now})$ to update the world state. However, it is not far-fetched to assume that the specification for the system model might contain an invariant that requires robots to have a certain minimum distance, i.e.

F = **forall**(r1 : robot, **forall**(r2 : robot, **implies**(r1 \= r2, dist(r1, r2) > 5)))

. Where `dist` is a *derived fluent* that is defined as the euclidean distance:

$$\begin{aligned}
&dist(r_1, r_2, s) = d \equiv\\
&\quad d = \sqrt{(pos_x(r_2, s) - pos_x(r_1, s))^2 + (pos_y(r_2, s) - pos_y(r_1, s))^2}
\end{aligned}$$

$$pos_x(r, do(a, s)) = x \equiv ((\exists t \in \mathbb{N}_0.\, a = tick(t)) \wedge \\ x = pos_x(r, s) + v_x(r, s)t) \vee ((\nexists t.\, a = tick(t)) \wedge x = pos_x(r, s))$$

$$pos_y(r, do(a, s)) = y \equiv ((\exists t \in \mathbb{N}_0.\, a = tick(t)) \wedge \\ y = pos_y(r, s) + v_y(r, s)t) \vee ((\nexists t.\, a = tick(t)) \wedge y = pos_y(r, s))$$

$$v_x(r, do(a, s)) = v \equiv a = setV_x(r, v) \vee \\ ((\nexists v'.a = setV_x(r, v')) \wedge v_x(r, s) = v)$$

$$v_y(r, do(a, s)) = v \equiv a = setV_y(r, v) \vee \\ ((\nexists v'.a = setV_y(r, v')) \wedge v_y(r, s) = v)$$

Figure 5.2: Excerpt from example model with deterministic trajectories between events.

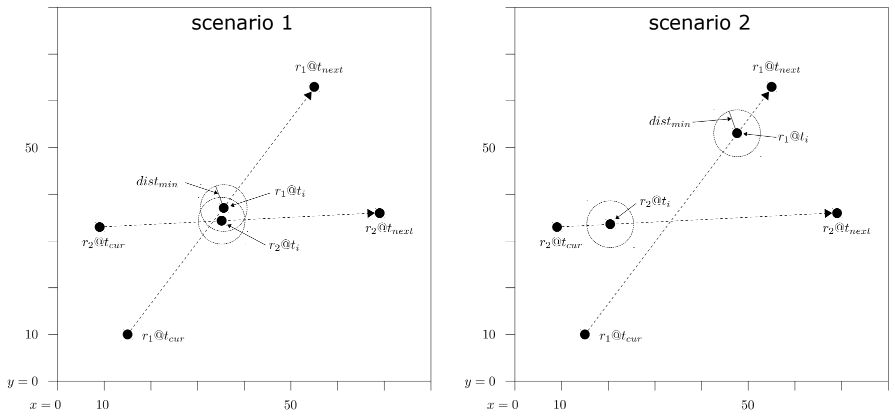


Figure 5.3: Example for robot movement during time advance.

When the property F is registered as an *invariant*, then the evaluation module has to check that the property holds in every time step between the current and the next event. This fact is clearly visible in Figure 5.3 in which the trajectories of two robots, r_1 and r_2, are sketched. Although information about the start and end coordinates is enough to tell that the paths of both robots have crossed, it is not clear whether they were actually closer than their minimum distance at any point in time.

In terms of the situation calculus, this means that the property has to be evaluated for every situation in which the time is between t_{cur} and t_{next}. However, since the model is time-advance stable and it is known that no event or action occurs between t_{cur} and t_{next} except *tick*, it is not necessary to perform full *progression* for the intermediate steps. Instead, it is possible to use a more efficient evaluation scheme based on regression, i.e. by substitution of the situation arguments in the formula itself. This is shown below in Theorem 5.1.

Theorem 5.1. *Let Φ be a SALMA-PSL formula and let $\mathcal{M}$ a system model that is time-advance-stable. Additionally, let t_{cur} and t_{next} be the time of the current simulation step and the time at which the next event is scheduled for the current simulation run. Furthermore, let S represent a situation and $[\![\Phi]\!]_S$ denote the evaluation of Φ for situation S. Then the following equivalence holds:*

$$\begin{aligned}&\forall S.\,\forall \Delta t \in \mathbb{N}_0.\,(\Delta t \leq t_{next} - t_{cur}) \land (time(S) = time(S_0) + \Delta t) \implies \\ &\quad [\![F]\!]_S \equiv [\![F]\!]_{do(tick(\Delta t),S_0)}\end{aligned} \tag{5.1}$$

Proof. By construction of the simulation algorithm we know that no other event or action other than *tick* can occur between t_{cur} and t_{next}. In terms of Definition 3.29, this means that $S \in \mathcal{S}_{ta}$. Therefore, since $\mathcal{M}$ is time-advance-stable, we know by Definition 3.29 that for all fluent instances, the value does not depend on how the situation term responsible for the time advance is constructed. In particular, this holds for all fluent instances that are included in the evaluation of Φ. Since all other elements of Φ have to be constant anyway, this means that the evaluation result $[\![F]\!]_S$ itself is independent from the construction of the situation term S. Therefore, the equivalence in the right side of the implication in (5.1) holds, which proves the theorem. □

The consequence of the theorem is that the evaluation algorithm can use a loop that simply increases the delay Δt step by step in order to evaluate Φ for all time steps between t_{cur} and t_{next}, which will be explained further in Section 5.6.3. The most important aspect of this design is that, in many cases, it is able to reduce the evaluation effort significantly. More precisely, for each of the steps in the loop mentioned before, only those fluent instance values are calculated that are really needed to evaluate the formula. This of course is a direct result of the recursive nature of the situation-calculus-based axioms in which the rules for fluent updates are specified in SALMA. In fact, depending on the structure of Φ, only a very small partition of the state space has to be calculated. First of all, there might be many fluents that are neither directly nor transitively referenced by any formula but only involved in agent control procedures. Additionally, only a few instances of a fluent might be relevant for a formula. In particular, invariants and goals are often conditioned, like in the formula below that requires robots that are involved in a transmission to stay within a certain maximum range.

```
G = forall(r1 : robot, forall(r2 : robot, implies(
        and(r1 \= r2, transmitting(r1, r2)),
        dist(r1, r2) < maxTransmissionRange)))
```

Here, the fluent `dist` only has to be evaluated for pairs of robots that are involved in a transmission since all other pairs will be filtered out by the implication.

Altogether, the SALMA's property evaluation mechanism is designed to leverage the obvious advantage of the event scheduling paradigm as far as possible, which is avoiding the calculation of unnecessary parts of the state space along deterministic trajectories. This can potentially provide a great efficiency advantage in comparison to *procedural* multi-agent simulation approaches where the state variables are updated directly by the agent control functions in every step. The next sections will explain the most important concepts, data structures, and algorithms that contribute to this design.

5.3 Discrete Temporal Interval Sequences

One of the most important aspects of SALMA's property evaluation algorithm is the way in which the points in time that are relevant for the evaluation goal schedule are represented. This topic will mainly be covered in Section 5.5. However, first it is necessary to introduce the basic data structures on which the algorithms in this chapter are based: sequences of discrete time intervals and result mappings, which add labels to interval sequences that represent evaluation states.

5.3.1 Unlabeled Temporal Interval Sequences

First, since a discrete time base is assumed, a temporal interval is merely a closed interval on natural numbers.

Definition 5.1 (Discrete temporal interval)**.** A temporal interval is a closed interval $[t_s, t_e]$ with $t_s \in \mathbb{N}_0$ and $t_e \in \mathbb{N}_0$. Furthermore, the set of all possible temporal intervals is denoted by $\mathcal{I}_T$.

In the further discussion within this chapter, temporal intervals will be used to reflect the evaluation state of a given property with respect to the time, i.e. at which time points the property was evaluated to true or false, and for which points a concise result has not been found yet. Since the state of a property will change in the course of events, it is necessary to represent time trajectories that contain gaps. This is achieved by aggregating intervals in sorted sequences.

Definition 5.2 (Discrete Temporal Interval Sequence)**.** A discrete temporal interval sequence, written like $\overbracket{T}$, is a sorted sequence of disjunctive closed intervals of natural numbers. Formally, $\overbracket{T} : \mathbb{N}_0 \to \mathcal{I}_T = ([t_{i,s}, t_{i,e}])_{i=0}^{N}$ where the following invariants hold:

1. $\forall 0 \leq i \leq N.\ t_{i,s} \in \mathbb{N}_0$ and $t_{i,e} \in \mathbb{N}_0$.
2. $\forall 0 \leq i \leq N.\ t_{i,s} \leq t_{i,e}$.
3. $\forall 1 \leq i \leq N.\ t_{i,s} > t_{i-1,e}$.
4. $\forall 0 \leq i \leq N-1.\ t_{i,e} < t_{i+1,s}$.

For some of the following definitions and theorems, it is useful to introduce some additional terminology and notation that facilitate referring to certain properties of temporal interval sequences.

Definition 5.3. Let $\overbracket{T}$ be a temporal interval sequence. Then the *cardinality* of $\overbracket{T}$, written as $|\overbracket{T}|$ is defined as the number of temporal intervals in $\overbracket{T}$. This means that for $\overbracket{T} = (I_i)_{i=0}^{N} = (I_0, \ldots, I_N)$, the cardinality is $|\overbracket{T}| = N + 1$.

Furthermore, when $\overbracket{T} = (I_i)_{i=0}^{N}$, then the *set of temporal intervals* in $\overbracket{T}$, written as $\mathcal{I}_T(\overbracket{T})$, is defined as

$$\mathcal{I}_T(\overbracket{T}) \subset \mathcal{I}_T := \{I_i \,|\, 0 \leq i \leq |\overbracket{T}| - 1\}$$

In the definition of temporal interval sequences, properties 1 and 2 simply state that the elements of the sequence are valid discrete temporal intervals, i.e. intervals on natural numbers. More importantly, properties 3 and 4 together establish that the intervals are sorted in ascending order and are *disjunct*, i.e. they don't overlap. Since these invariants are crucial for the evaluation algorithms, it will be necessary below to show that they are maintained by all used operations on temporal interval sequences.

Figure 5.4 shows an example for a temporal interval sequence. Later in this chapter, it will be described how sequences like that are used to represent time intervals that share a common category. For instance, the intervals in Figure 5.4 could reflect the times when a given property holds.

Figure 5.4: Example discrete temporal interval sequence.

Without going into details about how the evaluation algorithm actually creates such interval-based property state representations, it is clear that first of all, a *constructor* is required to build temporal interval sequences. More precisely, an operator is needed that adds a temporal interval to an interval sequence while maintaining the invariants of Definition 5.2.

Definition 5.4 (Addition of an interval to a temporal interval sequence)**.** Let $\overleftrightarrow{T} = ([t_{i,s}, t_{i,e}])_{i=0}^{N}$ be a temporal interval sequence and $I = [t_s, t_e]$ a single closed temporal interval. Then the addition of $\overleftrightarrow{T}$ and I, written as $\overleftrightarrow{T} \oplus I$ is defined as follows:

$$\overleftrightarrow{T} \oplus I = \left(I'_i\right)_{i=0}^{N'}$$

where

$$I'_i = \begin{cases} [t_{i,s}, t_{i,e}] & \text{if } i \leq i^< \\ [t'_s, t'_e] & \text{if } i = i^< + 1 \\ [t_{r(i),s}, t_{r(i),e}] & \text{if } i^< + 2 \leq i \leq N' \end{cases}$$

with

$$i^< = \begin{cases} \max\{i \mid t_{i,e} < t_s\} & \text{if } \exists i.\, t_{i,e} < t_s \\ -1 & \text{otherwise} \end{cases}$$

$$i^> = \begin{cases} \min\{i \mid t_{i,s} > t_e\} & \text{if } \exists i.\, t_{i,s} > t_e \\ N+1 & \text{otherwise} \end{cases}$$

$$r : \mathbb{N}_0 \to \mathbb{N}_0, r(i) = i + i^> - i^< - 2$$

$$N' = i^< + 2 + N - i^>$$

$$t'_s = \begin{cases} \min\left(t_s, t_{k,s}\right) & \text{if } \exists k.\, k = \min\{i \mid t_s \leq t_{i,e} \leq t_e\} \\ t_s & \text{otherwise} \end{cases}$$

$$t'_e = \begin{cases} \max\left(t_e, t_{l,e}\right) & \text{if } \exists l.\, l = \max\{i \mid t_s \leq t_{i,s} \leq t_e\} \\ t_e & \text{otherwise} \end{cases}$$

A schematic overview of the calculations in Definition 5.4 is shown in Figure 5.5. Unlike a regular insertion of an element into a list, the operator $\oplus$ not only has to maintain the order of the sequence but also potentially merge overlapping intervals to guarantee that the resulting sequence reflects the intuitive addition semantics and fulfills the conditions stated in Definition 5.2. This is established in the following theorem.

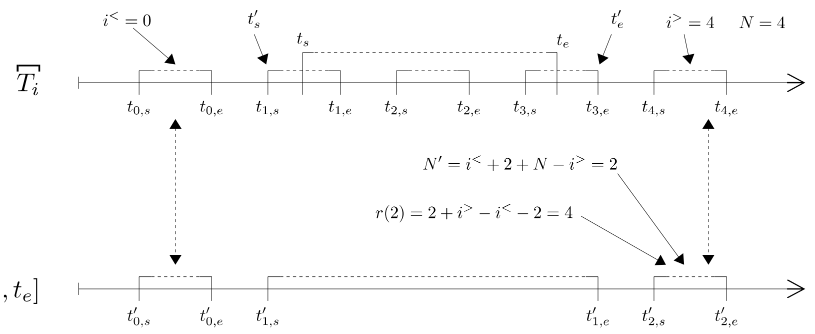


Figure 5.5: Addition of an interval to an interval sequence.

Theorem 5.2. *Let $\overset{\sqcap}{T} = ([t_{i,s}, t_{i,e}])_{i=0}^{N}$ be a temporal interval sequence for which the properties 1. to 4. of Definition 5.2 hold. Furthermore, let $I = [t_s, t_e]$ be a single closed temporal interval and $\overset{\sqcap}{T'} = \overset{\sqcap}{T} \oplus I$ the sequence that results from the addition of I to $\overset{\sqcap}{T}$. Then, the intervals in $\overset{\sqcap}{T'}$ cover all intervals from $\overset{\sqcap}{T}$ and I and satisfy properties 1. to 4. of Definition 5.2.*

Proof. The proof works by case analysis of all possible constellations regarding the temporal relation between the interval I and the sequence $\overset{\sqcap}{T}$:

i) I is strictly before $\overset{\sqcap}{T}$, i.e. $t_e < t_{0,s}$. The ordering constraints of Definition 5.2 imply $\forall i. t_{i,e} > t_{i,s} > t_e > t_s$. This yields $i^< = -1$, $i^> = 0$, $t'_s = t_s$ and $t'_e = t_e$. Therefore, $r(i) = i - 1$ and $N' = N + 1$. Altogether, this results in the intended temporal interval sequence $\overset{\sqcap}{T'} = ([t_s, t_e], [t_{0,s}, t_{0,e}], \ldots, [t_{N,s}, t_{N,e}])$. This means that all previous intervals as well as I are contained in $\overset{\sqcap}{T'}$ and the order of the original elements of $\overset{\sqcap}{T}$ is not modified. With $t_s < t_e < t_{0,s}$, all properties of Definition 5.2 hold.

ii) I starts before or within the first interval and ends before the start of the last interval. Then, as above, $i^< = -1$. Let c be the number of intervals that are fully covered by I. This means that $i^> = c$, $N' = -1+2+N-c = N-c+1$, and $r(i) = i+c-1$. Thus, $\overset{\sqcap}{T'} = ([t'_s, t'_e], [t_{c,s}, t_{c,e}], \ldots, [t_{N,s}, t_{N,e}])$. This immediately shows that properties 1. to 4. of Definition 5.2 are satisfied. Since t'_s and t'_e are defined so that $[t'_s, t'_e]$ is the smallest interval that contains I and all intervals that are partially or completely overlapped by I, it holds that $\overset{\sqcap}{T'}$ covers all intervals as intended.

iii) I is strictly between two intermediate intervals. In this case, there is a left part and a right part of the old sequence that are not altered internally and simply arranged before and after I. For the index transformation defined above, this means that $i^> = i^< + 1$, which implies $r(i) = i - 1$ and $N' = N + 1$. This corresponds to a valid construction of an interval sequence and therefore all required properties hold.

iv) I overlaps one or several intermediate intervals. This means that basically the same considerations apply as in case ii. Additionally, like before, the left and right surrounding sequence parts are not altered and therefore the required properties hold.

v) I overlaps or is adjacent to the last interval of $\overline{T}$. Again, it is easy to see that this case is symmetrical to ii so the same arguments apply.

vi) I is strictly after the last interval of $\overline{T}$, i.e. $t_e < t_{0,s}$. This means that $i^< = N$, $i^> = N + 1$, $t'_s = t_s$ and $t'_e = t_e$. Therefore, $r(i) = i - 1$ and $N' = N + 1$. As before, the construction corresponds to the requirements in Definition 5.2.

$\square$

Based upon the interval addition operator from above, it is possible to inductively define the union of two temporal interval sequences.

Definition 5.5 (Union of temporal interval sequences). Let $\overline{T} = (I_i)_{i=0}^{N}$ and $\overline{T'} = (I'_i)_{i=0}^{N'}$ be two temporal interval sequences. Furthermore, let $(I'_i)_{i=k}^{l}$ denote the subsequence of $(I'_i)_{i=0}^{N'}$ from index k to l. Then the union of $\overline{T}$ and $(I'_i)_{i=0}^{N'}$ is recursively defined as follows:

$$\overline{T} \cup \left(I'_i\right)_{i=k}^{l} := \begin{cases} (\overline{T} \oplus I'_k) \cup (I'_i)_{i=k+1}^{l} & \text{if } k < l \\ \overline{T} \oplus I'_k & \text{otherwise} \end{cases}$$

The union operator simply works by successively adding all intervals from one sequence to another sequence. Since this is recursively based on the interval addition operation from Definition 5.4, it is easy to show that temporal interval sequences are closed under union, i.e. the union of two temporal interval sequences also maintains the invariants from Definition 5.2. However, the proof necessarily relies on the fact that subsequences of temporal interval sequences maintain these invariants, which is shown in the following lemma.

Lemma 5.2. *Let $\overline{T} = (I_i)_{i=0}^{N}$ be a temporal interval sequence and let $(I_i)_{i=k}^{l}$ with $0 \le k \le l \le N$ denote the subsequence of $\overline{T}$ from index k to l. Then $(I_i)_{i=k}^{l}$ fulfills the invariants from Definition 5.2.*

Proof. Since no interval bounds are changed, properties 1 and 2 cannot be violated in any way. Furthermore, the subsequence $(I_i)_{i=k}^{l}$ can be seen as a sequence $(I'_i)_{i=0}^{l-k}$ where $I'_i = I_{i+k}$, i.e. $I_i = I'_{i-k}$. From Definition 5.2 we know that $\forall\, 1 \le i \le N.\ t_{i,s} > t_{i-1,e}$ and $\forall\, 0 \le i \le N-1.\ t_{i,e} < t_{i+1,s}$. In particular, this implies $\forall\, k+1 \le i \le l.\ t_{i,s} > t_{i-1,e}$ and $\forall\, k \le i \le l-1.\ t_{i,e} < t_{i+1,s}$. With the equivalence from above, this can be rewritten to $\forall\, k+1 \le i \le l.\ t'_{i-k,s} > t'_{i-k-1,e}$ and $\forall\, k \le i \le l-1.\ t'_{i-k,e} < t'_{i-k+1,s}$. Simple index transformation finally leads to $\forall\, 1 \le i \le l-k.\ t'_{i,s} > t'_{i-1,e}$ and $\forall\, 0 \le i \le l-k-1.\ t'_{i,e} < t'_{i+1,s}$, respectively. These two expressions correspond to the instantiation of property 3 and 4 of Definition 5.2 for the sequence $(I'_i)_{i=0}^{l-k}$, which proves the lemma. $\square$

Using Lemma 5.2, it can be shown that the temporal interval sequence invariants are maintained by the union operator.

Theorem 5.3. *Let $\overrightarrow{T} = (I_i)_{i=0}^{N}$ and $\overrightarrow{T'} = (I'_i)_{i=0}^{N'}$ be two temporal interval sequences and let $\overrightarrow{T''} = \overrightarrow{T} \cup \overrightarrow{T'}$ be the union of $\overrightarrow{T}$ and $\overrightarrow{T'}$. Then the invariants defined in Definition 5.2 hold for $\overrightarrow{T''}$.*

Proof. The proof works by induction over the length of $\overrightarrow{T'}$, i.e. $N'+1$. If $\overrightarrow{T'}$ is empty, $\overrightarrow{T''} = \overrightarrow{T}$, which fulfills the invariants by definition. Therefore, let the induction start with a length of 1, i.e. $\overrightarrow{T'} = (I'_i)_{i=0}^{0}$. In this case, $\overrightarrow{T''} = \overrightarrow{T} \oplus I'_0$, which fulfills the invariants due to Theorem 5.2.

For the induction step, let $\overrightarrow{T'} = (I'_i)_{i=0}^{N'}$ be a sequence with a length greater than 1, i.e. $N' > 0$. Then, $\overrightarrow{T''} = (\overrightarrow{T} \oplus I'_0) \cup (I'_i)_{i=1}^{N'}$. Due to Theorem 5.2, the left part of this union is an admissible interval sequence that fulfills the invariants. Additionally, Lemma 5.2 shows that right part also fulfills the invariants. By a simple index transformation, the subsequence $(I'_i)_{i=1}^{N'}$ can be rewritten to $(I'_i)_{i=0}^{N'-1}$, whose length is one less than the length of $(I'_i)_{i=0}^{N'}$. Therefore, by the induction hypothesis, a union with $(I'_i)_{i=0}^{N'-1}$ maintains the invariants, which concludes the proof. $\square$

5.3.2 Intersection Operators

As Section 5.6 will show, one of the most essential mechanism in the property evaluation algorithm is the selection of intervals within specific regions and the re-labeling of the selected segments with updated evaluation states. In terms of temporal interval sequences as introduced above, this mainly amounts to calculating intersections between intervals and interval sequences. First, an intersection between two intervals is defined in the usual way.

Definition 5.6 (Intersection of two intervals)**.** Let $I_1 = [t_{1,s}, t_{1,e}]$ and $I_2 = [t_{2,s}, t_{2,e}]$ be two discrete temporal intervals. Then the intersection between I_1 and I_2 is defined as follows:

$$I_1 \cap I_2 = \begin{cases} \emptyset & \text{if } t_{1,e} < t_{2,s} \vee t_{2,e} < t_{1,s} \\ [\max(t_{1,s}, t_{2,s}), \min(t_{1,e}, t_{2,e})] & \text{otherwise} \end{cases}$$

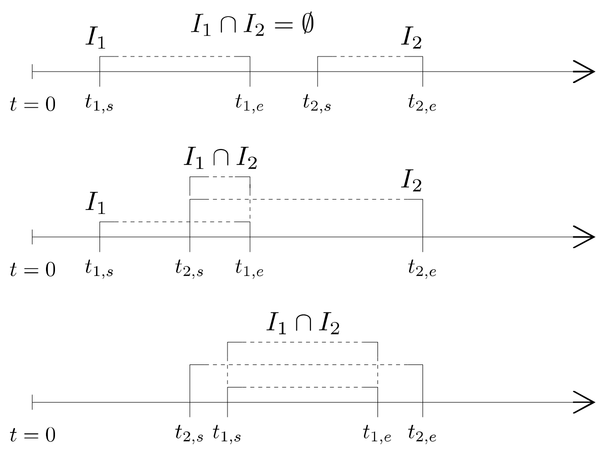


Figure 5.6: Intersection between intervals.

Figure 5.6 shows three different cases for an intersection between two intervals. The other possible arrangements of I_1 and I_2 are symmetric and are therefore covered by the maximum and minimum operators in Definition 5.6.

When an interval intersection is used to update the evaluation schedule, it is almost always also important to process the remainder of the intersection, i.e. the parts of the original intervals that are not included in the intersection. For the purpose of the algorithms of this chapter, it makes sense to view one of the two intervals in an intersection as the *query interval* and one as the *manipulated interval.* In many cases, only the remaining part of the manipulated interval has to be processed further. This remainder of an interval intersection is defined as the *relative complement* of the intervals in the set theoretic sense.

Definition 5.7 (Relative complement of two intervals)**.** Let $I_1 = [t_{1,s}, t_{1,e}]$ and $I_q = [t_{2,s}, t_{2,e}]$ be two discrete temporal intervals. Then the relative complement of I_2 in I_1, written as $I_1 \setminus I_2$ yields a temporal interval sequence that is defined as follows:

$$I_1 \setminus I_2 = \begin{cases} (I'_i)_{i=0}^{0} \text{ with } I'_0 = I_1 & \text{if } t_{1,e} < t_{2,s} \vee t_{2,e} < t_{1,s} \\ (I'_i)_{i=0}^{0} \text{ with } I'_0 = [t_{1,s}, t_{2,s} - 1] & \text{if } t_{1,s} < t_{2,s} \leq t_{1,e} \wedge t_{2,e} \geq t_{1,e} \\ (I'_i)_{i=0}^{0} \text{ with } I'_0 = [t_{2,e} + 1, t_{1,e}] & \text{if } t_{2,s} \leq t_{1_s} \wedge t_{1,s} \leq t_{2,e} < t_{1,e} \\ \emptyset & \text{if } t_{2,s} \leq t_{1,s} \wedge t_{2,e} \geq t_{1,e} \\ ([t_{1,s}, t_{2,s} - 1], [t_{2,e} + 1, t_{1,e}]) & \text{if } t_{1,s} < t_{2,s} \wedge t_{1,e} > t_{2,e} \end{cases}$$

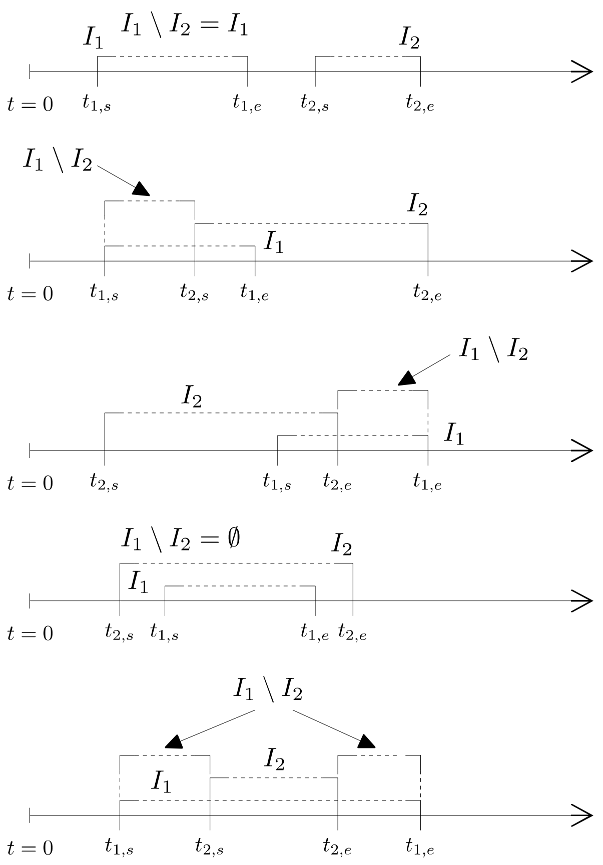


Figure 5.7: Relative complement of two intervals.

The calculation of the relative complement is visualized in Figure 5.7, where all cases from Definition 5.7 are shown. It is easy to see that the required properties for interval sequences are fulfilled by the complement sequence. This is stated as a simple theorem for later reference.

Theorem 5.4. *Let $I_1 = [t_{1,s}, t_{1,e}]$ and $I_q = [t_{2,s}, t_{2,e}]$ be two discrete temporal intervals. Then $I_1 \setminus I_2$ yields a temporal interval sequence that fulfills the invariants stated in Definition 5.2.*

Proof. The first two invariants from Definition 5.2, namely that all interval bounds have to be in $\mathbb{N}_0$ and that each interval has to contain at least one time point (i.e. $t_s \leq t_e$), directly follow from the conditions of the cases in Definition 5.7. More precisely, it is easy to check that for all cases except the fourth one, the conditions, together with the fact that both I_1 and I_2 are defined as non-empty intervals over $\mathbb{N}_0$, imply $0 \leq t'_s \leq t'_e$. The fourth case produces an empty sequence, so all invariants apply trivially.

For the third and fourth invariant, only the last case of Definition 5.7 is relevant since the other cases produce sequences with at most one element.

Therefore, it remains to show that $([t_{1,s}, t_{2,s}-1], [t_{2,e}+1, t_{1,e}])$ fulfills the third and fourth invariants. This again follows directly from the fact that I_2 is an admissible discrete temporal interval, which implies $t_{2,s} \leq t_{2,e}$ and hence $t_{2,s} - 1 < t_{2,e} + 1$. Since the sequence contains exactly two intervals, this satisfies both invariant 3 and 4. □

Based on the definitions above, it is now possible to specify the most important operator, namely the intersection between an interval sequence and an interval.

Definition 5.8 (Intersection of an interval sequence with an interval)**.** Let $\overleftrightarrow{T} = (I_i)_{i=0}^{N}$ be a temporal interval sequence with $(I_i)_{i=k}^{l}$ representing its subsequence from k to l. Additionally, let $I = [t_s, t_e]$ be a closed temporal interval. Then the intersection between $\overleftrightarrow{T}$ and I creates a new temporal interval sequence that is defined as follows:

$$(I_i)_{i=k}^{l} \cap I := \begin{cases} ((I_i)_{i=k+1}^{l} \cap I) \oplus (I_k \cap I) & \text{if } k < l \\ (I'_i)_{i=0}^{0} \text{ with } I'_0 = I_k \cap I & \text{otherwise} \end{cases}$$

Similar to the union operator from Definition 5.5, the intersection defined above recursively processes each interval in the sequence and adds its intersection with the query interval I to the result sequence. As in Definition 5.5, the interval addition operator $\oplus$ guarantees that the resulting interval sequence satisfies the invariants of Definition 5.2. Therefore, it is also easy to show that temporal interval sequences are closed under intersection with an interval, which is stated in the next theorem.

Theorem 5.5. *Let $\overleftrightarrow{T} = (I_i)_{i=0}^{N}$ be a temporal interval sequence and $I = [t_s, t_e]$ be a closed temporal interval. Then $\overleftrightarrow{T} \cap I$ yields a new admissible temporal interval sequence that fulfills the invariants from Definition 5.2.*

Proof. The proof works by induction over the cardinality of $\overleftrightarrow{T}$ and is structured very similar to the proof of Theorem 5.3. Since the intersection with an empty sequence is trivially empty, let the induction start with $N = 0$, which means that $\overleftrightarrow{T}$ contains exactly one interval. In this case, Definition 5.8 directly yields $\overleftrightarrow{T} \cap I = I_0 \cap$. For the induction step, consider a cardinality of $N+1$ for $\overleftrightarrow{T}$. Then, according to Definition 5.8, it holds that

$$\overleftrightarrow{T} \cap I = (I_i)_{i=0}^{N+1} \cap I = ((I_i)_{i=1}^{N+1} \cap I) \oplus (I_0 \cap I)$$

The right side of the $\oplus$ above is a single interval by definition and, given the induction hypothesis, the left side is a valid temporal interval sequence. Therefore, due to Theorem 5.2, $\overleftrightarrow{T} \cap I$ must also be an admissible temporal interval sequence. □

As in the case of two intervals, the property evaluation algorithms also need to process the remainder of intersections between an interval sequence and a query interval. Hence, the relative complement operator has to be extended similarly to the extension of the intersection.

Definition 5.9 (Relative complement of an interval in an interval sequence)**.** Let $\overbracket{T} = (I_i)_{i=0}^{N}$ be a temporal interval sequence with $(I_i)_{i=k}^{l}$ representing its subsequence from k to l. Additionally, let $I = [t_s, t_e]$ be a closed temporal interval. Then the relative complement of I in $\overbracket{T}$, written as $\overbracket{T} \backslash I$ yields a new temporal interval sequence that is defined as follows:

$$(I_i)_{i=k}^{l} \setminus I := \begin{cases} ((I_i)_{i=k+1}^{l} \setminus I) \oplus (I_k \setminus I) & \text{if } k < l \\ \\ (I'_i)_{i=0}^{0} \text{ with } I'_0 = I_k \setminus I & \text{otherwise} \end{cases}$$

For the sake of completeness, it is obviously necessary to show also in this case that the properties of Definition 5.2 are maintained.

Theorem 5.6. *Let $\overbracket{T} = (I_i)_{i=0}^{N}$ be a temporal interval sequence and $I = [t_s, t_e]$ be a closed temporal interval. Then the relative complement $\overbracket{T} \backslash I$ is an admissible temporal interval sequence according that fulfills the requirements of Definition 5.2.*

Proof. The proof works by induction over the cardinality of $\overbracket{T}$ and uses Theorem 5.4. Otherwise, it works exactly as the one in Theorem 5.5 and is therefore not repeated in detail. $\square$

Figure 5.8 visualizes the application of the intersection and relative complement operators between an interval sequence and a query interval. As mentioned before, these two operations are essential for the evaluation of temporal operators in SALMA's property evaluation algorithm and will therefore appear often during the next sections.

Finally, by a similar recursion scheme as before, it is also straightforward to define the relative complements of an interval sequence in another.

Definition 5.10 (Relative complement of an interval sequence in another interval sequence)**.** Let $\overbracket{T_1} = (I_{1,i})_{i=0}^{N}$ and $\overbracket{T_2}$ be temporal interval sequences. Then the relative complement of $\overbracket{T_1}$ in $\overbracket{T_2}$, written as $\overbracket{T_2} \backslash \overbracket{T_1}$ yields a new temporal interval sequence that is defined as follows:

$$\overbracket{T_2} \setminus (I_{1,i})_{i=k}^{N} := \begin{cases} (\overbracket{T_2} \backslash I_{1,k}) \setminus (I_{1,i})_{i=k+1}^{N} & \text{if } k < N \\ \overbracket{T_2} \backslash I_{1,k} & \text{otherwise} \end{cases}$$

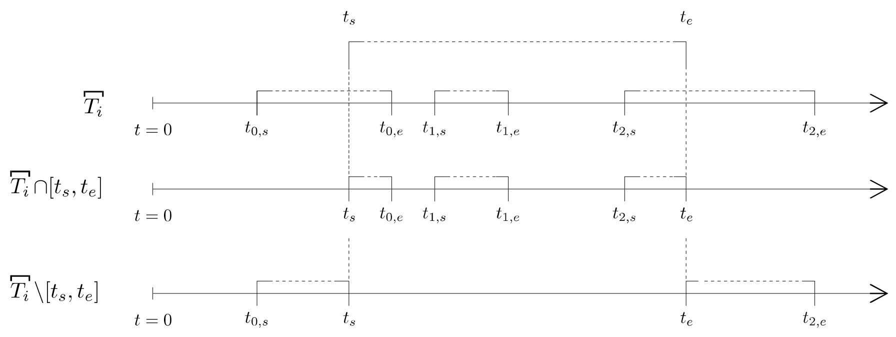


Figure 5.8: Intersection and complement of sequence and interval.

Again, the validity of the construction above is stated separately in the theorem below, which follows directly from the theorem above.

Theorem 5.7. *Let $\overline{T_1} = (I_{1,i})_{i=0}^N$ and $\overline{T_2}$ be temporal interval sequences. Then the relative complement $\overline{T_2} \setminus \overline{T_1}$ is an admissible temporal interval sequence according that fulfills the requirements of Definition 5.2.*

Proof. The proof works by induction over the cardinality of $\overline{T_1}$ and uses Theorem 5.6. Since it is structured exactly as the one in Theorem 5.5, it is not repeated in detail. □

The definitions above together form a solid basis that allow the creation, access, and manipulation of temporal interval sequences. Additionally, at several points of the algorithms, a function is needed to calculate the latest time point in an interval sequence.

Definition 5.11 (Latest time point in interval sequence)**.** Let $\overline{T} = (I_i)_{i=0}^N$ be a temporal interval sequence. Then the latest time point in $\overline{T}$, written as $max_t(\overline{T})$ is defined as follows:

$$max_t(\overline{T}) = max_t((I_i)_{i=0}^N) := \max\{t \in \mathbb{N}_0 \mid \exists i.\, t \in I_i\}$$

5.3.3 Result Mappings

Besides the plain interval sequences, suitable data structures are needed to propagate batches of evaluation *results* that are gathered when the schedule is processed. In fact, for reasons that will become clearer during the following sections, we need a way to associate individual temporal intervals with evaluation results, which is provided by the following definition.

Definition 5.12 (Temporal result mapping). A temporal result mapping $\overset{res}{\overline{R}}$ is a tuple $\langle \overline{T}, R\rangle$ that consists of a temporal interval sequence $\overline{T}$ and a function $R : \mathcal{I}_T(\overline{T}) \to \{\top, \bot, ?\}$ that labels each interval in $\overline{T}$ with a result, i.e. one of $\top$, $\bot$, or ?. Furthermore, let $\overset{res}{\overline{R}}$.intv refer to the temporal interval sequence of $\overset{res}{\overline{R}}$ and $\overset{res}{\overline{R}}$.rmap to its labeling function.

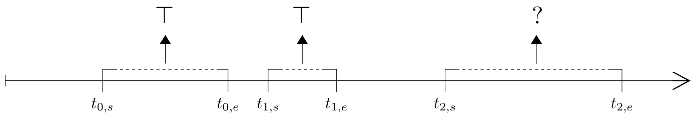


Figure 5.9: Example for a result mapping.

Figure 5.9 shows a possible mapping of the intervals in Figure 5.4 to different results. This particular example could describe a typical outcome of the evaluation of a property at the end of a simulation step. Here, the algorithm was able to decide that the scheduled evaluation goal instances that start within the first two intervals are successful, while no conclusive result has been found yet for the instances that start within the third interval.

Since result mappings are actually merely labeled temporal interval sequences, it is possible to reuse the operations that were defined in the last section and extend them in a very straightforward manner to incorporate the result labels. The first step is a simple constructor that assigns the same result label to all intervals.

Definition 5.13 (Unique result assignment). Let $\overline{T} = (I_i)_{i=0}^{N}$ be a temporal interval sequence and $r \in \{\top, \bot, ?\}$ a result label. Then the *unique result assignment* of r to the intervals in $\overline{T}$, written as $\overline{T}|_r$ yields a result mapping $\overset{res}{\overline{R}} = \langle \overline{T}, R\rangle$ with the mapping function $R : \mathcal{I}_T(\overline{T}) \to \{\top, \bot, ?\}$ defined as follows:

$$R(I) = r \qquad \text{for all } I \in \mathcal{I}_T(\overline{T})$$

The only other constructor needed for result mappings is the union operator. Since the evaluation algorithms allow a restriction to result mappings with non-overlapping intervals, the union an easily be defined by leveraging the union operator for interval sequences.

Definition 5.14 (Union of temporal result mappings). Let $\overset{res}{\overline{R}} = \langle \overline{T}, R\rangle$ and $\overset{res}{\overline{R'}} = \langle \overline{T'}, R'\rangle$ be two temporal result mappings with disjunct temporal interval sequences, i.e. it holds that

$$\forall I \in \mathcal{I}_T(\overline{T}).\, \forall I' \in \mathcal{I}_T(\overline{T'}).\, I \cap I' = \emptyset$$

. Then the union of $\overset{res}{\overbracket{R}}$ and $\overset{res}{\overbracket{R'}}$ is defined as

$$\overset{res}{\overbracket{R}} \cup \overset{res}{\overbracket{R'}} = \langle \overbracket{T} \cup \overbracket{T'}, R^{\cup} \rangle$$

where

$$R^{\cup} : \mathcal{I}_T(\overbracket{T}) \cup \mathcal{I}_T(\overbracket{T'}) \rightarrow \{\top, \bot, ?\} = \begin{cases} R(I) & \text{if } I \in \mathcal{I}_T(\overbracket{T}) \\ R'(I) & \text{if } I \in \mathcal{I}_T(\overbracket{T'}) \end{cases}$$

As expected, the union of result mappings basically amounts to creating a union of the interval sequences and combining the mapping functions by case distinction. However, a restriction to disjunct temporal interval sequences is necessary since overlapping intervals with different result labels would lead to ambiguity. Therefore, it will later be necessary to show that the participating result mappings in fact have disjunctive interval sequences whenever the union operator is used.

With the operations defined so far, it is possible to construct any result mapping simply by adding together mapping segments with unique labels that were constructed with the $|_r$ operator. Additionally, the evaluation algorithms in Section 5.6 need to be able to use result mappings within logical connectives, i.e. negation, conjunction, and disjunction. Naturally, the most basic one is the negation operator that simply flips all definite labels.

Definition 5.15 (Negation of a result mapping)**.** Let $\overset{res}{\overbracket{R}} = \langle \overbracket{T}, R \rangle$ be a result mapping. Then, the negation of $\overset{res}{\overbracket{R}}$, written as $\neg \overset{res}{\overbracket{R}}$ is defined as follows:

$$\neg \overset{res}{\overbracket{R}} = \langle \overbracket{T}, R^{\neg} \rangle$$

where $R^{\neg} : \mathcal{I}_T(\overbracket{T}) \rightarrow \{\top, \bot, ?\}$ with

$$R^{\neg}(I) = \begin{cases} \top & \text{if } R(I) = \bot \\ \bot & \text{if } R(I) = \top \\ ? & \text{otherwise} \end{cases}$$

The other two required connectives, $\wedge$ and $\vee$, are a bit more complex. The problem is that the interval sequences of the two participating result mappings are not necessarily equal. This means that intervals possibly have to be cut and recomposed according to the application of the logical operators to interval overlaps.

Definition 5.16 (Logical connectives for result mappings)**.**
Let $\overset{res}{\overline{R_1}} = \langle \overline{T_1}, R_1 \rangle$ and $\overset{res}{\overline{R_2}} = \langle \overline{T_2}, R_2 \rangle$ be result mappings. Then the conjunction and disjunction of the two can be defined as follows:

$$\overset{res}{\overline{R_1}} \wedge \overset{res}{\overline{R_2}} = \langle \overline{T^\wedge}, R^\wedge \rangle = \overline{T_{\wedge,?}}|_? \cup \overline{T_{\wedge,\top}}|_\top \cup \overline{T_{\wedge,\bot}}|_\bot$$

and

$$\overset{res}{\overline{R_1}} \vee \overset{res}{\overline{R_2}} = \langle \overline{T^\vee}, R^\vee \rangle = \overline{T_{\vee,?}}|_? \cup \overline{T_{\vee,\top}}|_\top \cup \overline{T_{\vee,\bot}}|_\bot$$

.

To construct the interval sequences in the definition above, first all intervals from the source sequences are sorted by their label in the result mappings and combined into three new sequences $\overline{T_\top}$, $\overline{T_\bot}$, and $\overline{T_?}$:

$$\overline{T_\top} = \overline{T_{1,\top}} \cup \overline{T_{2,\top}}, \quad \overline{T_\bot} = \overline{T_{1,\bot}} \cup \overline{T_{2,\bot}}, \quad \overline{T_?} = \overline{T_{1,?}} \cup \overline{T_{2,?}}$$

with

$$\overline{T_{j,\top}} = \bigoplus_{I \in \mathcal{I}_{1,\top}} I \quad \text{where } \mathcal{I}_{j,\top} = \{I \in \overline{T_j} \mid R_j(I) = \top\} \text{ for } j \in \{1,2\}$$

and analogically for $\overline{T_{j,\bot}}$ and $\overline{T_{j,?}}$.

With these combined intervals and the relative complement of one interval sequence in another from Definition 5.10, it is possible to define the sequences from which the final result mappings are constructed:

$$\begin{aligned}
\overline{T_{\wedge,?}} &= \overline{T_?} \setminus \overline{T_\bot} & \overline{T_{\vee,?}} &= \overline{T_?} \setminus \overline{T_\top} \\
\overline{T_{\wedge,\top}} &= (\overline{T_\top} \setminus \overline{T_\bot}) \setminus \overline{T_{\wedge,?}} & \overline{T_{\vee,\top}} &= \overline{T_\top} \\
\overline{T_{\wedge,\bot}} &= \overline{T_\bot} & \overline{T_{\vee,\bot}} &= (\overline{T_\bot} \setminus \overline{T_\top}) \setminus \overline{T_{\vee,?}}
\end{aligned}$$

Finally, a function is needed that determines a summary of a result mapping, which could be unanimous if all intervals have the same label, ambiguous if the mapping contains both *top* and $\bot$ intervals, and undetermined (?) if at least one interval is undetermined.

Definition 5.17 (Summary of a result mapping)**.** Let $\overset{res}{\overline{R}} = \langle \overline{T}, R \rangle$ be a result mapping. The summary of $\overset{res}{\overline{R}}$, written as $\mathsf{summary}(\overset{res}{\overline{R}})$, is defined as follows:

$$\mathsf{summary}(\overset{res}{\overbracket{R}}) = \begin{cases} \top & \text{if } \forall I \in \mathcal{I}_T(\overbracket{T}).R(I) = \top \\ \bot & \text{if } \forall I \in \mathcal{I}_T(\overbracket{T}).R(I) = \bot \\ \mathsf{ambiguous} & \text{if } (\forall I \in \mathcal{I}_T(\overbracket{T}).R(I) \neq ?) \\ & \quad \wedge (\exists I' \in \mathcal{I}_T(\overbracket{T}).R(I') = \top) \\ & \quad \wedge (\exists I'' \in \mathcal{I}_T(\overbracket{T}).R(I'') = \bot) \\ ? & \text{if } \exists I \in \mathcal{I}_T(\overbracket{T}).R(I) = ? \end{cases}$$

In the definition above, ? has precedence over ambiguous in the sense that even if the result mapping contains both positive and negative definite intervals, it is still seen as undetermined if at least one of its intervals is. This is important for the evaluation algorithm since it indicates that at least for some of the intervals, entries for the evaluation goal schedule have to be created.

Other mechanisms to manipulate result mappings are actually not necessary, since result mappings are only used as means for data collection throughout recursive function calls during property evaluation. The actual evaluation state information in the evaluation goal schedule is stored differently. This will be discussed in Section 5.5. However, before the data structures involved in the evaluation goal schedule can be examined properly, it is necessary to look at the way in which formulas are translated into the internal representation that is actually used by the Prolog-based property interpreter. This is the topic of the next section.

5.4 The Property Compiler

As mentioned in the beginning of this chapter, the SALMA-PSL formulas of the invariants and goals that are registered for evaluation are first sent to the *property compiler*, which transforms them to an internal representation that is interpreted by the evaluation algorithm. In fact, there are three main aspects that are addressed by this translation:

1. The quantifiers `forall` and `exists` have to be unfolded into conjunctions and disjunctions, respectively, over the set of entities that exist for the selected sort.

2. The functional-styled formulas have to be transformed into a representation that conforms to the evaluation and variable binding schema of logic programming.

3. The situation arguments at the last position of fluent usages have to be restored.

Due to its rather technical nature, it is not practicable to describe the whole compilation process in detail. However, this section will describe the most important transformation rules since they offer valuable insight in the expected structure of the internal representation. This in turn will later in Section 5.6 be very helpful for understanding some of the most essential design aspects of the evaluation algorithm.

First, however, it is necessary to introduce some essential concepts, namely variable substitution, term unification, and subterm substitution. First, variable substitution and unification are defined in the usual way in which it can be found throughout the literature, for instance in [BA12, CHAP. 10].

Definition 5.18 (Variable substitution)**.** A variable substitution $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is a set of mappings from variables to terms. The application of a substitution θ to a term ϕ, is written as $\theta\phi$. The result of $\theta\phi$ is defined as the term that results from replacing any occurrence of a variable x_1 to x_n in ϕ with the corresponding replacement term from the substitution θ.

Using variable substitution *unification* of two terms can be defined.

Definition 5.19 (Unification)**.** A substitution θ is a *unifier* for two terms ϕ and ψ if $\theta\phi = \theta\psi$. Furthermore, Two terms ϕ and ψ are *unifiable*, written $\phi \simeq \theta$ if there exists a unifier for them.

Besides being one of the most basic mechanism in logic programming and therefore in the evaluation of SALMA-PSL formulas, the unification concept is used here to define the *subterm substitution* operation, which is used in various places both in the property compiler and the formula evaluation algorithms.

Definition 5.20 (Subterm substitution)**.** Let Θ be a term and let $\Psi = \{\theta_1 \mapsto \theta'_1, \ldots, \theta_m \mapsto \theta'_m\}$ be a set of *mappings* that relate arbitrary ground or non-ground terms θ_i to ground replacement terms θ'_i. Then $subst_{term}(\Psi, \Theta)$ denotes the *subterm substitution* that results from rewriting Θ by replacing each subterm of Θ that unifies with one of $\theta_1, \ldots, \theta_m$ with the respective substitution term. Formally, let c be a symbol with arity 0, ϕ a symbol with arity n. Then $subst_{term}(\Psi, \Theta)$ is recursively defined as follows:

1. $subst_{term}(\Psi, c) = \begin{cases} \theta' & \text{if } \exists\theta.\theta \simeq c \land \theta \mapsto \theta' \in \Psi \\ c & \text{otherwise} \end{cases}$

2. $subst_{term}(\Psi, \phi(t_1, \ldots, t_n)) =$
$$\begin{cases} \theta' & \text{if } \exists\theta.\theta \simeq \phi(t_1, \ldots, t_n) \land \theta \mapsto \theta' \in \Psi \\ \phi(subst_{term}(\Psi, t_1), \ldots, & \\ \qquad subst_{term}(\Psi, t_n)) & \text{otherwise} \end{cases}$$

3. $subst_{term}(\{\theta \mapsto \theta'\} \cup \Psi, \Phi) = subst_{term}(\{\theta \mapsto \theta'\}, subst_{term}(\Psi, \Phi))$

The ability to replace arbitrary subterms allows in particular to substitute variables that are bound by quantifiers. In fact, since all sort domains within a SALMA model are constrained to be finite sets, the property compiler can eliminate any quantifier in the following way:

1. Iterate over the entities in the sort domain of the variable that is bound by the quantifier.
2. For each of these entities, create an instance of the original subformula inside the quantifier where this entity is substituted for the bound variable.
3. In case of universal quantification (`forall`), replace the quantifier block with a conjunction over the generated subformula instances. For existential quantification (`exists`), use a disjunction instead.

Formally, when $[\![\Phi]\!]_{comp}$ denotes the result of the compilation procedure for the SALMA-PSL formula Φ, then the quantifier elimination step can be expressed as follows:

Definition 5.21 (Quantifier elimination in the property compiler)**.** Let Φ be a SALMA-PSL formula and T be a finite sort with $domain(T) = \{e_1, \ldots, e_n\}$. Then the elimination of quantifiers in Φ, written as $elimQ(\Phi)$ can be defined recursively as follows:

$$\begin{aligned}
elimQ(forall(x:T,\Phi)) &= and(elimQ(subst_{term}(\{x \mapsto e_1\}, \Phi)), \ldots,\\
&\qquad elimQ(subst_{term}(\{x \mapsto e_n\}, \Phi)))\\
elimQ(exists(x:T,\Phi)) &= or(elimQ(subst_{term}(\{x \mapsto e_1\}, \Phi)), \ldots,\\
&\qquad elimQ(subst_{term}(\{x \mapsto e_n\}, \Phi)))
\end{aligned}$$

One direct result of this way of representing quantification is that formulas have to be re-compiled when the domain of a sort changes during simulation, i.e. when entities are created or destroyed during simulation. Although the use of dynamic sorts complicates reasoning about the system model and is therefore not generally recommended, it is used within SALMA's extension for modeling information transfer that is described in Chapter 6 (see Section 6.2.5). Having to re-compile formulas during simulations might seem like an unnecessary overhead compared to possible alternative solutions in which sort domains are cleanly separated from the formulas and accessed dynamically during evaluation. However, since formulas in the evaluation goal schedule are not re-compiled, the domain at the time a formula is added to the schedule is effectively fixated at that moment. A look back to Definition 4.10 reveals that this is exactly what is required by the intended semantics.

After all quantifiers have been eliminated, the property compiler has to translate SALMA-PSL formulas to a representation that can be evaluated by the Prolog interpreter. The main problem is that the evaluation scheme of Prolog does not really support the functional style used in SALMA-PSL formulas. Instead, each immediate result within an evaluation has to be bound to a Prolog variable [2] and propagated to the next steps. For instance, consider a formula that tests whether the combined weight of two items is less than 100. When it is assumed that `weight` is a constant defined to take one argument of type `item`, then this could be expressed with the following SALMA-PSL expression:

```
weight(item1) + weight(item2) < 100
```

However, the Prolog representation of the constant `weight` and the built-in function `+` are actually defined as predicates that bind their result to a variable that is passed as the last position. This means that the results of all steps of the calculation have to be collected in fresh Prolog variables and then combined in the comparison. In this case, the expression from above will be translated to

```
all([weight(item1, _530), weight(item2, _567), +(_530, _567, _515), _515 < 100])
```

Here, `_530`, `_567`, and `_515` are unnamed (anonymous) Prolog variables that are created programmatically during the compilation process. It can be seen that the first two of them, which store the results retrieved from the `weight` predicate, are actually substituted for the corresponding `weight()` subterms within the sum. The same happens again with `_515` which eventually passes the sum to the comparison with 100. All rewritten parts are finally combined in a conjunction, which is represented by `all` in the compilation result. The compiler assures that the binding of each variable appears before its use in the conjunction. Knowing that the evaluation algorithm processes the elements of the conjunction in order, this realizes the intended semantics.

The treatment of functional fluents is very similar to that of regular functions, except that the initial situation term (`s0`) has to be added as a last argument. With this general pattern in mind, the semantics of the translation procedure can be summarized.

[2] The name Prolog variable is used here and in the following to distinguish them from variables that appear in untranslated SALMA-PSL expressions.

Definition 5.22 (Compilation of SALMA-PSL formulas)**.** Let $R(\theta_1, \ldots, \theta_n)$ denote any *relation*, including relational fluents, situation-independent predicates, and comparisons like $<$, $\leq$, $=$, $\geq$, $>$, or $\neq$. On the other hand, let f represent a situation-independent function, a functional fluent, or an arithmetic operation. For brevity's sake, the operators for comparisons and arithmetics are thought to appear in prefix form (e.g. $+(weight(rob1), weight(rob2))$ instead of $weight(rob1) + weight(rob2))$ and thus can be treated exactly like situation-independent predicates or functions. For the subterms $\theta_1, \ldots, \theta_n$ and $\Theta_1, \ldots, \Theta_n$, it is assumed that they do not contain any quantifiers, i.e. there either were none in the original SALMA-PSL formula or they were eliminated by the procedure described above. Furthermore, let $[\Theta_1, \ldots, \Theta_n]$ represent a list of terms, and $A \circ B$ the concatenation of list A and list B. Additionally, $\langle X_1, \ldots, X_n \rangle$ is used as a notation to describe either a tuple of values that are returned by a function, or a list of variables to which the elements of a tuple are assigned. Besides these variables that are used in the rules below, it is necessary to distinguish between *symbolic variables* that appear in the SALMA-PSL expressions as regular subterms, and *logical variables* that are instantiated directly by the Prolog interpreter. Therefore, logical variables are marked with a hat, e.g. $\hat{x}$, while unmarked names are used for symbolic variables.

With these notational conventions, the compilation procedure can be described as follows:

1. $processEvalTerms(Bindings, [\Theta_1, \ldots, \Theta_n]) = \langle Bindings', [\Theta_1'] \circ Params' \rangle$

   where

   $\langle Bindings', Params' \rangle = processEvalTerms(Bindings'', [\Theta_2, \ldots, \Theta_n])$

   where $\langle Bindings'', \Theta' \rangle = processEvalTerm(Bindings, \Theta_1)$

2. $processEvalTerms(Bindings, [\Theta]) = processEvalTerm(Bindings, \Theta)$

3. $processEvalTerm(Bindings, v) = \langle Bindings, v \rangle$

   where v is a number or an entity literal

4. $processEvalTerm(Bindings, f(\theta_1, \ldots, \theta_n)) = \langle Bindings' \circ [\Theta'], \hat{x} \rangle$

   where

   $\langle Bindings', [\theta_1', \ldots, \theta_n'] \rangle = processEvalTerms(Bindings, [\theta_1, \ldots, \theta_n])$

   and

   $$\Theta' = \begin{cases} f(\theta_1', \ldots, \theta_n', \hat{x}, S_0) & \text{if } f \text{ is a fluent} \\ f(\theta_1', \ldots, \theta_n', \hat{x}) & \text{otherwise} \end{cases}$$

5. $\mathit{compile}(R(\theta_1, \ldots, \theta_n)) = c\_([\beta_1, \ldots, \beta_m, \Theta'])$

   where

   $$\Theta' = \begin{cases} R(\theta'_1, \ldots, \theta'_n, S_0) & \text{if } R \text{ is a relational fluent} \\ R(\theta'_1, \ldots, \theta'_n) & \text{otherwise} \end{cases}$$

   with

   $\langle [\beta_1, \ldots, \beta_m], [\theta'_1, \ldots, \theta'_n] \rangle = \mathit{processEvalTerms}([\,], [\theta_1, \ldots, \theta_n])$

6. $\mathit{compile}(\mathit{and}(\Theta_1, \ldots, \Theta_n)) = \mathit{all}([\mathit{compile}(\Theta_1), \ldots, \mathit{compile}(\Theta_n)])$

7. $\mathit{compile}(\mathit{or}(\Theta_1, \ldots, \Theta_n)) = \mathit{one}([\mathit{compile}(\Theta_1), \ldots, \mathit{compile}(\Theta_n)])$

8. $\mathit{compile}(\mathit{not}(\Theta)) = \mathit{not}(\mathit{compile}(\Theta))$

9. $\mathit{compile}(\mathit{implies}(\Phi, \Psi)) = \mathit{one}([\mathit{not}(\mathit{compile}(\Phi)), \mathit{compile}(\Psi)])$

10. $\mathit{compile}(\mathit{eventually}(T, \Phi)) = \mathit{eventually}(T, \mathit{compile}(\Phi))$

11. $\mathit{compile}(\mathit{always}(T, \Phi)) = \mathit{always}(T, \mathit{compile}(\Phi))$

12. $\mathit{compile}(\mathit{until}(T, \Phi, \Psi)) = \mathit{until}(T, \mathit{compile}(\Phi), \mathit{compile}(\Psi))$

13. $\mathit{compile}(\mathit{occur}(\Theta)) = \mathit{occur}(\mathit{compile}(\Theta))$

14. $\mathit{compile}(\mathit{let}(x : \Theta, \Phi)) = \mathit{let}(x : \hat{x}, \Theta', \Phi')$

    where $\Theta' = \mathit{compile}(= (\hat{x}, \Theta))$ and $\Phi' = \mathit{compile}(\Phi)$

Most of the rules in the definition above are straightforward. The most interesting part is certainly the compilation of predicates and relational fluents in rule 5, which also includes comparisons. In this rule, the compiler uses the recursive functions *processEvalTerm* and *processEvalTerms* to gather terms that instantiate freshly created variables, which in turn are substituted for the parameters of compiled relation R. The recursion works on a tuple that consists of two lists that are labeled *Bindings* and *Params*, respectively. The second list contains the parameters that are eventually used in the compilation result, i.e. either constants like numbers or entity symbols, or freshly created logical variables (see above). To this end, the first list, *Bindings*, contains the terms that, when evaluated, bind these variables to the values of the original subterms they represent. For this to work, the compiler has to assure that, for each variable, its binding term precedes all usages. Indeed, this can be verified by inspection of rule 4: since the binding terms gathered in the recursion step ($\mathit{Bindings}'$) are used as the left side of the concatenation with the new binding term Θ', the binding terms are effectively ordered from inside to outside. With the generated order, all terms are combined with the special $c\_(\ldots)$ operator,

$$\begin{aligned}
\mathit{comp}(p(f(g(h(t))))) &= c\_([\beta_1, \ldots, \beta_m, \Theta]) && (1)\\
\Theta &= p(\theta) && (2)\\
\langle[\beta_1, \ldots, \beta_m], [\theta]\rangle &= \mathit{pets}([f(g(h(t)))]) && (3)\\
&= \mathit{pet}(f(g(h(t)))) = \langle B' \circ [\Theta'], \hat{x}\rangle &&\\
\Theta' &= f(\theta', \hat{x}) && (4)\\
\langle B', [\theta']\rangle &= \mathit{pet}(g(h(t))) = \langle B'' \circ [\Theta''], \hat{y}\rangle && (5)\\
\Theta'' &= g(\theta'', \hat{y}) && (6)\\
\langle B'', [\theta'']\rangle &= \mathit{pet}(h(t)) = \langle B''' \circ [\Theta'''], \hat{z}\rangle && (7)\\
\Theta''' &= h(\theta''', \hat{z}) && (8)\\
\langle B''', [\theta''']\rangle &= \mathit{pet}(t) = \langle[], [t]\rangle && (9)
\end{aligned}$$

$$\begin{aligned}
9 \text{ in } 8: \quad \Theta''' &= h(t, \hat{z}) && (10)\\
9, 10 \text{ in } 7: \quad \langle B'', [\theta'']\rangle &= \langle[] \circ [h(t, \hat{z})], \hat{z}\rangle && (11)\\
11 \text{ in } 6: \quad \Theta'' &= g(\hat{z}, \hat{y}) && (12)\\
11, 12 \text{ in } 5: \langle B', [\theta']\rangle &= \langle[] \circ [h(t, \hat{z})] \circ [g(\hat{z}, \hat{y})], \hat{y}\rangle && (13)\\
13 \text{ in } 4: \Theta' &= f(\hat{y}, \hat{x}) && (14)\\
14, 13 \text{ in } 3: \langle[\beta_1, \ldots, \beta_m], [\theta]\rangle &= \langle[h(t, \hat{z}), g(\hat{z}, \hat{y}), f(\hat{y}, \hat{x})], [\hat{x}]\rangle && (15)\\
15 \text{ in } 2: \quad \Theta &= p(\hat{x}) && (16)\\
15, 16 \text{ in } 1: \mathit{comp}(p(f(g(h(t))))) &= c\_([h(t, \hat{z}), g(\hat{z}, \hat{y}), && (17)\\
&\qquad f(\hat{y}, \hat{x}), p(\hat{x})]) &&
\end{aligned}$$

Figure 5.10: Transformation of a predicate in the SALMA-PSL compiler.

which is a synonym for $\mathit{all}(\ldots)$ but marks the contained term sequence as a coherent evaluation unit that may not be separated by term transformations.

The example in Figure 5.10 illustrates the ideas sketched above. To make it more readable, *pet* and *pets* were used as abbreviations for the functions *processEvalTerm* and *processEvalTerms* from above. Otherwise, the example adheres strictly to the rules from Definition 5.22. In this case, p is supposed to be a situation-independent predicate, f, g, and h are unary situation-independent functions, and t is a literal, e.g. an entity name.

5.5 The Evaluation Goal Schedule

With the concepts introduced in the last sections, it is now possible to describe the evaluation goal schedule as one of the most important elements of the evaluation mechanism. Before a concise summary of the evaluation goal schedule is provided in Section 5.5.4, the involved data structures and mechanisms are introduced and explained by means of several detailed examples which gradually add more complexity. The intention is to present the rationale behind each design choice from the beginning and at the same time demonstrate their consequences.

5.5.1 Basic Structure

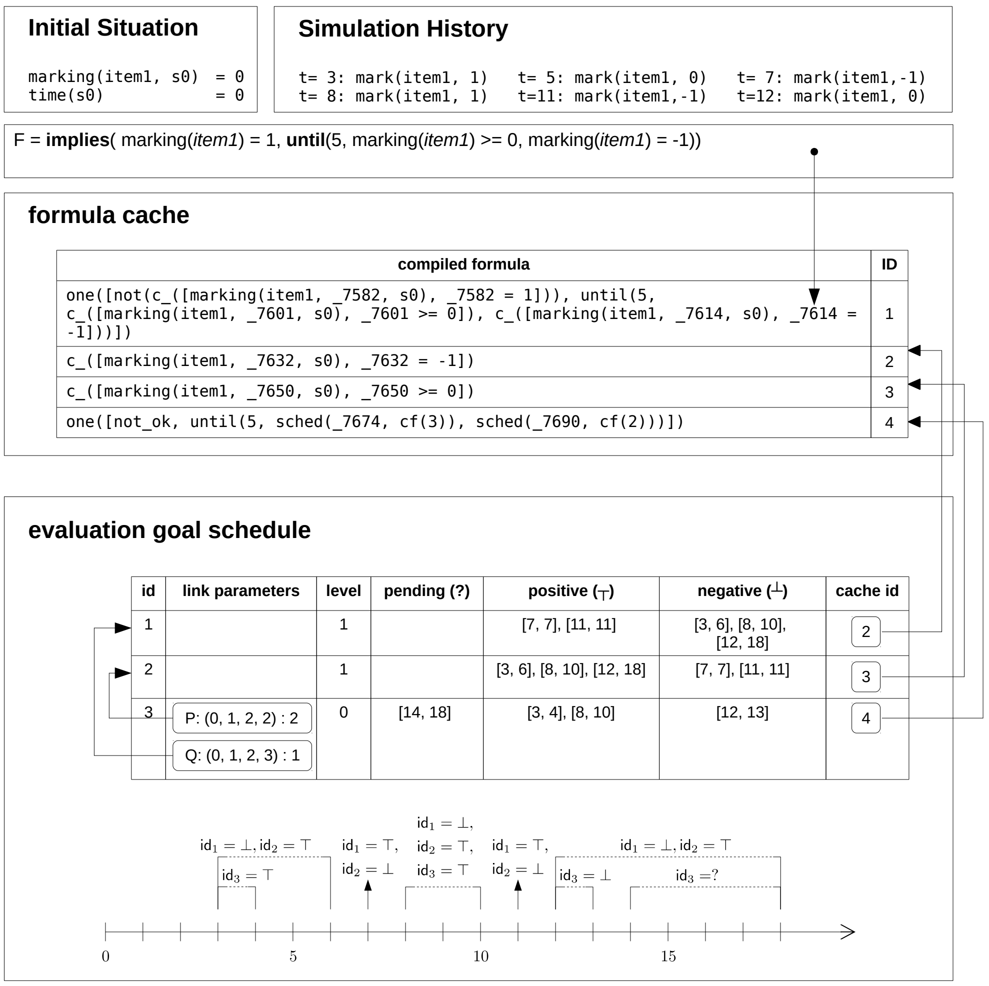
As expected, temporal interval sequences act as the basic structure for keeping track of the current state of the evaluation. The general idea is that the state of a (partial) formula can be represented by three temporal interval sequences - one that memorizes time points where the property was true, one for instances where it was false, and one that marks time points where the evaluation was not conclusive due to the involvement of temporal operators. The inconclusive case means in fact that a new obligation is added to re-evaluate the formula with the memorized start time as context in each future step until a conclusive result can be found.

Figure 5.11 shows an exemplary possible concrete evaluation state of a property. Here, a simulation was run for 18 time steps so far, and the following property F was evaluated in each step:

```
F = implies(marking(item1) = 1,
            until(5, marking(item1) >= 0, marking(item1) = -1))
```

The formula refers to a simple model that was created for testing different formula structures. The fluent marking (which can be manipulated with the corresponding action mark) associates an arbitrary term with an entity (here `item1`). Property F requires that at any time point where marking(*item1*) $= 1$, the same item will be marked with -1 within 5 time units, and until that, the marking will not go below 0. This rather artificial example allows the free construction of any desired scenario without semantic constraints and dependencies that would normally be imposed on a more realistic model.

The top of Figure 5.11 shows the initial situation, i.e. the valuation of the fluents for time step 0, and the simulation history by means of the sequence of actions that occurred during the simulation so far. From the structure of the formula, it is clear that each time where the marking of *item1* is 1, the evaluation result is open and hence the time point has to be marked and re-visited in all following steps. In Figure 5.11, it can be seen that the evaluation of the property actually resulted in three entries to the evaluation goal schedule - each represented by a line in the schedule table. More precisely, the entry with id 3 represents the whole formula F, while entries 1 holds the goal part of the until-operator in F, and 2 holds the invariant. Each schedule entry first of all contains the mentioned three temporal interval sequences to store positive, negative, and inconclusive (pending) outcomes. In Figure 5.11, the same



Initial Situation

```
marking(item1, s0)  = 0
time(s0)            = 0
```

Simulation History

```
t= 3: mark(item1, 1)   t= 5: mark(item1, 0)   t= 7: mark(item1,-1)
t= 8: mark(item1, 1)   t=11: mark(item1,-1)   t=12: mark(item1, 0)
```

F = **implies**(marking(*item1*) = 1, **until**(5, marking(*item1*) >= 0, marking(*item1*) = -1))

formula cache

compiled formula	ID
`one([not(c_([marking(item1, _7582, s0), _7582 = 1])), until(5, c_([marking(item1, _7601, s0), _7601 >= 0]), c_([marking(item1, _7614, s0), _7614 = -1]))])`	1
`c_([marking(item1, _7632, s0), _7632 = -1])`	2
`c_([marking(item1, _7650, s0), _7650 >= 0])`	3
`one([not_ok, until(5, sched(_7674, cf(3)), sched(_7690, cf(2)))])`	4

evaluation goal schedule

id	link parameters	level	pending (?)	positive (⊤)	negative (⊥)	cache id
1		1		[7, 7], [11, 11]	[3, 6], [8, 10], [12, 18]	2
2		1		[3, 6], [8, 10], [12, 18]	[7, 7], [11, 11]	3
3	P: (0, 1, 2, 2) : 2 Q: (0, 1, 2, 3) : 1	0	[14, 18]	[3, 4], [8, 10]	[12, 13]	4

Figure 5.11: Example for a state snapshot of the evaluation goal schedule.

information is additionally summarized graphically in the interval diagram in the bottom.

Besides the interval sequences, each scheduled goal contains a link to an entry in the *formula cache* that keeps the corresponding part of the compiled formula that will be processed by the evaluation algorithm. As implied by the arrows in the figure, the first cache entry that holds the whole formula is added during the initial compilation phase. The other entries, however, are added on the fly during evaluation to store parts of the formula that might be rewritten according to the current state of the situation when a new (sub-)goal is scheduled is created. In fact, the third schedule entry is not linked to the original compiled formula but to a rewritten version in the fourth row of the cache. There, the negated constraint `not(c_([marking(item1, _7582, s0), _7582 = 1]))` has been replaced by `not_ok`, the internal representation of a negative result ($\perp$). This can easily be understood when the fact is considered that the mentioned part actually corresponds to the premise of the implication in the original formula, which was translated using the equivalence $A \implies B \equiv \neg A \vee B$. Since a schedule entry is only created when the premise ($\mathsf{marking}(\mathit{item1}) = 1$) is true, the corresponding subformula can be replaced by a constant to avoid unnecessary evaluation steps.

Other than this obvious optimization measure, the formula was further rewritten by replacing the subformulas of the until operator by terms marked with the special pseudo function `sched`. These expressions contain a reference to the cache entry that holds the original subformula as well as a logical variable (denoted by a number with leading underscore). Each variable is instantiated during property evaluation with the id of an entry in the evaluation goal schedule, which is used to retrieve the relevant evaluation history of the property subformula. These ids are on their part stored as *link parameters* in the schedule entry (id 3) that is associated with the rewritten cache entry. In this case, the first part of the until operator (P) is associated with the second schedule entry and the second part (Q) with the third. The link parameters of the schedule entry each contain also a subterm position path that is used to locate the `sched(...)` expression in the cached formula and thereby select the variable for instantiation. This indirection allows the evaluation algorithm to re-use the same cache entry for subformulas (e.g. constraints) that appear multiple times at different nesting levels in a formula.

With the information given above, it is now possible to interpret the situation visualized in Figure 5.11. At first, it can be noticed that the information for the entry of the whole formula (id 3) starts not before 3 and contains several gaps between 5 and 7, and at 11. This is due to the fact that the whole formula is at the top level governed by the implication with the premise $\mathsf{marking}(\mathit{item1}) = 1$. When this premise is not true, then the formula evaluates to true at once and there is no need to memorize the starting point. The intervals $[3, 4]$ and $[8, 10]$ are labeled with $\top$ for entry id_3. This corresponds to the fact that entry id_1, which refers to $\mathsf{marking}(\mathit{item1}) = -1$, is true at the points 7 and 11, which confirms all instances of the top level formula that were scheduled within 5 time units before. Of course, this also requires that the first part of the until operator was maintained between the start and the confirmation, which is easily verified by a look at the diagram. In contrast, the property instances scheduled at times 12 and 13 were already declared as failed since the required goal of the **until** expression was not achieved within the time limit of 5 steps. Finally, the last segment $[14, 18]$ labeled as ? for id_3. This means that for each scheduled instance during this period both a positive and a negative result is still possible.

5.5.2 Property Context and Variable Binding

The term rewriting mechanism mentioned before actually serves yet another purpose that is worth being examined a bit further. When the premise of the implication (marking(*item1*) = 1) was rewritten to `not_ok` at the creation of the schedule entry, this actually created a context of the property at the given time point, namely the fact that at that time, the premise was true. However, the schedule context of a property can also contain more aspects. For instance, consider the following formula:

```
F2 = implies(marking(item1) > 0,
      let(x : marking(item1),
        until(5, marking(item1) < 2 * x, marking(item1) =:= -1 * x)))
```

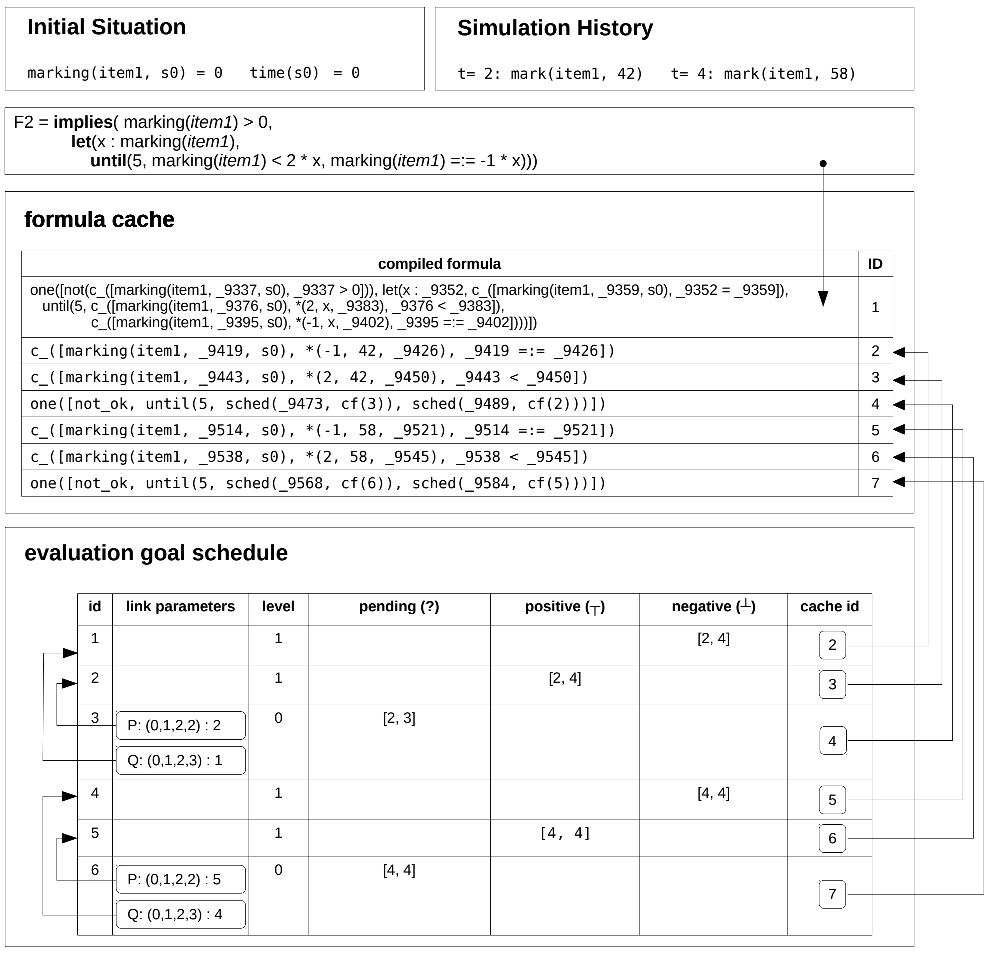
Here, the variable `x` is bound in the `let`-expression to remember the value of the item's marking at the time it was set to a value higher than 0. This value is used in the until-expression to check that, within 5 time units, the marking never exceeds twice its initial value, before it eventually is set to the negative initial amount. This means that the value with which the current marking is compared depends on the state (situation) in which the schedule entry is created, i.e. it becomes a part of the context of the schedule entry. In SALMA, this memorization is realized by rewriting the occurrences of the bound variables. This can be seen in Figure 5.12. In this example, the marking was first set to 42 at time 2 and then to 58 at time 4. This resulted in the creation of two congruent but distinct sets of event schedule entries, $1-3$ and $4-5$. By following the links between schedule entries and from schedule entries to cache entries, one can see that the `let` expression has vanished and the variable x in the products has been replaced by the value of the marking at the scheduled time.

5.5.3 Nested Temporal Operators

Both examples that were presented so far demonstrate the flexibility of SALMA's evaluation goal schedule. However, the main reason for its relatively complex reference-based design is the necessity to handle formulas with *nested* temporal operators. For example, a different property from the same domain as before might be the following:

```
G = implies(marking(item1) = 1,
      until(20,
        implies(
          occur(grab(rob1, item1)),
          until(10, carrying(rob1, item1), not(carrying(rob1,item1)))),
        marking(item1) = 0))
```

Here, G requires that when an item's marking is 1, then it holds for (at least) 20 time units that whenever robot *rob*1 picks up item *item*1, then keeps



Initial Situation

`marking(item1, s0) = 0   time(s0) = 0`

Simulation History

`t= 2: mark(item1, 42)   t= 4: mark(item1, 58)`

F2 = **implies**(marking(*item1*) > 0,
let(x : marking(*item1*),
until(5, marking(*item1*) < 2 * x, marking(*item1*) =:= -1 * x)))

formula cache

compiled formula	ID
one([not(c_([marking(item1, _9337, s0), _9337 > 0])), let(x : _9352, c_([marking(item1, _9359, s0), _9352 = _9359]), until(5, c_([marking(item1, _9376, s0), *(2, x, _9383), _9376 < _9383]), c_([marking(item1, _9395, s0), *(-1, x, _9402), _9395 =:= _9402]))))])	1
c_([marking(item1, _9419, s0), *(-1, 42, _9426), _9419 =:= _9426])	2
c_([marking(item1, _9443, s0), *(2, 42, _9450), _9443 < _9450])	3
one([not_ok, until(5, sched(_9473, cf(3)), sched(_9489, cf(2)))])	4
c_([marking(item1, _9514, s0), *(-1, 58, _9521), _9514 =:= _9521])	5
c_([marking(item1, _9538, s0), *(2, 58, _9545), _9538 < _9545])	6
one([not_ok, until(5, sched(_9568, cf(6)), sched(_9584, cf(5)))])	7

evaluation goal schedule

id	link parameters	level	pending (?)	positive ($\top$)	negative ($\perp$)	cache id
1		1			[2, 4]	2
2		1		[2, 4]		3
3	P: (0,1,2,2) : 2 Q: (0,1,2,3) : 1	0	[2, 3]			4
4		1			[4, 4]	5
5		1		[4, 4]		6
6	P: (0,1,2,2) : 5 Q: (0,1,2,3) : 4	0	[4, 4]			7

Figure 5.12: State snapshot of evaluation goal schedule with variable.

holding it until it puts it down again within 5 time units. Figure 5.13 shows the content of evaluation goal schedule and formula cache for an exemplary test sequence that was conducted for 15 steps. At time 3, the outer until operator was "activated" by marking *item1* with 1. This caused the creation of schedule entry 3, which represents, via the rewritten cache entry 4, the top-level instance where the premise of the outer implication is true. Since the marking of *item1* was never changed, each time point beginning from 3 had to be marked as *pending* because the goal ($marking(item1) = 0$) has not been reached yet and the time limit has not been exceeded. The invariant part of the outer until expression is true by default in time points where the expected grab event does not occur. At these points, the inner until expression is not evaluated at all and hence no nested schedule entry is created. However, at time 5, the grab action was performed and the inner until expression enters the picture. Now, a newly rewritten version of the top-level formula is added to the cache and schedule (cache id 8, schedule id 7) in which the invariant part is not any more linked to the plain inner subformula in the second schedule entry (cache id 3) but to the new schedule entry 6 that represents the "active" inner until expression. This entry in turn is connected to the schedule entries 4 and 5 that keep track of the whether the item is carried or not. The time point 5 is registered as a singleton interval in the pending sequence of schedule entry 7 that now acts as an alternative top-level "handler". However, the grab action does not occur in the following steps, so entry 7 remains inactive and entry 3 takes over again.

A closer look at the schedule table reveals that eventually, the item is dropped at time 10. This confirms the goal of the inner until expression for all instances that were scheduled since time 5. In fact, the inner goal is also satisfied for times 11 and 12, which altogether results in the interval $[5, 12]$ in the positive sequence of schedule entry 6. Finally, at time 13, the item is picked up again. As expected, this results in new pending marks for schedule entry 7 and 6.

Even though the example is still quite small, it already demonstrates how the nesting of temporal operators adds complexity to the data structures involved in evaluation. In fact, the main reason why the SALMA property evaluation algorithm is capable of handling complex nested formulas is its consequent use of interval-based representations and operations. This will be explained in detail in the next sections.

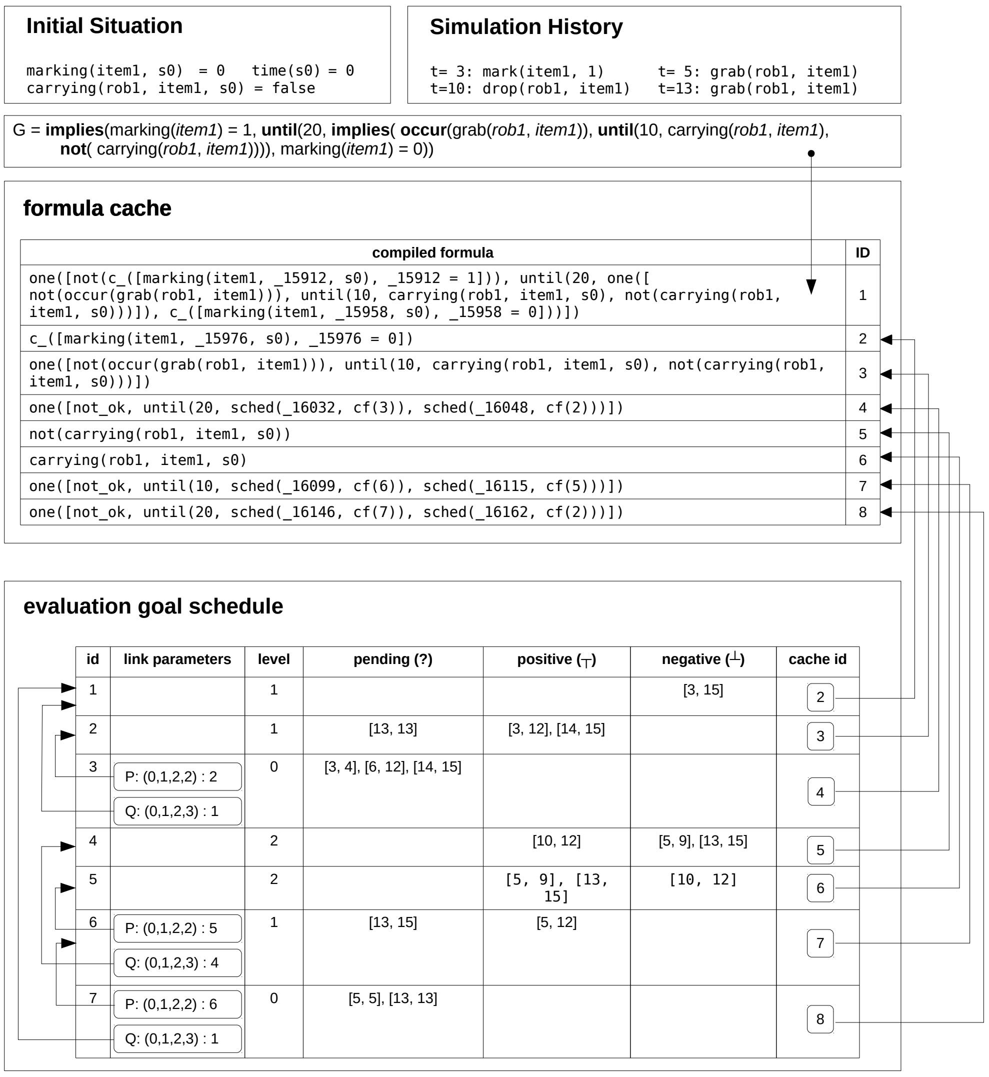
5.5.4 A Formal Interface To The Evaluation Goal Schedule

After all main aspects of the evaluation goal schedule have been introduced above, it is advisable to provide a more concise summary that facilitates referring to parts of the data structure or related mechanisms in the algorithms that will be described below. What is actually needed is a kind of a formal interface with which operators on the evaluation goal schedule can be referred to in an unambiguous and compact way. In doing so, it is practicable to use a representation of the schedule's data structure that can be integrated seamlessly in the imperative description of the algorithms. In fact, the examples above demonstrate that it makes sense to represent the state of each goal instance as a row or tuple in a table similar to a relational database.

Definition 5.23 (Evaluation goal schedule)**.** The evaluation goal schedule can be understood as a set of tuples of which each is identified by a unique id and describes the state of one scheduled goal instance by means of interval sequences and references as described above. Formally, this is written as

$$\mathsf{Sched}_{\mathsf{goal}} = \{G_1, \dots, G_n\}$$

where



Initial Situation

```
marking(item1, s0)  = 0   time(s0) = 0
carrying(rob1, item1, s0) = false
```

Simulation History

```
t= 3: mark(item1, 1)      t= 5: grab(rob1, item1)
t=10: drop(rob1, item1)   t=13: grab(rob1, item1)
```

G = **implies**(marking(*item1*) = 1, **until**(20, **implies**(**occur**(grab(*rob1*, *item1*)), **until**(10, carrying(*rob1*, *item1*), **not**(carrying(*rob1*, *item1*)))), marking(*item1*) = 0))

formula cache

compiled formula	ID
`one([not(c_([marking(item1, _15912, s0), _15912 = 1])), until(20, one([ not(occur(grab(rob1, item1))), until(10, carrying(rob1, item1, s0), not(carrying(rob1, item1, s0)))])), c_([marking(item1, _15958, s0), _15958 = 0]))])`	1
`c_([marking(item1, _15976, s0), _15976 = 0])`	2
`one([not(occur(grab(rob1, item1))), until(10, carrying(rob1, item1, s0), not(carrying(rob1, item1, s0)))])`	3
`one([not_ok, until(20, sched(_16032, cf(3)), sched(_16048, cf(2)))])`	4
`not(carrying(rob1, item1, s0))`	5
`carrying(rob1, item1, s0)`	6
`one([not_ok, until(10, sched(_16099, cf(6)), sched(_16115, cf(5)))])`	7
`one([not_ok, until(20, sched(_16146, cf(7)), sched(_16162, cf(2)))])`	8

evaluation goal schedule

id	link parameters	level	pending (?)	positive ($\top$)	negative ($\perp$)	cache id
1		1			[3, 15]	2
2		1	[13, 13]	[3, 12], [14, 15]		3
3	P: (0,1,2,2) : 2 Q: (0,1,2,3) : 1	0	[3, 4], [6, 12], [14, 15]			4
4		2		[10, 12]	[5, 9], [13, 15]	5
5		2		[5, 9], [13, 15]	[10, 12]	6
6	P: (0,1,2,2) : 5 Q: (0,1,2,3) : 4	1	[13, 15]	[5, 12]		7
7	P: (0,1,2,2) : 6 Q: (0,1,2,3) : 1	0	[5, 5], [13, 13]			8

Figure 5.13: Example for an evaluation goal schedule state with nested temporal operators.

$$G_i = \langle \mathsf{id}_i, \overline{T_{i,?}}, \overline{T_{i,\top}}, \overline{T_{i,\perp}}, \mathsf{level}_i, \mathsf{linkParams}_i, \mathsf{cacheRef}_i \rangle$$

Here, the columns, which will be used without the index i in the following, store all information that is needed to reconstruct the scheduled formula and for calculating the result of temporal operator expressions based on the evaluation history. The first column, id is the unique identifier of the goal in the evaluation goal schedule, $\overline{T_?}$, $\overline{T_\top}$, and $\overline{T_\perp}$ are temporal interval sequences that store the intervals where the evaluation result for the goal instance G_{id} was undetermined, positive, or negative, respectively. Additionally, $\mathsf{cacheRef}$ holds a pointer that references an entry in the formula cache which contains the goal's formula. How deep this (sub-)formula is nested within temporal operators is indicated by the column *level*. Finally, linkParams contains subterm position paths and the concrete sub-goal id's that are used to construct the history of goals with possibly nested temporal operators (see Section 5.5.1, paragraph in the middle of page 149)).

In order to refer to the contents of a goal, a dot notation can be used that is adopted from object-oriented programming languages. For instance, if a goal is denoted by G, the interval sequences can be accessed with $G.\overbracket{T_?}$, $G.\overbracket{T_\top}$, and $G.\overbracket{T_\bot}$, respectively. Additionally, several functions are necessary for the retrieval and manipulation of information in the schedule. Some of these functions are of rather technical nature, and their semantics are hence presented in a more informal style. In the following list, GOALS is the set of all evaluation goals, $\mathcal{ID}_{sched}$ represents the set of all possible evaluation goal schedule ids, $Formula_{PSL}$ the set of all possible SALMA-PSL formulas, $\overset{res}{\overbracket{\mathcal{R}}}$ the set of all result mappings, $\mathcal{ID}_{cache}$ the set of all cache ids, and $Paths$ the set of all possible position paths. With these conventions, the interface to the evaluation goal schedule is modeled as follows:

$\mathsf{loadFromSchedule}(id) : \mathcal{ID}_{sched} \rightarrow \mathsf{GOALS}$

Loads the evaluation goal with the given id from the schedule.

$\mathsf{applyLinkParameters}(id) : \mathcal{ID}_{sched} \rightarrow Formula_{PSL}$

Returns the instantiated formula of the goal with the given id. The returned formula is created by substituting the link parameters for the placeholders in the `sched` parts of the goal's formula in the cache.

$\mathsf{applyDecisions}(id, \overset{res}{\overbracket{R}}) : \mathcal{ID}_{sched} \times \overset{res}{\overbracket{\mathcal{R}}}$

Updates the evaluation goal schedule entry with the given id according to the new results in the given mapping. More precisely, each interval with a fixed decision in $\overset{res}{\overbracket{R}}$ "moves" this segment from the undetermined interval sequence $\overbracket{T_?}$ from the schedule to one of the sequences $\overbracket{T_\top}$ or $\overbracket{T_\bot}$, according to the label in the result mapping. Formally, when $\overset{res}{\overbracket{\mathcal{R}}}$ denotes the set of all possible result mappings and G'_{id} is the state of the scheduled goal after the application of the results, then the effect applyDecisions can be defined as follows:

$$\langle id, \overbracket{T_?}, \overbracket{T_\top}, \overbracket{T_\bot}, \mathsf{lev}, \mathsf{IP}, \mathsf{cR}\rangle \xrightarrow{\mathsf{applyDecisions}(id, \overset{res}{\overbracket{R}})} \langle id, \overbracket{T'_?}, \overbracket{T'_\top}, \overbracket{T'_\bot}, \mathsf{lev}, \mathsf{IP}, \mathsf{cR}\rangle$$

where

$$\mathcal{I}^r = \{I \in \mathcal{I}_T(\overset{res}{\overline{R}}.\mathsf{intv}) \mid \overset{res}{\overline{R}}.\mathsf{rmap}(I) = r\} \qquad \text{for } r \in \{\top, \bot, ?\}$$

$$\overline{T'_?} = \left(\bigcup_{I \in \mathcal{I}^\top \cup \mathcal{I}^\bot} \overline{T_?} \setminus I \right) \cup \left(\bigoplus_{I \in \mathcal{I}^?} I \right)$$

$$\overline{T'_\top} = \overline{T_\top} \cup \left(\bigoplus_{I \in \mathcal{I}^\top} I \right) \quad \text{and} \quad \overline{T'_\bot} = \overline{T_\bot} \cup \left(\bigoplus_{I \in \mathcal{I}^\bot} I \right)$$

$\mathsf{storeInSchedule}(lev, cId, lp) : \mathbb{N}_0 \times \mathcal{ID}_{cache} \times \mathcal{P}(Paths \times \mathcal{ID}_{sched}) \to \mathcal{ID}_{sched}$

Creates a new entry in the evaluation goal schedule that points to a formula with id cId in the formula cache which is nested at level lev. Additionally, the link parameters in lp are set up in the new goal, which potentially establishes the links seen in the examples above. Here, $Paths$ denotes the set of possible subterm position paths and $\mathcal{ID}_{cache}$ the set of ids in the formula cache. The result of a call to storeInSchedule is the unique id that was assigned to the newly created entry.

$\mathsf{addNondetScheduleInterval}(id, interval) : \mathcal{ID}_{sched} \times \mathcal{I}_T$ Adds the given interval to the sequence $\overline{T_?}$ of the evaluation goal schedule entry with the given id. In other words, it is declared that the evaluation result for each time point within the given interval is undetermined. This also effectively means that obligations to re-visit the property are added for each of these time instances. In terms of the operations defined in Section 5.3, this can simply be written as

$$\langle id, \overline{T_?}, \overline{T_\top}, \overline{T_\bot}, \mathsf{lev}, \mathsf{IP}, \mathsf{cR} \rangle \xrightarrow{\mathsf{addNondetScheduleInterval}(id,I)} \langle id, \overline{T_?} \oplus I, \overline{T_\top}, \overline{T_\bot}, \mathsf{lev}, \mathsf{IP}, \mathsf{cR} \rangle$$

With these definitions in place, it is now possible to express usages of the evaluation goal schedule in a precise way that does not add unnecessarily clutter to the presentation of the algorithms in the following section.

5.6 The Formula Evaluation Algorithm

The previous sections set the stage for the description of the main algorithms that are responsible for the evaluation of SALMA-PSL formulas. The algorithms are presented in a rather detailed fashion in order to allow the reader to follow the involved mechanism throughout the whole evaluation process. In a manner of speaking, this section is structured from the outside to the inside, starting with the top-level procedure that governs the evaluation process, and then recursing downwards to the specialized algorithms that interpret the individual constructs of the SALMA-PSL language. In each of the following sections, notational elements as well as auxiliary terms and concepts are introduced as required. First, however, it makes sense to establish some general definitions that are used repeatedly throughout the algorithm descriptions.

5.6.1 General Definitions

One of the mechanisms that appears at several points in the following algorithms, and that was already implicitly used above, is yet another form of subterm substitution. Besides the unification-based substitution mechanism introduced in Definition 5.20, in which the replaced subterm is located through pattern matching, it is also necessary to be able to define the relevant position within the path directly. The best way to do this is using a *position path* that recursively specifies the indexes of the arguments that have to be traversed in order to reach the desired location. In fact, this is obviously exactly what has already been used for specifying the *link parameters* in the examples of Section 5.5. Here, a formal definition is added in order to allow a detailed understanding of the way in which formulas are manipulated during the evaluation process.

Definition 5.24 (Subterm substitution based on position paths)**.**
Let $\Phi = \phi(\theta_1, \ldots, \theta_n)$ be a term with n-ary functor ϕ. Furthermore, let $\hat{p}_1 = (p_{1,1}, \ldots, p_{1,l_1})$ to $\hat{p}_m = (p_{m,1}, \ldots, p_{m,l_m})$ with $\forall i, j. p_{i,j} \in \mathbb{N}_0$ be sequences of natural numbers, called *position paths.* For all $\hat{p}_1, \ldots, \hat{p}_m$, it shall hold that no position path is a prefix of the other. In that sense, let $\Psi = \{\hat{p}_1 \mapsto \theta_1^*, \ldots, \hat{p}_m \mapsto \theta_m^*\}$ denote a mapping that assigns the *replacement terms* $\theta_1^*, \ldots, \theta_m^*$ to the positions $\hat{p}_1, \ldots, \hat{p}_m$. Finally, let c denote a constant, i.e. a number, an entity, or a function symbol with arity 0. Then the substitution of the subterms at positions $\hat{p}_1, \ldots, \hat{p}_m$ with the corresponding replacement terms, written $\text{subst}_{\text{pos}}(\Psi, \Phi)$, is recursively defined as follows:

$$
\begin{aligned}
\mathrm{subst_{pos}}\left(\Psi, c\right) &= c \\
\mathrm{subst_{pos}}\left(\{(0) \mapsto \theta^*\}, \Phi\right) &= \theta^* \\
\mathrm{subst_{pos}}\left(\{(0, p_2, \ldots, p_l) \mapsto \theta^*\}, \Phi\right) &= \mathrm{subst_{pos}}\left(\{(p_2, \ldots, p_l) \mapsto \theta^*\}, \Phi\right) \\
\mathrm{subst_{pos}}\left(\{(p_1, p_2, \ldots, p_l) \mapsto \theta^*\},\right. &= \phi(\theta_1, \ldots, \theta_{p_1 - 1}, \\
\left.\phi(\theta_1, \ldots, \theta_n)\right) &\qquad \mathrm{subst_{pos}}\left(\{(p_2, \ldots, p_l) \mapsto \theta^*\}, \theta_{p_1}\right), \\
&\qquad \theta_{p_1+1}, \ldots, \theta_n) \qquad \text{if } p_1 > 0 \\
\mathrm{subst_{pos}}\left(\{(p_1) \mapsto \theta^*\}, \phi(\theta_1, \ldots, \theta_n)\right) &= \phi(\theta_1, \ldots, \theta_{p_1 - 1}, \theta^*, \theta_{p_1+1}, \ldots, \theta_n) \\
\mathrm{subst_{pos}}\left(\{\hat{p} \mapsto \theta^*\} \cup \Psi, \Phi\right) &= \mathrm{subst_{pos}}\left(\Psi, \mathrm{subst_{pos}}\left(\{\hat{p} \mapsto \theta^*\}, \Phi\right)\right)
\end{aligned}
$$

Definition 5.24 is relevant for almost every algorithm below because the position path of the currently evaluated subformula is constructed alongside the recursive evaluation procedure and passed through as an argument of almost every function. Eventually, the generated path is used for rewriting the formula according to gathered results. Merely a convention is the use of 0 as a "pseudo-position" that refers to the entire term, which can be seen in the second and third equation of Definition 5.24.

Another convention that is ubiquitously used in the algorithms of this section is the way in which return values of functions are handled. Many of these functions return more than one value at once, e.g. the evaluation result of the current subformula together with additional information that will be needed for adding an entry to the goal schedule. To represent this, the pseudo-code dialect of the algorithm descriptions uses *tuples*, both for the specification of the returned value and for the assignment of a structured function result to multiple variables. For instance, if a function f returns three values simultaneously, the function definition will contain a statement like **return** $\langle x, y, z\rangle$ and elsewhere in another algorithm, a call to f that stores returned values in the variables a, b, and c, could be written as $\langle a, b, c\rangle \leftarrow f(\ldots)$. Since the names of the variables identify their purpose, some of the variables on the assignment side are omitted on occasion if they are not required for the further computation.

As mentioned above, only those notational elements were defined here that occur repeatedly in the following algorithm. Other elements will be defined when they appear first or the reader is referred to definitions in previous sections. With this in mind, it is now time to approach the main entry point of SALMA's property evaluation mechanism.

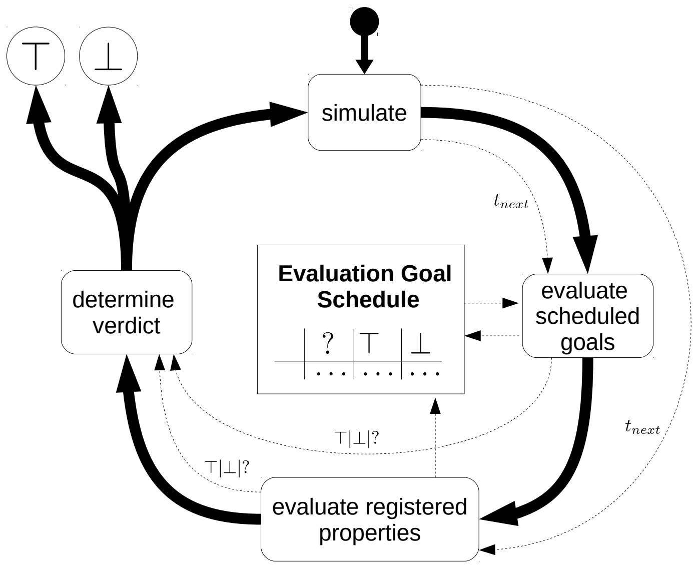


Figure 5.14: The SALMA property evaluation loop.

5.6.2 The Property Evaluation Loop

The property evaluation process in SALMA can best be understood as a loop that integrates the simulation step, the processing of the evaluation goal schedule, and the evaluation of the registered invariants and goals. This loop is visualized in Figure 5.14.

Each cycle of the loop is entered with a simulation step that is performed according to the semantics as described in Section 3.6. At the end of the step, the simulation algorithm determines the time advance, i.e. the time of the next scheduled event (t_{next} in Figure 5.14). As explained in Section 5.2, t_{next} is passed as a parameter to the evaluation phase where it marks the end of the interval for which the properties have to be tested. In the evaluation phase itself, first the entries in the evaluation goal schedule are processed as described in Section 5.5. Then, the algorithm iterates over all invariants and goals in the property registry and evaluates them for all time points from the current up to t_{next}. For both the scheduled goals and the new properties taken directly from the registry, the evaluation produces a collection of results that are combined to determine a verdict for the current step. This could lead to the termination of the current simulation run if an invariant is violated or all goals are met. Otherwise, the loop proceeds with the next simulation step.

The evaluation phase of the loop is described in Algorithm 5.1. First, it is worth noticing that the goals from the schedule are processed in an order where goals with a higher nesting level are evaluated before those with lower nesting levels. This ensures that, when a formula part with a temporal operator is evaluated, the intervals in the schedule that represent the subformulas of the operator have been updated already. For each of the scheduled goals, first the actual formula is restored from the cache by applying the link parameters. Then, this formula is evaluated by a call of the function $evaluateFormula$. The end of the time advance, t_{end}, is passed as the last parameter to determine the time point up to which the evaluation should be performed. Additionally, the interval sequence $G.\overline{T_?}$, which comprises the time instances for which no conclusive result could be found for the property, is passed to provide the set of start times of the goal instances that should be tested. Consequently, the call to $evaluateFormula$ returns a *result mapping* that assigns a result to each of the intervals in $G.\overline{T_?}$. The result mappings for each goal are collected in the associative list $\mathcal{R}_{sched}$.

Next, the properties in the property registry are processed one by one. Each property entry contains a unique user-defined name (*P.name*), which is used to interpret results, as well as a reference to the formula cache ($P.Id_{cache}$). With this reference, the formula is loaded from the cache using the function `loadFromCache`, which is not described in detail due to its purely technical nature. After that, the type of the property is determined, i.e. whether it is an invariant or a goal (see Definition 4.7 on page 97). According to this distinction, the property is evaluated iteratively *for all time steps* between the current time T_{cur} and t_{next} (see Algorithm 5.2). This yields yet another result which is stored in a second associative list $\mathcal{R}_{instant}$. Additionally, the function `evaluateForAllTimesteps` ensures that entries for the evaluation goal schedule are created for all property instances for which no conclusively result could be determined. Finally, the result mappings in $\mathcal{R}_{sched}$ and $\mathcal{R}_{instant}$ are returned and will be used by the simulation framework to determine a verdict for the current simulation run.

The following subsections will describe all important ingredients of the property evaluation loop. Most of the algorithmic details are covered in the main formula evaluation function `evaluateFormula` in Algorithm 5.4 and the sub-algorithms that handle the different cases during the recursion on the formula structural. However, before these details are addressed, it is helpful to take a closer look at the second phase of Algorithm 5.1 in which registered properties are evaluated in a lookahead manner and entries for the evaluation goal schedule are created.

Algorithm 5.1: Top-level property evaluation procedure.

```
evaluationStep : $\mathbb{N}_0 \to \langle \mathfrak{Res}, \mathfrak{Res} \rangle$

function evaluationStep($t_{next}$)
    $Goals \leftarrow$ all scheduled property goals, $sorted\ by\ goal\ level\ in$
            $descending\ order$
    $\mathcal{R}_{sched} \leftarrow \emptyset$
    foreach $G \in Goals$ do
        $\Phi \leftarrow$ applyLinkParameters($G$.id)
        $\overset{res}{\overleftrightarrow{R}} \leftarrow$ evaluateFormula($\Phi$, $(0)$, $G$.level, $0$, $G.\overleftrightarrow{T?}$, $t_{next}$)
        applyDecisions($G$.id, $\overset{res}{\overleftrightarrow{R}}$)          // updates goal schedule
        $\mathcal{R}_{sched} \leftarrow \mathcal{R}_{sched} \cup \{G \mapsto \overset{res}{\overleftrightarrow{R}}\}$
    end
    $T_{cur} \leftarrow time(S_0)$
    $\mathcal{R}_{instant} \leftarrow \emptyset$
    foreach $P \in Properties$ do
        $\Phi \leftarrow$ loadFromCache( $P.Id_{cache}$)
        if $\Phi = invariant(\phi)$ then
            $mode \leftarrow \Box$
        else
            assume $\Phi = goal(\phi)$
            $mode \leftarrow \Diamond$
        end
        $R_P \leftarrow$ evaluateForAllTimesteps($mode$, $\phi$, $(0)$, $0$, $0$, $T_{cur}$,
                                          $t_{next}$, $-1$, $-1$)
        $\mathcal{R}_{instant} \leftarrow \mathcal{R}_{instant} \cup \{P \mapsto R_P\}$
    end
    return $\langle \mathcal{R}_{sched}, \mathcal{R}_{instant} \rangle$
end
```

5.6.3 Lookahead Evaluation of Invariants And Goals

As explained above, all properties that are registered as invariant or goals have to be evaluated for multiple instances, in which the referenced start time is iteratively set to all points between the current time and the end of the time-advance. Therefore, the algorithm must employ a *lookahead technique* in order to evaluate formulas in the context of world states that have not been calculated by the simulation yet. The basic mechanism for evaluating such future states is to replace the default situation argument S_0 with situation terms that incorporate the time advance, i.e. with terms like $do(tick(\Delta T), S_0)$. Although it would seem obvious to use the substitution mechanism defined in Definition 5.20, this would also affect the content of nested temporal operators that by themselves trigger iterations over future time steps. Therefore, an alternative version of the substitution operator is needed that stops at temporal operators.

Definition 5.25 (Subterm substitution with boundaries)**.** The subterm substitution with boundaries $subst^b_{term}(\Psi, \Theta)$ works almost identically as the substitution defined in Definition 5.20 but does not recurse into occurrences of temporal operators. Formally, this is expressed as follows:

1. $subst^b_{term}(\Psi, c) = \begin{cases} \theta' & \text{if } \exists \theta . \theta \simeq c \wedge \theta \mapsto \theta' \in \Psi \\ c & \text{otherwise} \end{cases}$

2. $\begin{array}{l} subst^b_{term}(\Psi, \\ \quad \phi(\theta_1, \ldots, \theta_n)) \end{array} = \begin{cases} \phi(\theta_1, \ldots, \theta_n) & \text{if } \phi \in \{always, \\ & \qquad eventually, until\} \\ \phi(subst^b_{term}(\Psi, \theta_1), \ldots, & \text{otherwise} \\ \qquad subst^b_{term}(\Psi, \theta_n)) & \end{cases}$

3. $subst^b_{term}(\{\theta \mapsto \theta'\} \cup \Psi, \Theta) = subst^b_{term}(\{\theta \mapsto \theta'\}, subs^b_{term}(\Psi, \Theta))$

With this means for manipulating the situation context of a formula, the lookahead iteration can be realized as shown in Algorithm 5.2. The function `evaluateForAllTimesteps` expects several parameters: *mode* defines whether the given formula is interpreted as an invariant (marked by a $\Box$) or as a goal ($\Diamond$). Then, the (partial) formula itself is passed in Φ together with its nesting *level* and a position path in $\hat{p}_\Phi$ that locates Φ within the top-level formula. Additionally, T_{start} defines the start time that will be used as a reference point for temporal operators contained in Φ, and T_{end} defines the upper time bound of the iteration. When `evaluateForAllTimesteps` is used within the top-level evaluation loop, these parameters correspond to the current time and the end of the time-advance, respectively. As it will be explained in Section 5.6.10 and below, this can differ when the function is called within the evaluation of temporal operators. The last two arguments, Id_{sched} and Id_{cache} hold the ids that are associated with the currently evaluated formula in the evaluation goal schedule and in the cache, respectively. In case of the direct evaluation of properties from the registry, as in Algorithm 5.1 above, these ids do not exist yet and are therefore set to -1 as a signal for the function `evaluateAndSchedule` that new entries must be created if scheduling is necessary.

At the start of Algorithm 5.2, the variables t, Id'_{sched}, and Id'_{cache}, are initialized directly with start values from the parameters. Besides that, there are three variables that model the result of the iterative evaluation. First, $T^{<}_{def}$ will contain the earliest time point in the iteration at which a definite positive outcome is determined. Similarly, $T^{>}_{poss}$ represents the latest time point for which it is still possible that Φ eventually evaluates to true. Both of these time markers are not used in the main evaluation loop of Algorithm 5.1 but will be important later for the efficient evaluation of temporal operators. Their initial value is set to ?, which means in this case that no value has been set yet. The third variable, R, is used to store a *summarized result* over all iteration steps. Depending on whether *evaluateForAllTimesteps* is used in goal ($\Diamond$) or invariant mode ($\Box$), R is initialized with a default value either with $\bot$ or $\top$, which will make sense once the design of the iteration below is understood.

The loop in Algorithm 5.2 iterates through all time steps between T_{start} and T_{end} by increasing the variables t and *step*. In the loop, a new situation term S is constructed that manifests a time advance of the number of time steps currently held by *steps*. This situation term is inserted in Φ using the bounded subterm substitution mechanism of Definition 5.25. The resulting formula is then passed over to the function `evaluateAndSchedule`, which evaluates the formula and creates new entries in the evaluation goal schedule and formula cache as necessary. The tuple that this function returns contains most importantly the evaluation result for the current step itself, which is stored in R_{cur}. Besides, it contains values for Id_{sched} and Id_{cache} that will differ from the ids passed to *evaluateForAllTimesteps* originally when a new entry has been created. These new pointers are stored in the loop variables Id'_{sched} and Id'_{cache} and are used in subsequent iteration steps. Therefore, the results from evaluating the following property instances will be integrated in the interval lists of the existing schedule entry rather than in new ones (cf. Algorithm 5.3 below).

After Φ' has been evaluated, the result of the current step R_{cur} is used to update the result summary R. It can be seen that the loop is canceled as soon as a conclusive result is determined. For an undetermined outcome (?), the result summary R is set to ?. For the other cases, i.e. a positive outcome in invariant mode or a negative outcome in goal mode, R is left untouched. This means that as soon as one of the evaluation steps yields a ? outcome, R remains ? until one of the unambiguous cases mentioned above flips R to $\top$ or $\bot$. The values of $T^{<}_{def}$ and $T^{>}_{poss}$ are also updated in an intuitive way:

Algorithm 5.2: Iterative evaluation for all time steps.

```
function evaluateForAllTimesteps($mode$, $\Phi$, $\hat{p}_\Phi$, $level$,
                                  $T_{start}$, $T_{end}$, $Id_{sched}$, $Id_{cache}$)
    $t \leftarrow T_{start}$
    $step \leftarrow 0$
    $Id'_{sched} \leftarrow Id_{sched}, Id'_{cache} \leftarrow Id_{cache}$
    $T^{<}_{def} \leftarrow T^{>}_{poss} \leftarrow ?$
    $R \leftarrow \begin{cases} \bot & \text{if } mode = \Diamond \\ \top & \text{otherwise} \end{cases}$
    while $t \leq T_{end}$ do
        $S \leftarrow do(tick(step), S_0)$
        $\Phi' \leftarrow subst^b_{term}(\{S_0 \mapsto S\}, \Phi)$
        $\langle R_{cur}, Id'_{sched}, Id'_{cache}\rangle \leftarrow$ evaluateAndSchedule($\Phi'$, $\hat{p}_\Phi$, $level$,
                                  $step$, $t$, $T_{end}$, $Id'_{sched}$, $Id'_{cache}$)
        if $R_{cur} = \bot$ then
            if $mode = \Box$ then
                $R \leftarrow \bot$
                if $T^{>}_{poss} = ?$ then $T^{>}_{poss} \leftarrow t - 1$
                break
            end
        else if $R_{cur} = \top$ then
            if $T^{<}_{def} = ?$ then $T^{<}_{def} \leftarrow t$
            $T^{>}_{poss} \leftarrow t$
            if $mode = \Diamond$ then
                $R \leftarrow \top$
                break
            end
        else                                                /* undecided */
            $R \leftarrow ?$
            $T^{>}_{poss} \leftarrow t$
        end
        $t \leftarrow t + 1, step \leftarrow step + 1$
    end
    return $\langle R, T^{<}_{def}, T^{>}_{poss}, Id'_{sched}, Id'_{cache}\rangle$
end
```

both $\top$ and ? imply that a positive result is possible in this step; on the other hand, when a negative result is found for an invariant at a certain time point, it is clear that the step before the current step has to be the last one where a positive outcome is even possible.

When the loop has been left, it must either have been canceled because a definite result was found as described above, or the iteration has reached T_{end}. In either cases, the function returns the result together with the time markers and the possibly updated schedule and cache ids.

As mentioned before, the function `evaluateForAllTimesteps` is not only responsible for testing whether an invariant holds or a goal is reached during a certain timespan but also for creating schedule entries for those points between T_{start} and T_{end} for which the outcome is not certain yet. This happens in the function `evaluateAndSchedule` shown in Algorithm 5.3, which is in fact the only place where new entries are added to the evaluation goal schedule.

The parameters passed to the function `evaluateAndSchedule` are the known components of the formula description, namely Φ, $\hat{p}_\Phi$, and *level*, together with the current time in T_{cur} and the end of the time-advance in T_{end}. In addition to the absolute time parameters, there is also the parameter s_{cur} that contains the number of steps taken relative to S_0 in the iteration within `evaluateForAllTimesteps`. Additionally, the parameters Id_{sched} and Id_{cache} can either contain existing ids for the evaluation goal schedule and the formula cache or markers that control the creation of new entries as described below.

The first step in Algorithm 5.3 is to actually evaluate Φ using the main formula evaluation function that will be discussed in detail in the following subsections. For that purpose, a new "singleton" temporal interval sequence $\overline{T_s}$ is created that contains only the current time T_{cur}. The returned tuple of `evaluateFormula` contains a result, a possibly rewritten version of Φ and link parameters in the sense of Section 5.5 that are assigned to the variables R_{ov}, Φ', and *linkParams*, respectively. Based on the result and the values in Id_{sched} and Id_{cache}, the algorithm decides whether a new entry in the formula cache is created or an existing cache entry is updated. Basically, a new entry in created only if the Id_{cache} is either *new* or -1, which indicates that no entry for Φ exists yet, or the formula was rewritten, i.e. $\Phi' \neq \Phi$. Intuitively, caching is generally only necessary when a schedule entry has to be created because the result is unclear yet, i.e. when $R_{ov} =?$. However, for the evaluation of the *until* operator (see Section 5.6.10), the algorithm relies on the schedule history of the *until* operator's subformulas also to query for definite results. Therefore, the special *new* flag is needed to force caching and scheduling even if R_{ov} is $\top$ or $\bot$. If the algorithm decides that a new cache entry is needed, it first restores the situation argument that was rewritten in the calling function `evaluateForAllTimesteps` (see Algorithm 5.2) and then use `storeInCache`, which adds a new cache entry and returns the newly created cache id.

As mentioned before, a new entry in the evaluation goal schedule is created either because a) this is enforced by the flag *new*, or b) because the evaluation

Algorithm 5.3: Main evaluation goal scheduling mechanism.

```
function evaluateAndSchedule($\Phi$, $\hat{p}_\Phi$, $level$, $s_{cur}$, $T_{cur}$, $T_{end}$,
$Id_{sched}$, $Id_{cache}$)
    $\overleftarrow{T_s} \leftarrow ([T_{cur}, T_{cur}])$
    $\langle R_{ov}, \Phi', linkParams \rangle \leftarrow$ evaluateFormula($\Phi$, $\hat{p}_\Phi$, $level$, $s_{cur}$,
                                                  $\overleftarrow{T_s}$, $T_{end}$)

    $\Phi_{cache} \leftarrow \begin{cases} \Phi' & \text{if } R_{ov} =? \wedge (Id_{cache} \in \{new, -1\} \vee \Phi' \neq \Phi) \\ \Phi & \text{if } Id_{cache} = new \wedge R_{ov} \in \{\top, \bot\} \\ none & \textbf{otherwise} \end{cases}$

    if $\Phi_{cache} \neq none$ then
        /* Undo situation substitution done in
           evaluateForAllTimesteps.                                      */
        $\Phi'_{cache} \leftarrow subst^b_{term}(\{do(*) \mapsto S_0\}, \Phi_{cache})$
        $Id'_{cache} \leftarrow$ storeInCache($\hat{p}_\Phi$, $\Phi'_{cache}$)
    else
        $Id'_{cache} \leftarrow Id_{cache}$
    end
    if $Id_{sched} = new \vee (Id_{sched} = -1 \wedge R_{ov} =?) \vee Id'_{cache} \neq Id_{cache}$ then
        $Id'_{sched} \leftarrow$ storeInSchedule($level$, $Id'_{cache}$, $linkParams$)
    else
        $Id'_{sched} = Id_{sched}$
    end
    if $Id'_{sched} \neq -1$ then
        if $R_{ov} =?$ then
            addNondetScheduleInterval($Id'_{sched}$, $[T_{cur}, T_{cur}]$)
        else
            applyDecisions($Id'_{sched}$, $\{[T_{cur}, T_{cur}] \mapsto R_{ov}\}$)
        end
    end
    return $\langle R_{ovs}, Id'_{sched}, Id'_{cache} \rangle$
end
```

result is undetermined and no schedule entry exists yet, or c) because a new cache entry was created and hence the cache reference of the existing schedule entry is obsolete. In any case, `storeInSchedule` will not change any existing schedule entry but create a new one that could exist next to other entries that are linked to different versions of the same formula.

After the algorithm has ensured that a schedule entry exists if necessary, the current time point is added to the proper interval sequence using one of the functions `addNondetScheduleInterval` or `applyDecisions` that were introduced in Section 5.5.4. At this point, the schedule is updated properly and both the result and the possibly new cache and schedule ids are returned to the caller.

5.6.4 Main Formula Evaluation Function

Now that the outer structure of the evaluation process has been explained, it remains to discuss how formulas are actually interpreted. As mentioned before, the main function that controls the recursive evaluation on a formula is `evaluateFormula`, which is shown in Algorithm 5.4. The parameters of `evaluateFormula` contain the familiar Φ, $\hat{p}_\Phi$, *level*, s_{cur}, and T_{end} that already appeared in previous algorithms. Additionally, the temporal interval sequence $\overbracket{T_s}$ contains the time points for which Φ should be evaluated.

The basic structure of `evaluateFormula` consist of a distinction between a series of cases, most of which are not shown in Algorithm 5.4 itself but in one of the function fragments that are discussed in the following subsections. The first case in Algorithm 5.4 applies when Φ is a result itself. This is possible for formulas in the evaluation goal schedule that have been rewritten before scheduling (cf. Section 5.5). In this case, the obtained single result does not depend on the start time and is therefore assigned to all time points in $\overbracket{T_s}$, using the unique assignment operator $|_\Phi$ from Definition 5.13, to produce the result mapping $\overset{res}{\overbracket{R}}$ that will be returned eventually. Besides $\overset{res}{\overbracket{R}}$, `evaluateFormula` produces three other return values: R_{ov}, which holds an "overall result", i.e. a summary over $\overset{res}{\overbracket{R}}$, Φ', a possibly rewritten version of Φ that will be used for scheduling, and *linkParams*, which contains links to subgoals in the evaluation goal schedule as described in Section 5.5.

In all cases except Φ being a concrete result, Φ has to have the form $\phi(\theta_1, \ldots, \theta_n)$, i.e. an expression with a functor ϕ and subterms θ_1 to θ_n. Similar to Section 5.4, this also includes expressions with operators for arithmetics or comparisons that are originally used in infix notation, e.g. $\theta_1 > \theta_2$, which is understood as $>(\theta_1, \theta_2)$. Additionally, unary functions and predicates are included, e.g. *time*. In all cases, the responsible function segments produce the required return values $\overset{res}{\overbracket{R}}$, R_{ov} and *linkParams*. Additionally, they might populate the position-replacement map Ψ that is used in a position-path-based subterm substitution (see Definition 5.24) to produce a rewritten version of Φ which is returned for scheduling.

Algorithm 5.4: Main formula evaluation function.

```
function evaluateFormula($\Phi$, $\hat{p}_\Phi$, $level$, $s_{cur}$, $\overbracket{T_s}$, $T_{end}$)
    $T_{cur} \leftarrow time(do(tick(s_{cur}), S_0))$
    if $\Phi \in \{\top, \bot\}$ then
        $\overset{res}{\overbracket{R}} \leftarrow \overbracket{T_s}|_\Phi$
        $R_{ov} \leftarrow \Phi$, $\Phi' \leftarrow \Phi$, $linkParams \leftarrow \emptyset$
    else
        assume $\Phi = \phi(\theta_1, \ldots, \theta_n)$
        switch $\Phi$ do
            $\langle \overset{res}{\overbracket{R}}, R_{ov}, \Psi, linkParams \rangle \leftarrow$ handle cases for $\Phi$ in Algorithms
                                                  5.5, 5.7, 5.8, and 5.11
            $\Phi' \leftarrow \begin{cases} R_{ov} & \textbf{if } R_{ov} \in \{\top, \bot\} \\ subst_{pos}(\Psi, \Phi) & \textbf{otherwise} \end{cases}$
        endsw
    end
    return $\langle \overset{res}{\overbracket{R}}, R_{ov}, \Phi', linkParams \rangle$
end
```

The following subsections will discuss all cases in detail. The fragments that are presented in Algorithms 5.5, 5.7, 5.8, and 5.11 use the variables that are set up in `evaluateFormula`. Therefore, they prepare the values that are eventually returned to the caller, i.e. either the function `evaluateFor-AllTimesteps` or the schedule processing phase in `evaluationStep`.

5.6.5 Logical Connectives

An obvious choice for starting the case analysis within logical formulas is the treatment of the logical connectives, i.e. negations, conjunctions, and disjunctions. Algorithm 5.5 shows the corresponding fragment within `evaluateFormula`. It comprises the cases for *not*, *all*, *one*, and also c_{-}, which is a special synonym for *all* that marks a coherent evaluation unit. As described in Section 5.4, implications are actually translated by means of the equivalence $A \implies B \equiv \neg A \vee B$, so they do not need to be handled separately.

The first case in Algorithm 5.5 is the negation operator. Here, the nested formula is evaluated as normal, and the result is negated using the operator defined in Definition 5.15. Additionally, the result summary R_{ov} is negated if

Algorithm 5.5: Evaluation of logical connectives.

```
. . .
case $\Phi = not(\theta)$
    $\langle \overset{res}{\overline{R'}}, R'_{ov}, \theta', linkParams\rangle \leftarrow$ evaluateFormula($\theta$, $\hat{p}_\Phi \circ 1$, $level$,
                                    $s_{cur}$, $\overline{T_s}$, $T_{end}$)
    $\overset{res}{\overline{R}} \leftarrow \neg \overset{res}{\overline{R'}}$
    $R_{ov} \leftarrow \begin{cases} \neg R'_{ov} & \textbf{if } R'_{ov} \in \{\top, \bot\} \\ R'_{ov} & \textbf{otherwise} \end{cases}$
    $\Psi \leftarrow \{(1) \mapsto \theta'\}$
end
case $\Phi = all([\theta_1, \dots, \theta_n]) \vee \Phi = c\_([\theta_1, \dots, \theta_n])$
    $\langle \overset{res}{\overline{R}}, R_{ov}, \Psi, linkParams\rangle \leftarrow$ evaluateConOrDisjunction(
                                    $\wedge$, $(\theta_1, \dots, \theta_n)$, $\hat{p}_\Phi$, $level$, $s_{cur}$, $\overline{T_s}$, $T_{end}$)
end
case $\Phi = one([\theta_1, \dots, \theta_n])$
    $\langle \overset{res}{\overline{R}}, R_{ov}, \Psi, linkParams\rangle \leftarrow$ evaluateConOrDisjunction(
                                    $\vee$, $(\theta_1, \dots, \theta_n)$, $\hat{p}_\Phi$, $level$, $s_{cur}$, $\overline{T_s}$, $T_{end}$)
end
. . .
```

the result was definite, i.e. either $\top$ or $\bot$. Finally, the rewritten subterm θ' is established as the single replacement term of the substitution mapping Ψ.

For the evaluation of conjunctions and disjunctions, the algorithm uses the function `evaluateConOrDisjunction` from Algorithm 5.6, which expects the operator as first argument. In Algorithm 5.6, every subformula in the encompassed list is evaluated and the results are combined into one single result mapping. For that, one of the logical operators $\wedge$ or $\vee$ for result mappings is used, which were introduced in Definition 5.16. The aggregation of results is achieved by using a separate result mapping $\overset{res}{\overline{R}}$ that is updated in every iteration step by a conjunction or disjunction between itself and the result mapping produced in the formula evaluation of the current step, using the operators $\wedge$ and $\vee$ from Definition 5.16).

Besides the calculation of the result, Algorithm 5.6 also compiles a substitution set Ψ in which each subformula θ_i is either replaced by its rewritten

Algorithm 5.6: Evaluation of conjunctions and disjunctions.

```
function evaluateConOrDisjunction(
                    $op$, $(\theta_1, \ldots, \theta_n)$, $\hat{p}_\Phi$, $level$, $s_{cur}$, $\overbracket{T_s}$, $T_{end}$)
    $\overset{res}{\overbracket{R}} \leftarrow \begin{cases} \overbracket{T_s}|_\top & \text{if } op = \wedge \\ \overbracket{T_s}|_\bot & \text{if } op = \vee \end{cases}$
    $\Psi \leftarrow \emptyset$
    $linkParams \leftarrow \emptyset$
    foreach $i \in [1, n]$ do
        $\langle \overset{res}{\overbracket{R'}}, R'_{ov}, \theta', linkParams' \rangle \leftarrow$ evaluateFormula($\theta_i$, $\hat{p}_\Phi \circ 1 \circ i$,
                                        $level$, $s_{cur}$, $\overbracket{T_s}$, $T_{end}$)
        if $op = \wedge$ then $\overset{res}{\overbracket{R}} \leftarrow \overset{res}{\overbracket{R}} \wedge \overset{res}{\overbracket{R'}}$ else $\overset{res}{\overbracket{R}} \leftarrow \overset{res}{\overbracket{R}} \vee \overset{res}{\overbracket{R'}}$
        if $R'_{ov} = ?$ then $\theta'' \leftarrow \theta'$ else $\theta'' \leftarrow R'_{ov}$
        $\Psi \leftarrow \Psi \cup \{(1, i) \mapsto \theta''\}$
        $linkParams \leftarrow linkParams \cup linkParams'$
    end
    $R_{ov} \leftarrow$ summary($\overset{res}{\overbracket{R}}$)
    return $\langle \overset{res}{\overbracket{R}}, R_{ov}, \Psi, linkParams \rangle$
end
```

version θ'_i, which is produced in the nested recursion of `evaluateFormula`, or by its result if that is definite (i.e. $\top$ or $\bot$). As discussed in Section 5.5, this means that if the encompassing formula is added to the schedule, these definite parts do not have to be evaluated again in future steps, which possibly leads to a significant optimization. The leading index 1 in the position path, which is also found above in the call to `evaluateFormula`, is necessary because the subformulas $\theta_1, \ldots, \theta_n$ are actually contained in a list which represents a nested term by itself. After all subformulas have been evaluated, a summary R_{ov} of the combined result mapping $\overset{res}{\overbracket{R}}$ is calculated as defined in Definition 5.17. Finally, the result mapping, its summary, the substitution set, and the combined *linkParams* list, are returned to the calling instance of `evaluateFormula`, where the updated formula Ψ' is either produced by applying the substitution Ψ or set to the result mapping summary if it is definite.

5.6.6 Relational Fluents and Predicates

The most basic element in which the recursion of the formula evaluation algorithm inevitably ends is either a comparison, a predicate, or a relational fluent. At this point, the algorithm relies on Prolog to perform the evaluation, i.e. it either uses a built-in comparison operator or calls a Prolog goal that realizes the predicate or the successor state axiom of the fluent. This is shown in Algorithm 5.7. Here, comparison operators that are normally used in infix notation are treated as predicates in prefix notation, e.g. $=(\theta_1, \theta_2)$. The result itself is determined directly by evaluating Φ as a Prolog expression. However, this only works if all subterms $\theta_1, \ldots, \theta_n$ are ground, i.e. they are either literals or Prolog variables that have been bound to values during the preceding evaluation. In fact, the compilation process described in Section 5.4 ensures that this is the case by collecting the subterms in a conjunction, ordered from the innermost to the outermost. Since the subterms of a conjunction are evaluated "from left to right", all variables are bound before the predicate or fluent is evaluated.

Algorithm 5.7: Evaluation of relational fluents and predicates.

```
...
case $\Phi = p(\theta_1, \ldots, \theta_n) \vee \Phi = f(\theta_1, \ldots, \theta_n, S)$ where $p$ is a
          situation independent predicate or a comparison operator,
          $f$ is a boolean fluent, and $S$ is a situation term.
    assume $\forall 1 \leq i \leq n.\ \theta_i$ is a ground term.
    call $\Phi$
    $R_{ov} \leftarrow \begin{cases} \top & \text{if calling } \Phi \text{ as a Prolog goal evaluates to true} \\ \bot & \text{otherwise} \end{cases}$
    $\overset{res}{\overline{R}} \leftarrow \overline{T_s}|_{R_{ov}}$
    $linkParams \leftarrow \emptyset$
end
...
```

5.6.7 Functions and Functional Fluents

When discussing the evaluation of relational fluents and predicates, it was assumed that variables occurring in the predicate or relational fluent expression are bound beforehand. This binding actually occurs directly at the evaluation of functional fluents or situation-independent functions. As before, operators like $+, -, *$ are understood as functions, i.e. $+(\theta_1, \theta_2)$ etc. Analogically to the case with relational fluents or predicates, functions are directly evaluated as Prolog expressions. The difference is that a function or functional fluent expects one uninstantiated variable as its last argument (before the situation term), which will contain the result after evaluation of the function. Like before, it can be seen that the property compiler ensures that all arguments except the result variable are literals or Prolog variables that have been instantiated in the preceding evaluation. After Φ has been evaluated as a Prolog expression, the result variable will be instantiated. Of course, this is only guaranteed if the Prolog goal in f or g indeed binds a value to θ_n. For both Prolog's built-in operators and functional fluents that have been specified by means of effect axioms (see Section 3.2.4), this is automatically true. For all other user-defined functions, the modeler is responsible for establishing the correct structure.

Algorithm 5.8: Evaluation of functions and functional fluents.

```
...
case $\Phi = f(\theta_1, \ldots, \theta_n, S) \vee \Phi = g(\theta_1, \ldots, \theta_n)$ where $f$ is a
        non-boolean fluent, $g$ is a situation independent
        function or operator, and $S$ is a situation term.
    assume $\theta_1, \ldots, \theta_{n-1}$ are literals or instantiated Prolog variables
    assume $\theta_n$ is uninstantiated Prolog variable
    call $\Phi$                          /* binds all variables in $\Phi$ */

    assume $\theta_n$ is instantiated

    $\overset{res}{\overline{R}} \leftarrow \overline{T_s}|_\top$
    $R_{ov} \leftarrow \top$, linkParams $\leftarrow \emptyset$
end
...
```

5.6.8 Action Occurrences

Besides regular functions and predicates that are evaluated as described above, there is also the special function **lastTime** and the predicate **occur** that refer to the last occurrence of an event or action. Both are realized through the *action clock store* that is integrated in SALMA's progression mechanism. In fact, the first time a concrete action or event instance is executed in a progression step, a time-stamp is created by adding a *dynamic clause* to the Prolog database that stores the current time. At each subsequent occurrence of the same action or event instance, this dynamic clause is replaced by an updated version with the latest occurrence time. The special function *lastTime* provides direct access to this time-stamp by instantiating the Prolog variable passed as the last argument in the same ways as during the evaluation of regular functions. On the other hand, *occur* simply compares the time-stamp to the current time in order to test whether the action instance was part of the last progression step.

Algorithm 5.9: Evaluation of action occurrence tests.

```
...
case $\Phi = occur(\alpha)$
    $T_{cur} \leftarrow time(do(tick(s_{cur}), S_0))$
    $T_\alpha \leftarrow actionClock(\alpha)$
    $R_{ov} \leftarrow \begin{cases} \top & \text{if } T_{cur} = T_\alpha \\ \bot & \text{otherwise} \end{cases}$
    $\overset{res}{\overbracket{R}} \leftarrow \overbracket{T_s}|_{R_{ov}}$
    $linkParams \leftarrow \emptyset$
end
case $\Phi = lastTime(\alpha, v)$
    assume $v$ is an uninstantiated Prolog variable.)
    $v \leftarrow actionClock(\alpha)$
    $\overset{res}{\overbracket{R}} \leftarrow \overbracket{T_s}|_{\top}$
    $linkParams \leftarrow \emptyset$
end
...
```

5.6.9 Variable Assignments

When expressions get more complex, one common way to enhance readability is using variables to store the values of sub-expressions for later reuse. Inspired by functional programming languages, the SALMA-PSL offers the *let*-construct that binds the current value of an expression to a variable that is valid within a nested subformula.

Algorithm 5.10 shows how such a formula is evaluated in SALMA. Here, the expression $let(x : \hat{x}, \Theta', \Phi')$ is the compilation result from a formula of the form $let(x : \Theta, \Phi)$ (see Definition 5.22), where $\hat{x}$ is a fresh Prolog variable, Θ' is the compilation result of the expression $\hat{x} = \Theta$, and Φ' is the compiled version of Φ. Since $=$ is treated like a normal predicate by the SALMA-PSL compiler and interpreter, Θ' is actually a $c\_$-term term that can be evaluated directly with `evaluateFormula`. In the recursion, this will eventually lead to the evaluation of the predicate $=$, which is interpreted directly as a Prolog goal (see Algorithm 5.7). Due to the standard Prolog semantics, this will assign the current value of Θ to $\hat{x}$. After that, $\hat{x}$ will contain a ground term that is then substituted for all occurrences of the variable symbol x in Φ'. The resulting expression is then evaluated as a regular formula with a second call to `evaluateFormula`. As described in Section 5.6.4, this returns a rewritten

Algorithm 5.10: Handling *let*-expressions.

```
...
case $\Phi = let(x : \hat{x}, \Theta', \phi)$
    assume $\hat{x}$ is a fresh (uninstantiated) Prolog variable.
    assume $\Theta' = c\_([\theta_1, \ldots, \theta_n, = (\hat{x}, v)])$ where $v$ is a Prolog variable
    that is instantiated during evaluation of $\theta_1, \ldots, \theta_n$.
    evaluateFormula($\Theta'$, $\hat{p}_\Phi$, $level$,
                            $s_{cur}$, $\overline{T_s}$, $T_{end}$)
    $\phi' \leftarrow subst^b_{term}(\{x \mapsto \hat{x}\}, \phi)$
    $\langle \overset{res}{\overline{R}}, R_{ov}, \Phi'', linkParams \rangle \leftarrow$ evaluateFormula($\phi'$, $\hat{p}_\Phi$, $level$,
                            $s_{cur}$, $\overline{T_s}$, $T_{end}$)
    $\Psi \leftarrow \{(0) \mapsto \Phi''\}$
end
...
```

version of the evaluated formula, which is stored in the variable Φ''. In the end, Φ'' is set as a replacement for the entire original formula (selected by the position path (0)) in the final substitution Ψ. Back in `evaluateFormula`, Ψ is used to create a rewritten formula version that is stored in the evaluation goal schedule when an entry needs to be created. Effectively, this means that the *let*-expression is eliminated in the schedule entry and the value for the variable x is fixed as the value of Θ at the current situation.

5.6.10 Evaluation of Temporal Operators

As one would expect, the major part of the evaluation algorithm's complexity is found within the cases for temporal operators. In fact, it is the ability to use interval operations for realizing the semantics of the temporal operators that makes SALMA's evaluation algorithm efficient. Therefore, this subsection discusses the involved strategies and design decisions in detail. Algorithm 5.11 gives an overview of the three cases for temporal operators, namely *eventually*, *always*, and *until*. It can be seen that *eventually* and *always* are handled by the same function that distinguishes the cases via its first parameter. This implies that the evaluation of both operators is similar to some extent, which is indeed confirmed below. On the other hand, there are several special difficulties for the evaluation of the *until* operator, so this case is handled separately in order to achieve a comprehensible presentation.

Algorithm 5.11: Evaluation of temporal operators.

```
...
case $\Phi = always(T_{max}, P)$
    $\langle \overset{res}{\overline{R}}, R_{ov}, \Psi, linkParams \rangle \leftarrow$ evaluateAlwaysOrEventually($\square$, $P$,
                    $\hat{p}_\Phi$, $level$, $s_{cur}$, $\overline{T_s}$, $T_{end}$, $T_{max}$)
end
case $\Phi = eventually(T_{max}, P)$
    $\langle \overset{res}{\overline{R}}, R_{ov}, \Psi, linkParams \rangle \leftarrow$ evaluateAlwaysOrEventually($\diamondsuit$, $P$,
                    $\hat{p}_\Phi$, $level$, $s_{cur}$, $\overline{T_s}$, $T_{end}$, $T_{max}$)
end
case $\Phi = until(T_{max}, P, Q)$
    $\langle \overset{res}{\overline{R}}, R_{ov}, \Psi, linkParams \rangle \leftarrow$ evaluateUntil($P$, $Q$,
                    $level$, $\hat{p}_\Phi$, $s_{cur}$, $\overline{T_s}$, $T_{end}$, $T_{max}$)
end
```

A Common procedure for $always(T_{max}, P)$ and $eventually(T_{max}, P)$

The common scaffolding for the evaluation of *always* and *eventually* is the function shown in Algorithm 5.12. First, the upper time bound for the evaluation interval $T^{>}$ is calculated as the minimum of the end of the time advance T_{end} and the *evaluation limit* T_{lim}. The latter is given by the latest point that is reachable from any point in the start time interval sequence $\overline{T_s}$ within the time bound of the operator T_{max}. To calculate this, the operator max_t from Definition 5.11 is used, which yields the latest time point in the given interval sequence. The other relevant time point, T_{cur}, is determined as before by evaluating the fluent *time* for the situation given by the current step s_{cur}. After these initializations have been made, the algorithm has to distinguish between two cases. First, the subformula P of the temporal operator could be of the form $sched(Id_{sched}, Id_{cache})$, which implies that in fact the original subformula at this position has been scheduled as a subgoal because at least for one time instance the result of the evaluation was not definite. In this case, the formula is loaded from the cache using the id that was conveyed in the *sched*-term. Otherwise, if P is no *sched*-term, it is used directly for the further evaluation and the ids for the formula cache and the schedule are set to -1, which act as commands that will cause the creation of new entries as required.

The next step is to evaluate the subformula for all distinct time steps from the current time T_{cur} to the interval end $T^{>}$. As expected, this is performed by the function `evaluateForAllTImesteps` from Algorithm 5.2 that was already presented as part of the top-level property evaluation procedure (Algorithm 5.1). It was shown before that during this iteration, schedule entries are created for each time point for which the evaluation did not produce a definite result (?). Therefore, it can be assumed after the call to `evaluateForAllTImesteps` that the evaluation goal schedule is set up with all entries that are necessary for further calculations. The variables Id'_{sched} and Id'_{cache} contain either -1 or the ids that refer to the relevant entries in the schedule and the cache for P *after* the lookahead evaluation, which could have been created in this process. Besides the updated ids, the call to `evaluateFor-AllTImesteps` returns three other values, namely the result summary R_{ov1}, the time of the earliest positive definite outcome, $T^{<}_{def}$, and the time of the latest possibility for a positive outcome, $T^{>}_{poss}$. While these time markers were ignored in the top-level evaluation procedure (Algorithm 5.1), they are crucial for the evaluation of temporal expressions.

At this point, the treatment of *always* and *eventually* diverges into the functions `handleAlways` and `handleEventually` that are shown in Algorithms 5.13 and 5.15 and are discussed below. Both functions return a result mapping $\overset{res}{\overline{R_1}}$ and an interval sequence $\overline{T'_s}$ that contains the start times from $\overline{T_s}$ for which no definite result could be determined. Consequently, the intervals in $\overline{T'_s}$ are labeled with ? and combined with the results from before into the final result mapping $\overset{res}{\overline{R}}$. In the rest of Algorithm 5.12, a result summary is determined and, if the result is not entirely definite, a *sched*-term is established as replacement for P that, together with the link parameters that are created along the way, reference the goal that represents P in the evaluation goal schedule.

In the following, the individual algorithms for the evaluation of *always* and *eventually* are described. Although they are structured very similarly, the full functions are shown to make them easier to understand.

The Evaluation of $always(T_{max}, P)$

Algorithm 5.13 shows the function `handleAlways`, which expects as arguments the previously calculated result summary of the lookahead evaluation R_{ov1}, the time of the latest possible positive result $T^{>}_{poss}$, the time bound of the *always*-expression T_{max}, the evaluation interval end $T^{>}$, the start time interval sequence $\overline{T_s}$, and the schedule id of the goal that represents the subformula P, which could also be -1 as described above.

The interval-based calculations described above are visualized in Figure 5.15. In the top row (Figure 5.15a), the first part of `handleAlways` can be seen, where the algorithm tries to leverage results from the lookahead evaluation that was done beforehand. This is possible when a negative result was found at some point in `evaluateForAllTimesteps`. Then, $T^{>}_{poss}$ will point to the last time at which a positive result is still possible, which has to be the step just before the one in which the negative result was found. This implies that all scheduled instances of P that start not longer than T_{max} before $T^{>}_{poss}$ will be terminated by the negative result. Therefore, these time points can be marked as negative,

Algorithm 5.12: Evaluation of *always* or *eventually*.

```
function evaluateAlwaysOrEventually($mode$, $P$, $\hat{p}_\Phi$, $level$, $s_{cur}$,
                                                        $\overline{T_s}$, $T_{end}$, $T_{max}$)
    $T^{>}_{start} \leftarrow max_t(\overline{T_s})$
    $T_{lim} \leftarrow T^{>}_{start} + T_{max}$
    $T^{>} \leftarrow min(T_{lim}, T_{end})$
    $T_{cur} \leftarrow time(do(tick(s_{cur}), S_0))$
    if $P = sched(Id_{sched}, Id_{cache})$ then
        $P' \leftarrow$ loadFromCache($Id_{cache}$)
    else
        $P' \leftarrow P, Id_{sched} \leftarrow -1, Id_{cache} \leftarrow -1$
    end
    $\langle R_{ov1}, T^{<}_{def}, T^{>}_{poss}, Id'_{sched}, Id'_{cache}\rangle \leftarrow$
              evaluateForAllTimesteps($mode$, $P'$, $\hat{p}_\Phi \circ 2$, $level + 1$,
                                      $s_{cur}$, $T_{cur}$, $T^{>}$, $Id_{sched}$, $Id_{cache}$)
    assume A schedule entry has been added for every time point
             where the evaluation of $P$ yielded ?.
    if $mode = \Box$ then
        $\langle \overset{res}{\overline{R_1}}, \overline{T'_s}\rangle \leftarrow$ handleAlways($R_{ov1}$, $T^{>}_{poss}$,
                                              $T_{max}$, $T^{>}$, $\overline{T_s}$, $Id'_{sched}$)
    else
        $\langle \overset{res}{\overline{R_1}}, \overline{T'_s}\rangle \leftarrow$ handleEventually($R_{ov1}$, $T^{<}_{def}$,
                                              $T_{max}$, $T^{>}$, $\overline{T_s}$, $Id'_{sched}$)
    end
    $\overset{res}{\overline{R_2}} \leftarrow \overline{T'_s}|_?$
    $\overset{res}{\overline{R}} \leftarrow \overset{res}{\overline{R_1}} \cup \overset{res}{\overline{R_2}}$
    $R_{ov} \leftarrow \mathsf{summary}(\overset{res}{\overline{R}})$
    if $R_{ov} = ?$ then
        $\Psi \leftarrow \{2 \mapsto sched(*, Id'_{cache})\}$         /* * represents
                  a placeholder for the concrete schedule id */
        $linkParams \leftarrow \{\hat{p}_\Phi \circ 2 \mapsto Id'_{sched}\}$
    else
        $\Psi \leftarrow \emptyset$, $linkParams \leftarrow \emptyset$
    end
    return $\langle \overset{res}{\overline{R}}, R_{ov}, \Psi, linkParams\rangle$
end
```

which is done by building the intersection of $\overline{T_s}$ with the described interval and labeling it with the unique assignment operator $|$, which creates a result mapping that is stored in the variable $\overset{res}{\overline{R_1}}$. The relative complement of the original start times $\overline{T_s}$ and the said interval yields an interval sequence that contains the time start time instances for which the result is still open. However, it is still possible that more definite results can be found. First, if there is an entry for P in the evaluation goal schedule, the evaluation history could convey enough information to determine additional results. This is pursued in the function *checkScheduleAlways* in Algorithm 5.14.

In Algorithm 5.14, the goal evaluation history is loaded from the schedule using the given id. By matching all intervals within the negative history $G.\overline{T_\bot}$ against the start times $\overline{T_s}$, it is possible to mark all those instances as failed that are at most T_{max} time units before a time point with negative result. Similar to the first step in `handleAlways`, this labeling of multiple time instances is reduced to a single intersection between an interval sequence and an interval. The result mapping that is created in this way is appended to a collective result mapping $\overset{res}{\overline{R_{tot}}}$ that is returned later, and as before, the sequence of undetermined start time intervals is updated using the relative complement. Figure 5.15c shows an example for the negative labeling of several interval segments based on one interval from $\mathtt{G}.\overline{T_\bot}$.

The other source for determining definite results is the positive history $G.\overline{T_\top}$. Indeed, this sequence might contain positive intervals that are at least as long as T_{max}. Then, any goal instance that starts within that interval and at least T_{max} time units before its end, can be labeled as positive because the invariant is guaranteed to have been fulfilled within the given time bound. This can be seen in Figure 5.15d. All results that are calculated based on the evaluation history are returned back to `handleAlways`, together with the start time intervals for no results have been found, yet, which will later be marked with ?, as shown in Algorithm 5.12.

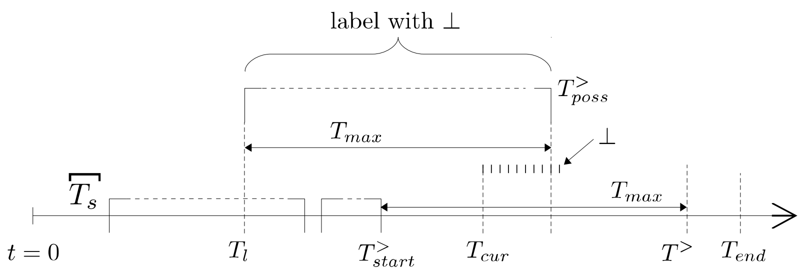
Back in `handleAlways`, it is also possible that no schedule entry exists. However, some definite results can still be found if the overall result of the lookahead evaluation was positive. This means that there must have been positive results for all time points up to $T^>$. Hence, all instances in $\overline{T_s}$ that begin at least T_{max} time units before $T^>$ must be satisfied and can be labeled with $\top$. Figure 5.15b shows the involved interval operations. It can be seen that the algorithm effectively assumes that the evaluation results for the subformula P have also been positive for all relevant time points that are overlapped by intervals in $\overline{T_s}$. Indeed, this assumption is justified because if any of these time points had yielded a negative result, then the intervals in $\overline{T_s}$ that are within a distance of T_{max} would have been negated already in a previous evaluation step.

Algorithm 5.13: Handling of temporal operator *always*.

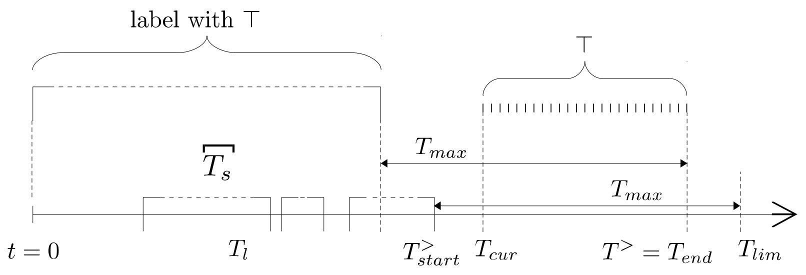
```
function handleAlways($R_{ov1}$, $T^{>}_{poss}$, $T_{max}$, $T^{>}$, $\overline{T_s}$, $Id_{sched}$)
    if $R_{ov1} = \bot$ then
                                                      // See Figure 5.15a
        $T_l \leftarrow T^{>}_{poss} - T_{max}$
        $\overset{res}{\overline{R_1}} \leftarrow (\overline{T_s} \cap [T_l, T^{>}])|_{\bot}$
        $\overline{T'_s} \leftarrow \overline{T_s} \setminus [T_l, T^{>}]$
    else
        $\overset{res}{\overline{R_1}} \leftarrow \emptyset, \overline{T'_s} \leftarrow \overline{T_s}$
    end
    if $\overline{T'_s} \neq \emptyset$ then
        if $Id_{sched} > -1$ then
            $\langle \overset{res}{\overline{R_2}}, \overline{T''_s} \rangle \leftarrow$ checkScheduleAlways($Id_{sched}$, $\overline{T'_s}$, $T_{max}$)
        else
            // There is no schedule entry for the subformula $P$.
            if $R_{ov1} = \top$ then
                                                      // See Figure 5.15b
                $T_r \leftarrow T^{>} - T_{max}$
                $\overset{res}{\overline{R_2}} \leftarrow (\overline{T'_s} \cap [0, T_r])|_{\top}$
                $\overline{T''_s} \leftarrow \overline{T'_s} \setminus [0, T_r]$
            else
                $\overset{res}{\overline{R_2}} \leftarrow \emptyset, \overline{T''_s} \leftarrow \overline{T'_s}$
            end
        end
    else
        $\overset{res}{\overline{R_2}} \leftarrow \emptyset, \overline{T''_s} \leftarrow \emptyset$
    end
    return $\langle \overset{res}{\overline{R_1}} \cup \overset{res}{\overline{R_2}}, \overline{T''_s} \rangle$
end
```

Algorithm 5.14: Calculating schedule decisions for *always*.

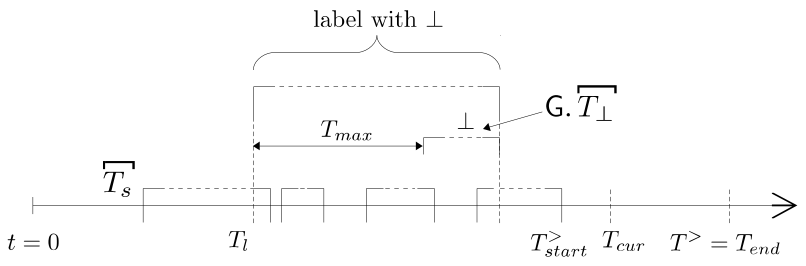
```
function checkScheduleAlways($Id_{sched}$, $\overline{T_s}$, $T_{max}$)
    $G \leftarrow$ loadFromSchedule($Id_{sched}$)
    $\overline{T'_s} \leftarrow \overline{T_s}$
    $\overset{res}{\overline{R_{tot}}} \leftarrow \emptyset$
    /* Negate all starting points within range of $\perp$
       instances.                                                     */
    foreach $[T_1, T_2] \in G.\overline{T_\perp}$ do
                                                  // See Figure 5.15c
        $T_l \leftarrow \max(0, T_1 - T_{max})$
        $\overset{res}{\overline{R}} \leftarrow (\overline{T'_s} \cap [T_l, T_2])|_\perp$
        $\overset{res}{\overline{R_{tot}}} \leftarrow \overset{res}{\overline{R_{tot}}} \cup \overset{res}{\overline{R}}$
        $\overline{T'_s} \leftarrow \overline{T'_s} \setminus [T_l, T_2]$
    end
    /* Confirm goals that are entirely covered in $\top$
       intervals.                                                     */
    foreach $[T_1, T_2] \in G.\overline{T_\top}$ do
                                                  // See Figure 5.15d
        if $T_2 - T_1 \geq T_{max}$ then
            $T_r \leftarrow T_2 - T_{max}$
            $\overset{res}{\overline{R}} \leftarrow (\overline{T'_s} \cap [T_1, T_r])|_\top$
            $\overset{res}{\overline{R_{tot}}} \leftarrow \overset{res}{\overline{R_{tot}}} \cup \overset{res}{\overline{R}}$
            $\overline{T'_s} \leftarrow \overline{T'_s} \setminus [T_1, T_r]$
        end
    end
    return $\langle \overset{res}{\overline{R_{tot}}}, \overline{T'_s} \rangle$
end
```



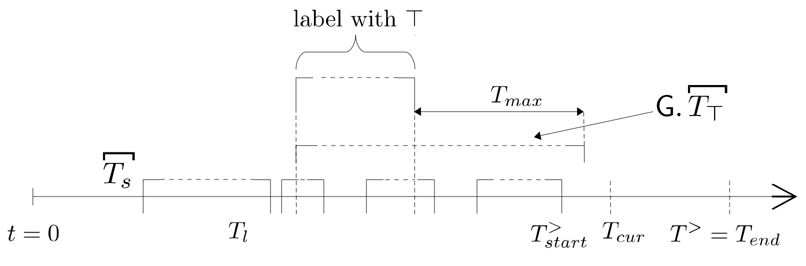
(a) Determining $\perp$ instances through lookahead evaluation results.



(b) Determining $\top$ instances through lookahead evaluation results.



(c) Determining $\perp$ instances through schedule history.



(d) Determining $\top$ instances through schedule history.

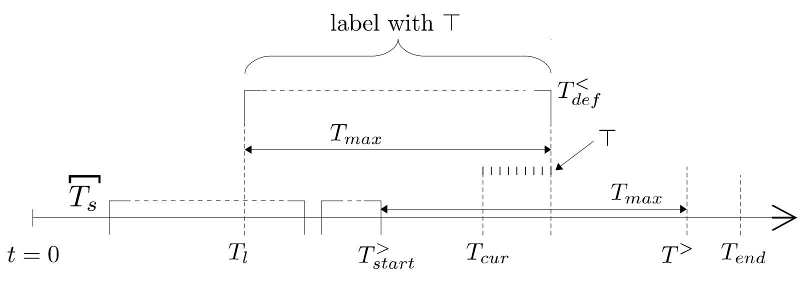
Figure 5.15: Calculation of results for *always*.

The Evaluation of $eventually(T_{max}, P)$

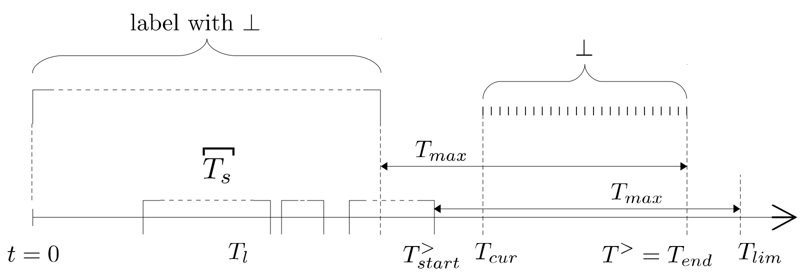
As expected, the operator *eventually* is treated very similarly to its *always* counterpart. In fact, the only difference is that the roles of positive results are reversed. This means that exactly the same structure that is used to “reject” goal instances based on negative lookahead results in the case of *always* is used to confirm goal instances with positive results. The same congruence obviously holds also for the other cases. Therefore, it is not very instructional to discuss the algorithms for *eventually* in detail. Nonetheless, both the algorithms (Algorithm 5.15, 5.16) and the interval diagrams (Figure 5.16) are included here for the sake of completeness.

Algorithm 5.15: Handling of temporal operator *eventually*.

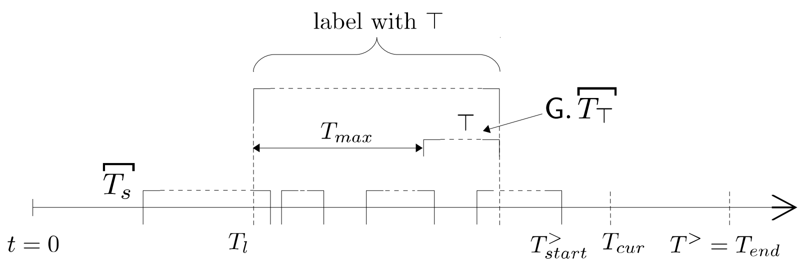
```
function handleEventually($R_{ov1}$, $T^{<}_{def}$, $T_{max}$, $T^{>}$, $\overbracket{T_s}$, $Id_{sched}$)
    if $R_{ov1} = \top$ then
        $T_l \leftarrow T^{<}_{def} - T_{max}$
        $\overset{res}{\overbracket{R_1}} \leftarrow (\overbracket{T_s} \cap [T_l, T^{>}])|_{\top}$
        $\overbracket{T'_s} \leftarrow \overbracket{T_s} \setminus [T_l, T^{>}]$
    else
        $\overset{res}{\overbracket{R_1}} \leftarrow \emptyset$, $\overbracket{T'_s} \leftarrow \overbracket{T_s}$
    end
    if $\overbracket{T'_s} \neq \emptyset$ then
        if $Id_{sched} > -1$ then
            $\langle \overset{res}{\overbracket{R_2}}, \overbracket{T''_s} \rangle \leftarrow$ checkScheduleEventually($Id_{sched}$, $\overbracket{T'_s}$, $T_{max}$)
        else
            // There is no schedule entry for $P$.
            if $R_{ov1} = \bot$ then
                $T_r \leftarrow T^{>} - T_{max}$
                $\overset{res}{\overbracket{R_2}} \leftarrow (\overbracket{T'_s} \cap [0, T_r])|_{\bot}$
                $\overbracket{T''_s} \leftarrow \overbracket{T'_s} \setminus [0, T_r]$
            else
                $\overset{res}{\overbracket{R_2}} \leftarrow \emptyset$, $\overbracket{T''_s} \leftarrow \overbracket{T'_s}$
            end
        end
    else
        $\overset{res}{\overbracket{R_2}} \leftarrow \emptyset$, $\overbracket{T''_s} \leftarrow \emptyset$
    end
    return $\langle \overset{res}{\overbracket{R_1}} \cup \overset{res}{\overbracket{R_2}}, \overbracket{T''_s} \rangle$
end
```



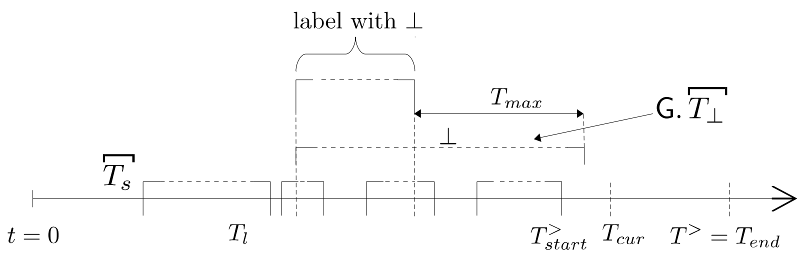
(a) Determining $\top$ instances through lookahead evaluation results.



(b) Determining $\perp$ instances through lookahead evaluation results.



(c) Determining $\top$ instances through schedule history.



(d) Determining $\perp$ instances through schedule history.

Figure 5.16: Calculation of results for *eventually*.

Algorithm 5.16: Calculating schedule decisions for *eventually*.

```
function checkScheduleEventually($Id_{sched}$, $\overline{T_s}$, $T_{max}$)
    $G \leftarrow \mathsf{loadFromSchedule}(Id_{sched})$
    $\overline{T'_s} \leftarrow \overline{T_s}$
    $\overset{res}{\overline{R_{tot}}} \leftarrow \emptyset$
    /* Acknowledge all starting points within range of $\top$
       instances.                                                    */
    foreach $[T_1, T_2] \in G.\overline{T_\top}$ do
        $T_l \leftarrow \max(0, T_1 - T_{max})$
        $\overset{res}{\overline{R}} \leftarrow (\overline{T'_s} \cap [T_l, T_2])|_\top$
        $\overset{res}{\overline{R_{tot}}} \leftarrow \overset{res}{\overline{R_{tot}}} \cup \overset{res}{\overline{R}}$
        $\overline{T'_s} \leftarrow \overline{T'_s} \setminus [T_l, T_2]$
    end
    /* Handle time-outs.                                             */
    foreach $[T_1, T_2] \in G.\overline{T_\bot}$ do
        if $T_2 - T_1 \geq T_{max}$ then
            $T_r \leftarrow T_2 - T_{max}$
            $\overset{res}{\overline{R}} \leftarrow (\overline{T'_s} \cap [T_1, T_r])|_\bot$
            $\overset{res}{\overline{R_{tot}}} \leftarrow \overset{res}{\overline{R_{tot}}} \cup \overset{res}{\overline{R}}$
            $\overline{T'_s} \leftarrow \overline{T'_s} \setminus [T_1, T_r]$
        end
    end
    return $\langle \overset{res}{\overline{R_{tot}}}, \overline{T'_s} \rangle$
end
```

Evaluation of *until*(T_{max}, P, Q)

For the efficient evaluation of *until*-expressions, the algorithm has to extend and combine the ideas from the handlers for *eventually* / *always* so that definite results can be found as early as possible. The main procedure for *until* is shown in Algorithm 5.17. The first part performs an initialization that is very similar to that of the Algorithms 5.13 and 5.15. Of course, two subformulas have to be set up, namely P for the invariant part and Q for the goal part. After the initialization, lookahead evaluation is performed for both P and Q, with Q treated like an *eventually* formula ($\Diamond$) and P as in *always* ($\Box$). It is worth noticing that for Q, the iteration is not necessarily performed up to the general end of the evaluation period ($T^>$), but only to the earliest point where

Algorithm 5.17: Evaluation of *until*.

```
function evaluateUntil($P$, $Q$, $\hat{p}_\Phi$, $level$, $s_{cur}$, $\overbracket{T_s}$, $T_{end}$, $T_{max}$)
    $T^{>}_{start} \leftarrow max_t(\overbracket{T_s})$
    $T_{lim} \leftarrow T^{>}_{start} + T_{max}$
    $T^{>} \leftarrow min(T_{lim}, T_{end})$
    $T_{cur} \leftarrow time(do(tick(s_{cur}), S_0))$
    if $P = sched(Id_{sched,P}, Id_{cache,P})$ then
        $P' \leftarrow$ loadFromCache($Id_{cache,P}$)
    else
        $P' \leftarrow P$
        $Id_{sched,P} \leftarrow$ new, $Id_{cache,P} \leftarrow$ new          // force scheduling
    end
    if $Q = sched(Id_{sched,Q}, Id_{cache,Q})$ then
        $Q' \leftarrow$ loadFromCache($Id_{cache,Q}$)
    else
        $Q' \leftarrow Q, Id_{sched,Q} \leftarrow$ new, $Id_{cache,Q} \leftarrow$ new
    end
    $\langle T^{<}_{def,Q}, Id'_{sched,Q}, Id'_{cache,Q}\rangle \leftarrow$ evaluateForAllTimesteps($\Diamond$,
            $Q'$, $\hat{p}_\Phi \circ 3$, $level + 1$, $s_{cur}$, $T_{cur}$, $T^{>}$, $Id_{sched,Q}$, $Id_{cache,Q}$)
    if $T^{<}_{def,Q} \neq ?$ then $T^{>}_{P} \leftarrow T^{<}_{def,Q}$ else $T^{>}_{P} \leftarrow ?$
    $\langle Id'_{sched,P}, Id'_{cache,P}\rangle \leftarrow$ evaluateForAllTimesteps($\Box$, $P'$, $\hat{p}_\Phi \circ 2$,
            $level + 1$, $s_{cur}$, $T_{cur}$, $T^{>}_{P}$, $Id_{sched,P}$, $Id_{cache,P}$)
    assume all relevant results between $T_{cur}$ and $T^{>}$
            have been added to the goal schedule.
    $\langle \overset{res}{\overbracket{R}}, \overbracket{T'}\rangle \leftarrow$ checkScheduleUntil($Id'_{sched,P}$, $Id'_{sched,Q}$, $\overbracket{T_s}$, $T_{max}$)
    $\overset{res}{\overbracket{R}} \leftarrow \overset{res}{\overbracket{R}} \cup \overbracket{T'}|_{?}$
    $R_{ov} \leftarrow$ summary($\overset{res}{\overbracket{R}}$)
    if $R_{ov} = ?$ then
        $\Psi \leftarrow \{2 \mapsto sched(*, Id'_{cache,P}), 3 \mapsto sched(*, Id'_{cache,Q})\}$
        $linkParams \leftarrow \{\hat{p}_\Phi \circ 2 \mapsto Id'_{sched,P}, \hat{p}_\Phi \circ 3 \mapsto Id'_{sched,Q}\}$
    else
        $\Psi \leftarrow \emptyset$, $linkParams \leftarrow \emptyset$
    end
    return $\langle \overset{res}{\overbracket{R}}, R_{ov}, \Psi, linkParams\rangle$
end
```

a positive definite result for Q is found – if that is the case. Another specialty of the *until*-case is that the creation of evaluation goal schedule entries is forces by the *new* flag, even in cases where the lookahead evaluation produced only definite results. This guarantees that a history for both P and Q exists that contains all results up to $T^{>}$. The evaluation algorithm can therefore use the same decision mechanisms over the whole relevant time span, which allows for a much clearer design. In that sense, the function `checkScheduleUntil` in Algorithm 5.18 arranges all schedule-based calculations into four phases that are by themselves sourced out to four separate functions.

In the first phase, the function `confirmScheduledUntilGoals` (Algorithm 5.19) uses the evaluation goal history of both P and Q to determine instances that can be confirmed definitively. First, this is possible when the goal Q is true directly at the start time of a goal instance, i.e. when the intersection between an interval in $\mathsf{G_Q}.\overbracket{T_\top}$ and $\overbracket{T_s}$ is not empty. Figure 5.17a demonstrates this situation. Therefore, the function in Algorithm 5.19 iterates through all intervals in the positive history of Q and successively fills a result mapping with $\top$-entries that are retrieved by means of the intersection operator from Definition 5.8. Besides that, the start time points that are confirmed in this way are "cut out" of $\overbracket{T_s}$ by the relative complement and the iteration is canceled at the end of the loop body if $\overbracket{T_s}$ is empty.

As shown at in the right part of Figure 5.17a, even if Q is not true at an instance's start, a positive outcome can still be confirmed when a true result for Q is found within T_{max} time units and P is true up to that point. The second part of this condition is equivalent to the existence of an interval in $\mathsf{G_P}.\overbracket{T_\top}$ that is left adjacent to a positive Q interval. If such an interval exists, then all time points can be confirmed that are both within that interval and

Algorithm 5.18: Calculating schedule decisions for *until*.

```
function checkScheduleUntil($Id_{sched,P}$, $Id_{sched,Q}$, $\overbracket{T_s}$, $T_{max}$)
    $G_P \leftarrow \mathsf{loadFromSchedule}(Id_{sched,P})$
    $G_Q \leftarrow \mathsf{loadFromSchedule}(Id_{sched,Q})$
    $\overbracket{T'} \leftarrow \overbracket{T_s}$                    /* unhandled start times */
    $\langle \overset{res}{\overbracket{R_1}}, \overbracket{T'} \rangle \leftarrow$ confirmScheduledUntilGoals($\overbracket{T'}$, $G_P$, $G_Q$, $T_{max}$)
    $\langle \overset{res}{\overbracket{R_2}}, \overbracket{T'} \rangle \leftarrow$ rejectUntilGoalsP($\overbracket{T'}$, $G_P$)
    $\langle \overset{res}{\overbracket{R_3}}, \overbracket{T'} \rangle \leftarrow$ rejectUntilGoalsPQ($\overbracket{T'}$, $G_P$, $G_Q$)
    $\langle \overset{res}{\overbracket{R_4}}, \overbracket{T'} \rangle \leftarrow$ detectTimedOutUntilGoals($\overbracket{T'}$, $G_Q$, $T_{max}$)
    return $\langle \overset{res}{\overbracket{R_1}} \cup \overset{res}{\overbracket{R_2}} \cup \overset{res}{\overbracket{R_3}} \cup \overset{res}{\overbracket{R_4}}, \overbracket{T'} \rangle$
end
```

not farther than T_{max} from the positive Q interval. This is done in the middle block of Algorithm 5.19, again using the intersection operator and the relative complement to compile results and to remove handled start times from $\overbracket{T_s}$.

After all positive results have been gathered, the produced result mapping is returned to `checkScheduleUntil` together with a temporal interval sequence $\overbracket{T'}$ that contains the "unhandled" start times, i.e. the goal instances that have not been confirmed yet. In the next step, this remaining start time interval sequence is passed to the function `rejectUntilGoalsP` (Algorithm 5.20) that processes cases in which goal instances can be rejected because P is known to be violated at their start point (see right part of Figure 5.17b). On the other hand, goal instances can also be identified as failed when the period between their start and the earliest time point where Q is true is interrupted by a negative outcome for P. Transferred to the interval-based viewpoint, this situation can be identified when an interval of $\mathsf{G_Q}.\overbracket{T_\perp}$ overlaps intervals of $\mathsf{G_P}.\overbracket{T_\perp}$. Then, all goal instances that start within the interval with negative Q and before the last time point in the overlapped region of $\mathsf{G_P}.\overbracket{T_\perp}$ are inevitably interrupted and can thus be marked as failed (see Algorithm 5.21 and left part of Figure 5.17b). Finally, in the last step of `checkScheduleUntil`, the algorithm searches for goal instances that have expired their time bounds (see Algorithm 5.22 and Figure 5.17c). After all these possible definite cases have been handled, the result for `evaluateUntil` is combined very similarly to the cases for *eventually* and *always*.

Algorithm 5.19: Confirmation of *until* goals.

```
function confirmScheduledUntilGoals($\overline{T_s}$, $G_P$, $G_Q$, $T_{max}$)
    $\overline{T'} \leftarrow \overline{T_s}$, $\overset{res}{\overline{R}} \leftarrow \emptyset$
    foreach $[T_{s,Q}, T_{e,Q}] \in \mathsf{G_Q}.\overline{T_\top}$ do
        /* Confirm instances where $Q$ holds directly at the
           start point.                                              */
        $\overset{res}{\overline{R_1}} \leftarrow (\overline{T'} \cap [T_{s,Q}, T_{e,Q}])|_\top$
        $\overline{T'} \leftarrow \overline{T'} \setminus [T_{s,Q}, T_{e,Q}]$
        /* If $P$ has been true continuously for an interval
           before $T_{s,Q}$, we can confirm additional goals.        */
        if $\exists [t_s, t_e] \in \mathsf{G_P}.\overline{T_\top}\,.\,T_{s,Q} \in [t_s, t_e]$ then
            $T_l \leftarrow \max(t_s, T_{s,Q} - T_{max})$
            $\overset{res}{\overline{R_2}} \leftarrow (\overline{T'} \cap [T_l, T_{s,Q}])|_\top$
            $\overline{T'} \leftarrow \overline{T'} \setminus [T_l, T_{s,Q}]$
        else
            $\overset{res}{\overline{R_2}} \leftarrow \emptyset$
        end
        $\overset{res}{\overline{R}} \leftarrow \overset{res}{\overline{R}} \cup \overset{res}{\overline{R_1}} \cup \overset{res}{\overline{R_2}}$
        if $\overline{T'} = \emptyset$ then break
    end
    return $\langle \overset{res}{\overline{R}}, \overline{T'} \rangle$
end
```

Algorithm 5.20: Rejection of *until* goals that lie within a $\neg P$-interval.

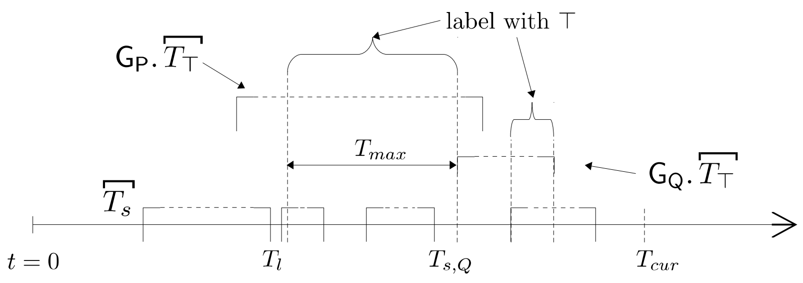
```
function rejectUntilGoalsP($\overline{T_s}$, $G_P$)
    $\overline{T'} \leftarrow \overline{T_s}$                    /* unhandled start times */
    $\overset{res}{\overline{R}} \leftarrow \emptyset$
    foreach $[T_s, T_e] \in \mathsf{G_P}.\overline{T_\perp}$ do
        $\overset{res}{\overline{R'}} \leftarrow (\overline{T'} \cap [T_s, T_e])|_\perp$
        $\overset{res}{\overline{R}} \leftarrow \overset{res}{\overline{R}} \cup \overset{res}{\overline{R'}}$
        $\overline{T'} \leftarrow \overline{T'} \setminus [T_s, T_e]$
        if $\overline{T'} = \emptyset$ then break
    end
    return $\langle \overset{res}{\overline{R}}, \overline{T'} \rangle$
end
```

Algorithm 5.21: Rejection of inevitably interrupted *until* goals.

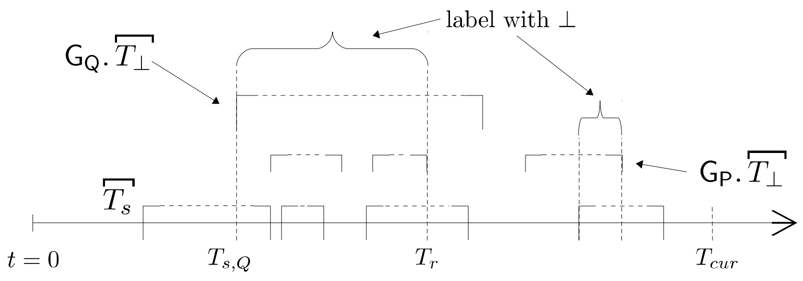
```
function rejectUntilGoalsPQ($\overline{T_s}$, $G_P$, $G_Q$)
    $\overline{T'} \leftarrow \overline{T_s}$                    /* unhandled start times */
    $\overset{res}{\overline{R}} \leftarrow \emptyset$
    foreach $[T_{s,Q}, T_{e,Q}] \in \mathsf{G_Q}.\overline{T_\perp}$ do
        $\overline{T_{intr}} \leftarrow \mathsf{G_P}.\overline{T_\perp} \cap [T_{s,Q}, T_{e,Q}]$
        if $\overline{T_{intr}} \neq \emptyset$ then
            $T_r \leftarrow max_t(\overline{T_{intr}})$
            $\overset{res}{\overline{R'}} \leftarrow (\overline{T'} \cap [T_{s,Q}, T_r])|_\perp$
            $\overset{res}{\overline{R}} \leftarrow \overset{res}{\overline{R}} \cup \overset{res}{\overline{R'}}$
            $\overline{T'} \leftarrow \overline{T'} \setminus [T_{s,Q}, T_r]$
        end
        if $\overline{T'} = \emptyset$ then break
    end
    return $\langle \overset{res}{\overline{R}}, \overline{T'} \rangle$
end
```

Algorithm 5.22: Rejection of expired *until* goals.

```
function detectTimedOutUntilGoals($\overline{T_s}$, $G_Q$, $T_{max}$)
    $\overline{T'} \leftarrow \overline{T_s}\ \overset{res}{\overline{R}} \leftarrow \emptyset$
    foreach $[T_s, T_e] \in \mathsf{G_Q}.\overline{T_\perp}$ do
        if $T_e - T_s \geq T_{max}$ then
            $\overset{res}{\overline{R'}} \leftarrow (\overline{T'} \cap [T_s, T_e - T_{max}])|_\perp$
            $\overset{res}{\overline{R}} \leftarrow \overset{res}{\overline{R}} \cup \overset{res}{\overline{R'}}$
            $\overline{T'} \leftarrow \overline{T'} \setminus [T_s, T_e - T_{max}]$
        end
        if $\overline{T'} = \emptyset$ then break
    end
    return $\langle \overset{res}{\overline{R}}, \overline{T'} \rangle$
end
```

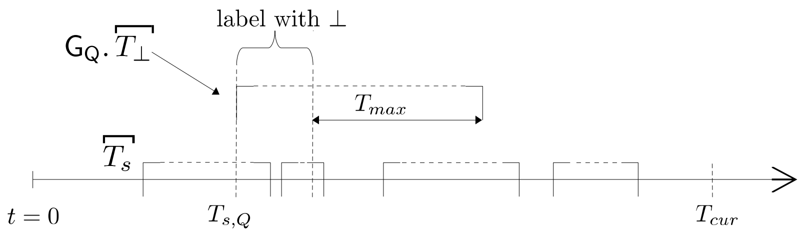


(a) Determining ⊤ instances.



label with ⊥
$\mathsf{G_Q}.\overline{T_\perp}$
$\mathsf{G_P}.\overline{T_\perp}$
$\overline{T_s}$
$t=0$
$T_{s,Q}$
T_r
T_{cur}

(b) Detecting terminated goal instances.



(c) Detecting expired goal instances.

Figure 5.17: Calculation of results for *until*.

5.7 Validation of the Property Evaluation Mechanism

The detailed description of the property evaluation algorithm in this chapter presented a semi-formal justification for the claim that the mechanism that is based on querying and manipulating temporal interval sequences indeed realizes the semantics of the SALMA property specification language as defined in Section 4.1. A formal proof for the algorithm's correctness would not add much further insight and would be very complex to do thoroughly. Therefore, such a formal proof has not been done in the context of this thesis.

Instead, an extensive collection of automated tests has been used to gain confidence in the correctness of SALMA's property evaluation module. One part of these tests are complete simulation experiments with predictable probabilistic outcomes that are combined with SALMA-PSL properties for which the success probability can be calculated in closed form. An example for such a setup was already presented in Section 4.3. These kinds of system level tests can certainly increase the confidence that SALMA as a full statistical model checking stack works as expected. However, they focus on demonstrating expected functionality instead of systematically testing for errors. In some sense, such tests could be regarded as *acceptance tests* since they focus on what the user actually expects from the software (see, e.g. [PY08, Chap. 2.1, Chap. 22.3]). On the other hand, tests that are suitable for helping the developer with detecting faults, should be functional tests with much finer granularity that systematically attempt to cover different paths of the property evaluation algorithm. In fact, what is needed are subsystem- or module tests (see [PY08, Chap. 21]) that operate directly on the Prolog interface to the property evaluation mechanism and the situation calculus model instead of using the full simulation stack. This avoids any probabilistic uncertainty and facilitates locating faults.

One of the main challenges for the creation of automatized tests for more complex algorithms is that the test code itself can become hard to understand and validate for human reviewers, which would significantly degrade the value of these tests. Therefore, it is necessary to create suitable abstractions for describing *test scenarios*, i.e. collections of concrete inputs together with expected outcomes. For SALMA's property evaluation mechanism, a test case is represented by a SALMA PSL property, a deterministic sequence of actions and events with fixed occurrence times, and of *expectations* for the verdicts and evaluation schedule states at given points in time. Along these lines, a Prolog test API has been created that integrates a simple domain model, which is roughly similar to the delivery robots example from Chapter 3. Figure 5.18 shows one test case that was specified with this API. It refers to the formula that is included in the top of the figure. This property requires that any robot that grabs an item continues carrying that item until it trespasses a certain goal within a given time limit. As in the example in Section 4.3, all robots move strictly along the x-axis by one unit in each time step, which makes it easy to predict the behavior of the system.

The test case in Figure 5.18 is specified using the function `runTest` of the test API. All necessary information is passed via three lists of terms. First, a sequence of `ev` terms is given that contain events or actions together with the time steps at which they occur. The second list specifies expectations for the verdicts that are yielded by the property evaluation module at each time step. Each `expect` term contains the start and the end of an interval together with the verdict that is expected for each step within this interval. Similarly, the third list contains expectations for the content of the evaluation schedule that are represented by intervals together with lists that contain terms with the pattern `s(t1, t2) : result`. Each of these terms declares a mapping of an interval to an evaluation result, so each list in an `expect` corresponds to a result mapping in the sense of Section 5.3.3. In the example in Figure 5.18, the first list describes a scenario in which the first robot grabs its item at time step 0 and the second robot does the same at step 3. The second list declares that the evaluation of F yields a positive verdict in every step except in the steps where the `grab` actions happen. At these points, i.e. step 0 and 3, both robots have not yet reached their goal ($xpos > 20$) and the occurrence of the `grab` action "activates" the `until` operator. Therefore, a conclusive verdict cannot be determined in steps 0 and 3 and in both steps a corresponding evaluation goal has to be scheduled. This also has to be reflected by the content of the evaluation schedule, which is verified by the expectations that are given in the third list in the call to `runTest`. What is not shown here is that `runTest` initializes the model with a *test fixture* in which the two robots `rob1` and `rob2` are at the positions $\langle x = 10, y = 10\rangle$ and $\langle x = 10, y = 20\rangle$, respectively. This means that if the test case shown in Figure 5.18 is executed, all expectations will be fulfilled and the test is successful. On the other hand, if the event sequence or any of the expectations were altered, the test case would fail with an assertion exception that describes the failure.

Using the high-level language described above, it is possible to systematically construct test cases that, when executed together as a *test suite*, achieve a high *test coverage*, i.e. exercise a significant part of the property evaluation mechanism. Although several different *coverage criteria* have been defined in the software testing literature, the most widely used are statement coverage and branch coverage, which measure the ratio of statements or condition branches that are traversed during execution of the test suite (see [PY08, Chap. 12]). One reason why these criteria are chosen so frequently is that statement and branch coverage can easily be measured automatically. This is done using *code instrumentation*, which essentially means inserting auxiliary statements at certain points in the tested program that record when they are executed. However, although ECL^{i}PS^e actually provides facilities for performing this instrumentation automatically, it was not possible to leverage them

```
test case(until_ok_two_robots) :-
    F = forall(r: robot, forall(i: item,
            implies(occur(grab(r,i)),
                until(15, carrying(r, i), xpos(r) > 20)))),
    runTest(F, 20,
        [ev(0, grab(rob1, item1)),
            ev(3, grab(rob2, item2))],
        [expect(0, 0, nondet),
            expect(1, 2, ok),
            expect(3, 3, nondet),
            expect(4, 20, ok)],
        [expect(1, 3, [s(0,0) : nondet]),
            expect(4, 10, [s(0,0) : nondet, s(3,3) : nondet]),
            expect(11, 11, [s(0,0) : ok, s(3,3) : ok]),
            expect(12, 20, [])]).
```

Figure 5.18: Example test cases for the property evaluation mechanism.

for the SALMA property evaluation mechanism because the instrumentation failed for a technical reason that could not be resolved in time. As a pragmatic solution for this problem, the instrumentation has been done manually. First, machine-readable comments were added to the Prolog source code of the property evaluation module to mark the (relevant) branches of the algorithm. Then, a Python script was used to generate a copy of that source code with the markers replaced by statements that record any entrance of their branch by increasing a counter in a globally accessible hash table. Additionally, a CSV file is created that contains a list of all inserted probes together with the corresponding source file and line number. The test API provides a command `report_coverage` that compares the content of this CSV file with the probe counters in the hash table and uses this to calculate a coverage ratio. Guided by this metric, a test suite has been created that covers all relevant branches of the property evaluation algorithm and the property compiler. It is publicly available together with the main source code of the SALMA toolkit at the SALMA website [Kro16].

As mentioned in the beginning of this chapter, not only the correctness of the property evaluation algorithm is important but also its efficiency. In particular, an efficient handling of the evaluation goal schedule is crucial for cases in which temporal operators have to be evaluated for many consecutive time steps. An example for a case like this can be seen in formula F in Figure 5.19. With a naive approach, every step in which a robot is carrying an item would cause the addition of a new entry to the evaluation schedule. Since all entries of the schedule have to be re-evaluated in each step, this would mean that the run time of evaluation steps increase approximately linearly over time. For

```
F = forall(r: robot, forall(i: item,
          implies(carrying(r, i), eventually(2000, xpos(r) >  3000))))

G = forall(r: robot, forall(i: item,
          implies(carrying(r, i),
                let(maxX : xpos(r) + 2000, eventually(2000, xpos(r) > maxX)))))
```

Figure 5.19: Example formulas for testing the efficiency of the property evaluation mechanism.

long-running experiments, this can obviously become problematic, especially since this effect is amplified for formulas with nested temporal operators.

However, as this chapter described, SALMA's property evaluation algorithm uses an interval-based approach in which many goal schedule instances can be handled together. In fact, for the formula F, the algorithm is able to merge all goal instances to one single interval. This also means that the run time for each evaluation step should approximately stay constant. In contrast to this, formula G in Figure 5.19 is constructed in a way so that the property evaluation algorithm cannot merge scheduled goals because the robot's target is set to a new value in each step through the binding of the variable $maxX$. This effectively means that a new version of the property G is created in each time step and has to be scheduled separately. As mentioned above, this means that the runtime of evaluation steps for property G should increase over time. To verify these expectations, two simulations were performed, with a length of 1000 steps each, and for each simulation only one of the two properties were registered. In each step, all robots moved one step further along the X-axis but no other event or action was performed during the simulation runs. The durations of the property evaluation phase (ΔT) in each step were measured using the high-resolution timer provided by the operating system, which has a resolution of 1 µs. Figure 5.20 shows the results for both simulation runs. In the left diagram, which contains the graph for the evaluation of F, it can be seen that the evaluation run-times for F stay almost constant between 0.4 ms an 7.1 ms over the whole simulation. The peaks occur at random points and have to be caused by internal effects within the ECL^{i}PS^e interpreter or the operating system. On the other hand, the graph with the evaluation run-times for G in the right diagram clearly shows the predicted linear growth.

Altogether, this example demonstrates the benefit of the interval-based representation of the evaluation goal schedule that was described in this chapter. However, it also reveals that for some cases, the algorithm is not able to coalesce the schedule entries and therefore a growth of the schedule size cannot be avoided, which leads to an increase of the evaluation step run-time. In cases

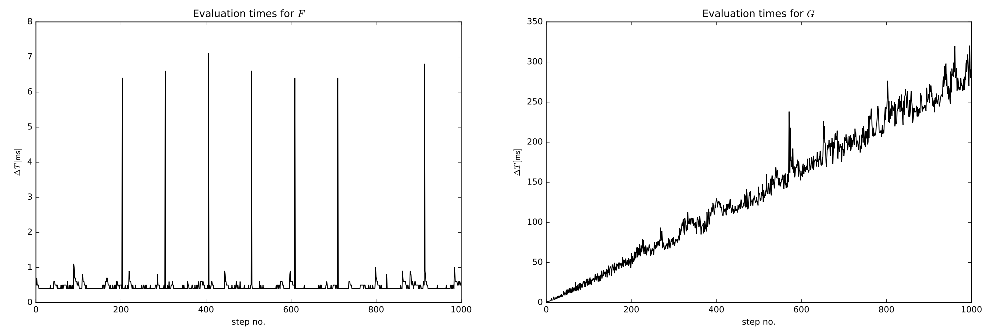


Figure 5.20: Evaluation run-times for formulas F an G from Figure 5.19.

when bounds of temporal operator are set very high like in F or G, this could lead to significant performance problems. Some of these problematic cases could certainly be mitigated by further optimization of the evaluation algorithm. In general, however, the modeler remains responsible for avoiding these situations by choosing appropriate time bounds, constraints, and conditions for the checked formulas.

5.8 Summary

This chapter presented the most complex part of SALMA that can also be regarded as one of the most significant contributions of this thesis, namely an efficient evaluation mechanism for properties specified in the SALMA property specification language (SALMA-PSL), which was introduced in Chapter 4 as a first-order variant of time-bounded LTL that is tied closely to the situation calculus. The chapter explained both the technical architecture and the algorithmic design of the evaluation mechanism in detail. It was shown that essentially two main ideas can be found at its foundation: the consequent use of logic programming principles within the property evaluation algorithm and an evaluation goal scheduling mechanism that uses *temporal interval sequences* to represent the recorded history of property evaluation results in an efficient way.

The chapter first introduced the overall structure of the property evaluation process in SALMA and described its main goals as well as the challenges it has to face. In particular, it was explained how the fact that the SALMA simulation uses event scheduling with *variable time advances* imposes special requirements for formula interpretation and for the way the evaluation system keeps track of states for which a conclusive decision about the property result cannot be made immediately.

Following this introduction, a specialized data structure called temporal interval sequence was presented and defined formally together with a set of operators that are used within the property evaluation algorithm. Additionally, it was described how SALMA-PSL formulas are translated by the *property compiler* to an intermediate language that is closer to the native Prolog execution structure and can therefore be interpreted more easily.

Based on this foundation, the evaluation goal scheduling mechanism was described, which uses the temporal interval sequences defined before to keep track of the property evaluation results for all passed time points. It was shown how the interval-based representation together with a multi-layer reference scheme and optimizations through term rewriting were used to achieve an efficient solution for evaluation goal scheduling.

The property evaluation algorithm itself was presented in great detail. In particular, it was shown how the temporal operators of the SALMA property specification language can be handled by means of operations on temporal interval sequences. As demonstrated in this chapter, this approach often allows the evaluation module to calculate and store the result of a formula for many time instances at once. It is obvious that this is much more efficient than handling each instance separately.

Instead of a formal proof for the correctness of the introduced algorithms, the chapter provided more comprehensible justifications for their design by means of detailed interval diagrams. Additionally, it was explained how automated tests can be realized that systematically exercise all relevant parts of the algorithm and validate the results. Finally, the efficiency benefits of the algorithm's interval-based representation of the evaluation state were demonstrated by means of a short performance experiment.

5.9 Related Work

One of the core technical aspects of SALMA's formula evaluation approach is that the state of the algorithm is represented as a series of temporal intervals. The data structures and the operations that are used in SALMA to query and update the temporal data were defined rather rigorously in Section 5.3 of this chapter, mainly with the purpose of setting the stage for the later discussion of the evaluation algorithm itself. To some extent, the temporal interval sequences used in this chapter provide similar functionality as *interval trees* [CLRS01, sec. 14.3], which are basically balanced binary search trees that store intervals in their nodes. However, although the tree-based representation would be more efficient in the general case, SALMA realizes interval sequences by regular Prolog lists. This makes sense since data is always accesses in linear order within each iteration of the SALMA evaluation algorithm, so the benefits of using a balanced tree structure would not come into effect.

Another important part of the SALMA property evaluation module that was described in this chapter is evaluation goal scheduling, i.e. the way the evaluation algorithm keeps track of states for which the decision is still pending whether they satisfy a formula containing temporal operators. This aspect is typically not addressed in the description of statistical model checking approaches. Typically, like in [SVA04], the evaluation procedure is described in a functional fashion in which the algorithm can traverse through the samples by means of recursion. In contrast to this, it could be said that SALMA uses a *reactive* approach since the evaluation progresses as a result of reactions to events and actions that are submitted by the simulation engine in each step. In this respect, the SALMA property evaluation mechanism actually bears resemblance to *runtime verification* and *runtime monitoring* approaches. There, properties, which are often specified in temporal logics like LTL, are evaluated against action and event traces that are generated by the system at runtime. This means that a *reactive* evaluation structure is inevitable, and different solutions have been designed for that. For instance, in [BLS11] the authors describe a mechanism that synthesizes automata-based monitors for LTL and timed LTL (TLTL) formulas. Similar to SALMA's representation of the evaluation state of a subformula by means of three values ($\top$, $\bot$, and ?), they actually use a three-valued temporal logic they call LTL_3 that is able to express the fact it may not yet be possible to make a definite decision for a given formula at the current time step. A different approach for runtime verification is presented in [BGHS04], where *rewriting rules* are used to transform formulas in each step so that the resulting formulas capture the updated evaluation state.

As mentioned before, one of the core benefits of SALMA's first-order version of LTL as property description language is that it allows expressing requirements on a more concrete level than with the widely used propositional style. As runtime verification is per definition concerned with the concrete behavior of the real system, it is not surprising that there have been attempts to lift the property specification languages in runtime verification from propositional temporal logics to first-order versions. For example, in [BKV13], the authors describe a runtime monitoring approach that is able to detect violation or compliance of monitored action sequences with respect to properties formulated in a first-order variant of LTL. The monitor construction algorithm uses an automaton type the authors call *spawning automaton* that extends is constructed similar to the generalized Büchi automaton (GBA) that is typically used as a monitor for LTL. The spawning automaton adds a *spawning function* that is triggered when an event occurs and creates sub-automata for quantified subformulas in which the bound variables are instantiated corresponding to the attributes of the event. From this point on, incoming events are routed through to these spawned sub-automata and the conjunction or disjunction over their verdicts realize the quantification over all instances observed so far. In this sense, these sub-automata are similar to entries on the evaluation goal schedule of SALMA (cf. Section 5.5). However, although automata-based monitoring is clearly less computationally expensive than SALMA's evaluation algorithm, the latter allows for much more flexibility, e.g. the integration of custom Prolog functions or predicates. However, it could be worth investigating a combined solution in which monitors for suitable subformulas are precompiled into automata to optimize the evaluation efficiency.

Chapter 6

Modeling Information Transfer in Cyber Physical Multi-Agent Systems

In this chapter, an extension of SALMA (and the situation calculus in general) is presented which explicitly addresses one aspect that is particularly important for *cyber-physical* [Lee08] multi-agent systems, namely the distributed gathering and transfer of information. Agents not only have to continuously sense their environment, but also share these readings with other agents, acquire information of others, and participate in coordination activities. In the cyber-physical context, these information transfer processes are subject to stochastic effects, e.g. due to sensor errors or unreliable communication channels. Furthermore, accuracy and timing of information transfer can strongly influence the behavior of the whole system. In particular, the efficacy of mechanisms for self-adaptation or optimization typically degrades when certain time-constraints are violated or the accuracy of sensors is insufficient.

Using pure logical formalisms like the basic situation calculus for describing such scenarios results in rather verbose and tedious representations that are not practicable in more complex cases. What is needed instead are high-level constructs that establish a bridge between the underlying logical semantics and the typical requirements for modeling information transfer in multi-agent CPS. Although higher-level extensions on top of the situation calculus have been designed for related aspects like sensing and knowledge (e.g. [SL03]), there has, to the author's knowledge, not been a detailed reflection of information propagation in CPS in the context of the situation calculus.

Therefore, in the context of this thesis a generic model of information transfer was developed that is based on a stochastic timed version of the situation calculus and allows capturing a wide range of effects that may be imposed on information transfer processes. Additionally, a set of macro-like abstractions for common information transfer scenarios within CPS were defined, such as message passing or sensor data propagation. This creates a concise interface for the modeler that hides the stochastic details of information propagation but makes them fully accessible in simulation and verification.

In the following sections, the scene is first set by means of an example from the e-mobility domain, namely the automatic parking lot assignment scenario that was already mentioned shortly in the introduction of this thesis. The main contribution of this chapter starts in Section 6.2, where the generic model for information transfer is introduced and it is described how this model is mapped to the situation calculus. After that, Section 6.3 defines several extensions to SALMA's modeling language that provide pragmatic abstractions for the information transfer model. This is continued in Section 6.4, where the focus is set on the use of SALMA for statistical model checking in the context of information transfer processes. As an evaluation of the presented approach, Section 6.5 discusses its application to the parking lot assignment example that was introduced in the introduction of this thesis and describes experiences gained during first concrete experiments.

Remark: *The content of this chapter has mainly been taken over from [KB16] which is the result of a collaboration with Tomáš Bureš during his stay at the PST chair. Besides giving valuable advice on the structure of that paper, Tomáš provided background information about the e-mobility case study that was used as an example and about cyber-physical systems in general. Additionally, he contributed to the elicitation and description of related work.*

6.1 Running Example: Optimized Parking Lot Assignment

As a running example to illustrate the information transfer extension of SALMA, a scenario was used that was derived from the e-mobility case-study of the EU project ASCENS [1] that has been described before, e.g., in [B^+13]. The case study focuses on a scenario in which electric vehicles compete for parking lots with integrated charging stations (PLCS) in an urban area. The goal is to find an optimal assignment of PLCS to vehicles. Technically, the assignment is performed by an agent called super-autonomic manager (SAM) that acts as a central communication hub for vehicle requests and monitors the capacity of all PLCS in the system. Figure 6.1 shows an overview of the communication protocol between all agent roles. The basic idea is that vehicles send *assignment requests* to the SAM, including a start time, a duration, and a list of preferred PLCS that is compiled by the vehicle's on-board computer. The SAM tries to find optimal suggestions for parking lot assignments, based on the knowledge about driver's intentions, and on occupancy information that is

[1] `www.ascens-ist.eu`

sent repeatedly by the PLCS. The suggested PLCS assignments are sent back to the vehicles, who in turn send reservation requests to the corresponding stations. Each PLCS checks again whether reservations can be granted and responds accordingly. This two-stage protocol allows reservations to be made even when the SAM is not reachable. In this case, vehicles would simply send out requests to the PLCS that is closest to the intended destination and a PLCS would grant reservations on a first-come first-served basis.

In fact, true to the distributed CPS principle, all the agents (vehicles, PLCS, SAM) are *autonomous* and communicate via some wireless data transmission infrastructure like a VANET or 3G/4G network. This implies that neither transmission delays nor the possibility of errors can be neglected. At the same time, timing clearly plays an important role in the scenario described above. First of all, the reservation service would simply not be accepted if the delay between reservation requests and reservation responses was too high. Also, the communication timing affects the convergence of the optimization, thus it directly influences the functionality of the distributed CPS.

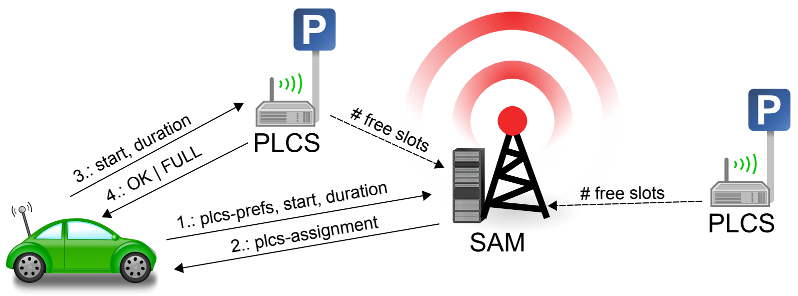
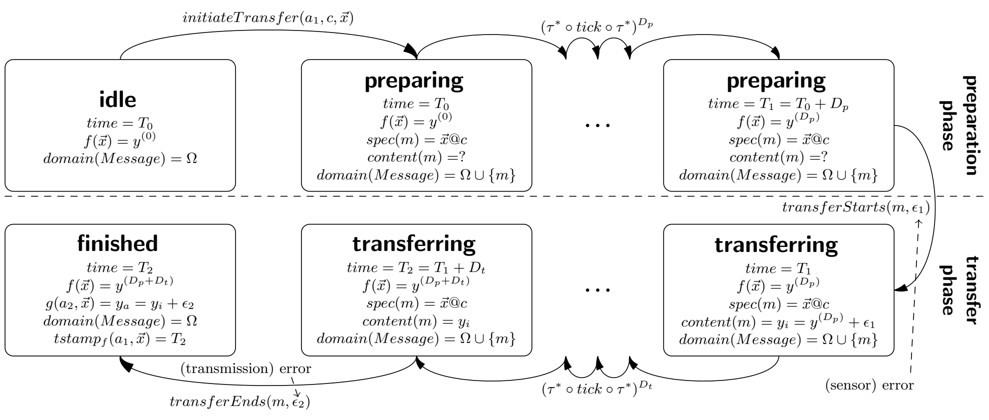


Figure 6.1: Optimized parking lot assignment scenario.

6.2 A Generic Situation Calculus Model for Information Transfer

In order to use SALMA for analyzing scenarios like the one described in Section 6.1, concepts like sensing and communication have to be mapped to SALMA's modeling language framework. As a first step, a generic model for information transfer in the situation calculus is developed. This model is able to describe both sensing and inter-agent communication in a unified way and allows capturing stochastic effects with a variable level of detail. This section introduced the main principles of this model and outlines how they are expressed by means of situation calculus concepts. After that, Section 6.3 shows how the model is actually used in SALMA through a set of high-level language constructs that facilitate addressing the main interaction patterns in CPS.



$initiateTransfer(a_1, c, \vec{x})$
$(\tau^* \circ tick \circ \tau^*)^{D_p}$
idle
$time = T_0$
$f(\vec{x}) = y^{(0)}$
$domain(Message) = \Omega$
preparing
$time = T_0$
$f(\vec{x}) = y^{(0)}$
$spec(m) = \vec{x}@c$
$content(m) = ?$
$domain(Message) = \Omega \cup \{m\}$
...
preparing
$time = T_1 = T_0 + D_p$
$f(\vec{x}) = y^{(D_p)}$
$spec(m) = \vec{x}@c$
$content(m) = ?$
$domain(Message) = \Omega \cup \{m\}$
preparation phase
$transferStarts(m, \epsilon_1)$
finished
$time = T_2$
$f(\vec{x}) = y^{(D_p+D_t)}$
$g(a_2, \vec{x}) = y_a = y_i + \epsilon_2$
$domain(Message) = \Omega$
$tstamp_f(a_1, \vec{x}) = T_2$
transferring
$time = T_2 = T_1 + D_t$
$f(\vec{x}) = y^{(D_p+D_t)}$
$spec(m) = \vec{x}@c$
$content(m) = y_i$
$domain(Message) = \Omega \cup \{m\}$
...
transferring
$time = T_1$
$f(\vec{x}) = y^{(D_p)}$
$spec(m) = \vec{x}@c$
$content(m) = y_i = y^{(D_p)} + \epsilon_1$
$domain(Message) = \Omega \cup \{m\}$
transfer phase
(transmission) error
$transferEnds(m, \epsilon_2)$
$(\tau^* \circ tick \circ \tau^*)^{D_t}$
(sensor) error

Figure 6.2: Overview of the general information transfer model.

In general, the approach is based on the notion that information is transferred from a *source fluent* to a *destination fluent* that is directly accessible by the receiving agent. The source fluent can either represent a *feature of the physical world* or data created by some artificial process, e.g. a message queue. A *connector* is a virtual entity that defines modalities of an information transfer process, including the fluent endpoints and the types and roles of participating agents. The *messages* that are transmitted over connectors are treated as first-level model citizens by representing them as entities of the dedicated sort Message. Both the content and the state of each message are stored separately by a set of auxiliary fluents and evolve independently as result to several types of events. This representation provides great flexibility for the realization of arbitrary propagation structures.

In the following, the information transfer model will first be introduced from a more abstract perspective as a combination of two general *phases* of the information propagation. Later, Section 6.2.2 will describe more concretely how this model is actually integrated into SALMA.

6.2.1 Information Transfer Phases

Based on the foundational concepts described above, two phases of information transfer are distinguished, which is visualized in Figure 6.2:

A) The **preparation phase** starts when an agent (a_1 in Figure 6.2) initiates the information transfer (sensing or communication). For that, the agent specifies a *connector* (c) and a parameter vector ($\vec{x}$) that fully qualifies the information source and, in case of a point-to-point transmission, contains the identity of the receiving agent. In response to this action, a new message (m) is created and initialized with the transfer metadata but without content yet. Depending on the concrete scenario, the actual transfer can be delayed for various reasons, for instance because sensors or communication devices have to be initialized first. This means that there may be an arbitrary sequence of time steps (*tick* events), which could be interleaved with actions and events (denoted as τ in Figure 6.2) that change the information source (f) but are not recognized by the agent. Hence, after that sequence, the actual value that is eventually used as message content can deviate from the value of the information source that was present at the time when the transfer was initiated.

B) The **transfer phase** follows the preparation phase and begins when a *transferStarts* event occurs. At that point, the current value of the source fluent f is fixated as the content of the message. This message is now transferred to its destination over the connector c whose stochastic characteristics are specified within the simulation model. Like above, this phase may take an arbitrary amount of time during which unrecognized or unrelated actions and events occur. Eventually, a *transferEnds* event finishes the transfer process. Thereupon, the destination fluent instance $g(a_2, \vec{x})$ is updated and the message entity is removed. This moment, as well as the starting points of both phases, are memorized in timestamp fluents that can, for instance, be used to reason about the age of a measurement.

The diagram in Figure 6.2 omits the fact that, due to malfunctions and disturbances in the environment, the transfer could *fail* at any time, which would be represented by an additional event *transferFails*. Additionally, the transfer process may be affected by stochastic errors that eventually cause the received value to deviate from the original input, which is reflected by the error terms ϵ_1 and ϵ_2 in the events *transferStarts* and *transferEnds*.

In general, both stochastic errors and delays are governed by a set of probability distributions that are used during simulation to decide when the events mentioned above occur and which errors they introduce. By adjusting these parameters, a wide variety of different scenarios can be modeled, ranging from nearly perfect local sensing to wireless low-energy communication with interferences. This topic is examined further in Section 6.2.4.

6.2.2 Information Transfer Paradigms

The generic abstract model presented above contains several degrees of freedom, in particularly with respect to how the content of messages is set, and how the recipients are determined. These variation points have to be handled according to the particular kind of information transfer. In the following, four classes of information transfer are presented which cover the typical scenarios in cyber-physical multi agent systems. To make this section more comprehensible, the situation calculus semantics of each class is described in a reduced form by means of annotated state diagrams. A more detailed axiomatization will be given later in Section 6.2.5.

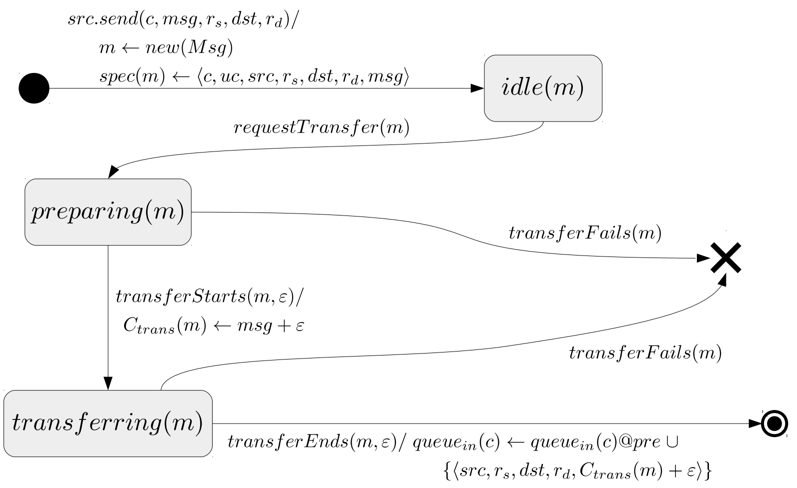
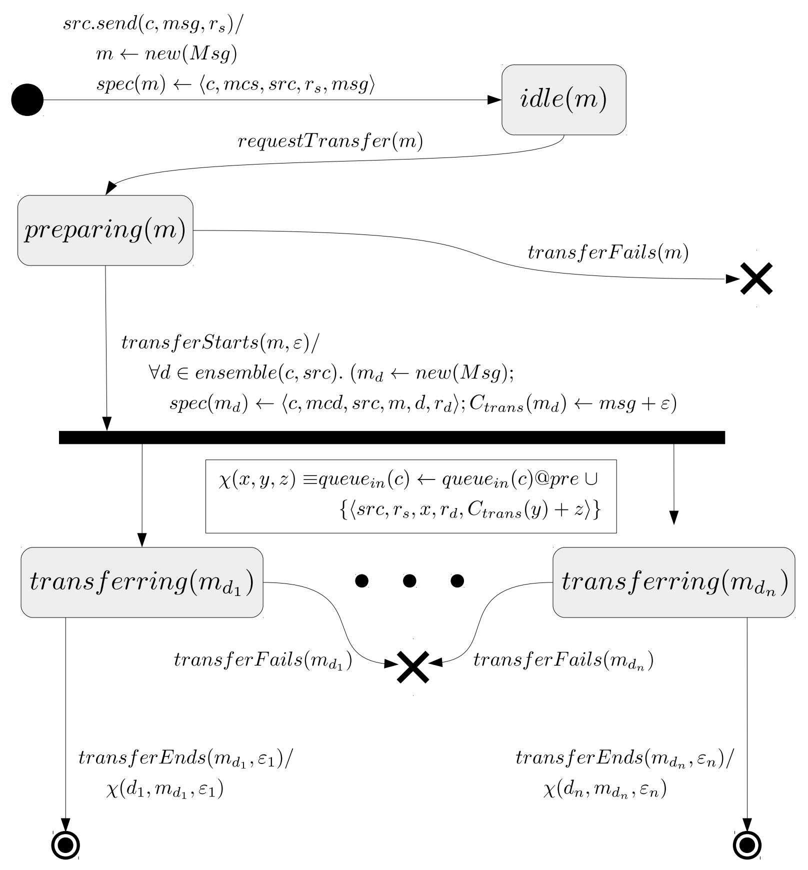

$src.send(c, msg, r_s, dst, r_d)/$
$m \leftarrow new(Msg)$
$spec(m) \leftarrow \langle c, uc, src, r_s, dst, r_d, msg\rangle$
$idle(m)$
$requestTransfer(m)$
$preparing(m)$
$transferFails(m)$
$transferStarts(m, \varepsilon)/$
$C_{trans}(m) \leftarrow msg + \varepsilon$
$transferFails(m)$
$transferring(m)$
$transferEnds(m, \varepsilon)/\ queue_{in}(c) \leftarrow queue_{in}(c)@pre \cup$
$\{\langle src, r_s, dst, r_d, C_{trans}(m) + \varepsilon\rangle\}$

Figure 6.3: Unicast channel based communication.

Generally, the following basic types of information transfer are identified:

1. **Channel-based communication:** An agent actively sends data to one or several other agents. The well-known channel paradigm fits well to the asynchronous communication style predominant in CPS and to the relational way of identifying information in the situation calculus. As usual, a distinction is made between the following communication schemes:

   a) **Unicast (point-to-point)**: Figure 6.3 shows a unicast message transfer from agent *src* to agent *dest* via channel *c*. Here, a message is sent from the source to a single destination that is specified when the transfer is initiated. In the situation calculus model, this means that the specification of the message $spec(m)$ is set when the message is created. This specification includes all relevant static information about the message, namely the channel (c), the message type (*uc* for *unicast*), source and destination, as well as source and destination roles (r_s and r_d, respectively) and the original message content (msg). The actual *transferred content* (fluent C_{trans}) of this message is set when the transfer starts, possibly tampered by an error ε originating in the preparation phase. When the message transfer ends successfully (event *transferEnds* occurs), a tuple is added to the channel's incoming message queue ($queue_{in}$). This message tuple identifies source and destination, their roles with respect to the channel definition, and the eventually received message content that again might incorporate an error originating from the transfer phase. In contrast, when the transfer fails (event *transferFails* occurs) either during the preparation or transfer phase, the information is lost. In both cases, the message entity is removed from the model.



$src.send(c, msg, r_s)/$
$m \leftarrow new(Msg)$
$spec(m) \leftarrow \langle c, mcs, src, r_s, msg\rangle$
$idle(m)$
$requestTransfer(m)$
$preparing(m)$
$transferFails(m)$
$transferStarts(m, \varepsilon)/$
$\forall d \in ensemble(c, src).\ (m_d \leftarrow new(Msg);$
$spec(m_d) \leftarrow \langle c, mcd, src, m, d, r_d\rangle; C_{trans}(m_d) \leftarrow msg + \varepsilon)$
$\chi(x, y, z) \equiv queue_{in}(c) \leftarrow queue_{in}(c)@pre \cup \{\langle src, r_s, x, r_d, C_{trans}(y) + z\rangle\}$
$transferring(m_{d_1})$
$transferring(m_{d_n})$
$transferFails(m_{d_1})$
$transferFails(m_{d_n})$
$transferEnds(m_{d_1}, \varepsilon_1)/$
$\chi(d_1, m_{d_1}, \varepsilon_1)$
$transferEnds(m_{d_n}, \varepsilon_n)/$
$\chi(d_n, m_{d_n}, \varepsilon_n)$

Figure 6.4: Multicast channel based communication.

b) **Multicast**: here, the destination of the message transfer is not specified directly when the message is sent but instead via some shared address property. This case is shown in Figure 6.4. Other than in the unicast case, the specification of the source message m does not contain the destination. Instead, the destinations are chosen on the arrival of a *transferStarts* event by evaluating the channels *ensemble predicate* (see Section 6.2.3). For each selected recipient, the source message is replicated and hence transferred on an independent path. In particular, terminating messages (*transferEnds* and *transfer*) occur independently for each message, which allows capturing phenomena that are caused by lack of synchronization or deviating information among agents.

2. **Generalized sensing:** an agent acquires information about a feature of the world that can be assessed through sensing. This can be refined further into local (or direct) and remote sensing:

a) **Direct (local) sensing**: the querying agent can produce the desired result on its own, although the sensing process may take a considerable

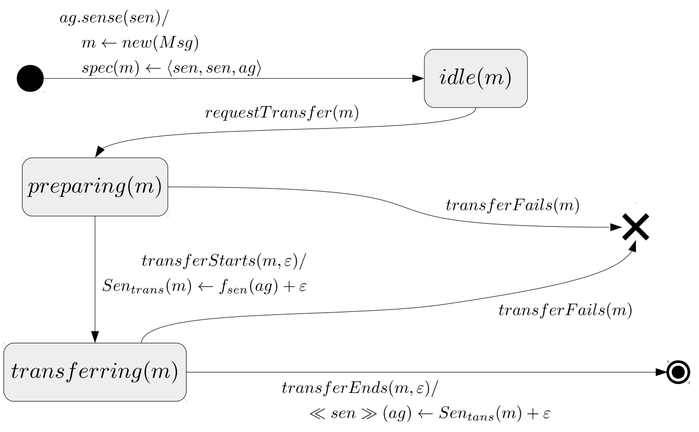


Figure 6.5: Local sensing.

amount of time and can be disturbed by internal or external factors.

In Figure 6.5, agent *ag* uses its local sensor *sen* to make a measurement. Different from the communication models explained above, the transferred content of the sensor message (Sen_{trans}) is not specified at the creation of the message but retrieved from the *sensed fluent* (f_{sen}) when *transferStarts* occurs. Also, the received sensor values are not written to a queue like messages but are used to update a *local sensor view* fluent (*sen* in Figure 6.5) that the agent can access.

b) **Remote sensing**: the agent cannot observe the desired information itself by using its local sensors but has to gather information from one or several other agents.

The remote sensing abstraction reflects the delays and disturbances of the involved communication processes but abstracts away their technical details. In particular, both for direct and remote sensing, the most recently retrieved value is made available in a *local fluent* that can be accessed without considering the underlying infrastructure. In SALMA's situation calculus model, remote sensing is realized very similar to regular intentional multicast message communication as described above. Figure 6.6 shows the transmission of data from the local sensor *sen* of agent *src* to recipients within the ensemble that is defined for the *remote sensor rsen*. The first main difference from multicast message transmission is that the transferred content is not defined explicitly by the sending agent but instead taken from the local sensor (*sen*) that is declared as the information source. Furthermore, the *ensemble* is specified from the perspective of the receiving agents. After the destination messages have been created, the rest of the transfer is identical to the multicast case from Figure 6.4.

As mentioned above, remote sensing tries to hide the communication

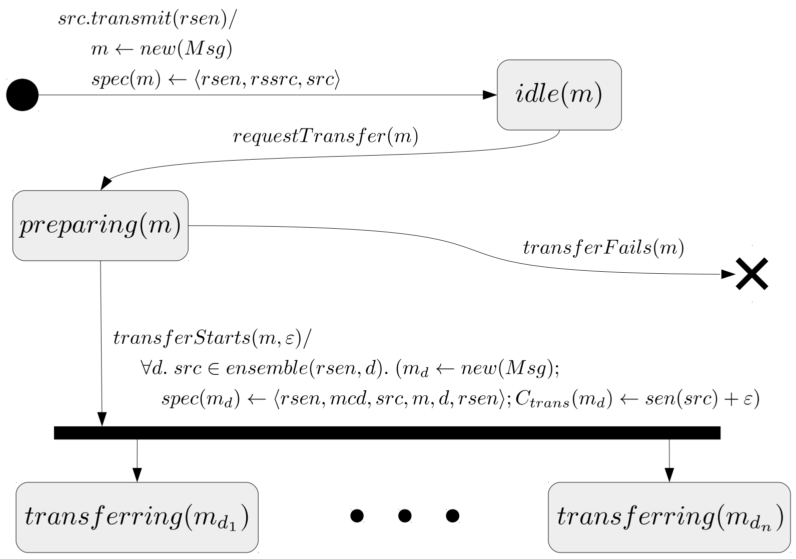

$src.transmit(rsen)/$
$m \leftarrow new(Msg)$
$spec(m) \leftarrow \langle rsen, rssrc, src \rangle$
$idle(m)$
$requestTransfer(m)$
$preparing(m)$
$transferFails(m)$
$transferStarts(m, \varepsilon)/$
$\forall d.\ src \in ensemble(rsen, d).\ (m_d \leftarrow new(Msg);$
$spec(m_d) \leftarrow \langle rsen, mcd, src, m, d, rsen \rangle; C_{trans}(m_d) \leftarrow sen(src) + \varepsilon)$
$transferring(m_{d_1})$
$transferring(m_{d_n})$

Figure 6.6: Transmission of remote sensor data.

infrastructure and therefore present the acquired data similar to local sensor data. This is achieved with the *remote sensor view*, a fluent that keeps the most recent value from each source. The actualization of this fluent is shown if Figure 6.7. It can be seen that this update step has to be performed actively by the receiving agents. This is intended as it resembles the asynchronous nature of remote sensing and allows capturing effects caused by delayed updates, etc. However, this aspect is by default hidden from the modeler as a background process that is installed automatically according to the remote sensor declarations in the model (see Section 6.3.3).

6.2.3 Predicate-based Addressing

An important concern that arises in modeling multi-agent information propagation is how the set of receiving agents is determined. In many cases, it is either impossible or impracticable to do this statically. A particularly elegant alternative, supported by SALMA, is *predicate-based addressing* [L+14]. In this approach, the set of recipients for each information transfer is determined by a *characteristic ensemble predicate* that is evaluated for each (properly typed)

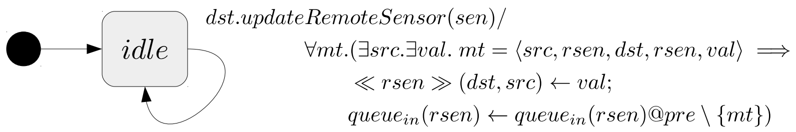

idle
$dst.updateRemoteSensor(sen)/$
$\forall mt.(\exists src.\exists val.\ mt = \langle src, rsen, dst, rsen, val \rangle \implies$
$\ll rsen \gg (dst, src) \leftarrow val;$
$queue_{in}(rsen) \leftarrow queue_{in}(rsen)@pre \setminus \{mt\})$

Figure 6.7: Reception of remote sensor data.

agent pair. An *ensemble predicate* may describe *intentional selection criteria* as well as *intrinsic constraints* imposed by agent attributes or the environment. For instance, the Prolog code in Figure 6.8, which is taken from the e-mobility example, declares an ensemble for the multicast channel *assignment*. The predicate selects all super-autonomous manager agents within a given maximum distance from a vehicle as recipients of messages that this vehicle sends on the channel. One intuitive interpretation of this "ensemble" would be that is simply reflects the maximum transmission range of the communication devices with which the vehicles are equipped.

6.2.4 Influence of the Choice of Probability Distributions

Although the core information transfer mechanisms of SALMA are to a large extend determined by situation calculus axioms, the concrete characteristics of the model are governed by the choice of probability distributions for the occurrence of the events $transferStarts$, $transferEnds$, and $transferFails$. In Section 3.4.2 it was already shown that the SALMA framework provides access to common predefined distributions like Gaussian or exponential, but also allows using custom distributions by using Python functions that have full access to the current system state. In particular, this allows defining situation-dependent distributions for the events involved in information transfer processes. For example, a reasonable choice for the duration of the transmission of messages with variable sizes would be a Gaussian distribution with a mean that is derived from the package size and the currently available channel bandwidth. If the channel and the size of message m are denoted with $\mathit{chan}(m)$ and $\mathit{size}(m)$, and if it is assumed that there is a function $\mathit{currentBandwidth}(c, S)$ that returns the currently available bandwidth capacity of channel c at situation S, then a distribution for the delay of the *transferEnds* event could be given as in (6.1) below.

$$\begin{aligned} P(\Delta T = \delta t \mid \mathit{transferEnds}(m), S) &\sim \mathcal{N}(\mu, \sigma^2) \\ \text{where } \mu &= \frac{\mathit{size}(m)}{\mathit{currentBandwidth}(\mathit{chan}(m), S)} \\ \text{and } \sigma &= F\mu \end{aligned} \tag{6.1}$$

```
ensemble(assignment, Vehicle, SAM, S) :-
  distance(Vehicle, SAM, D, S), D < max_comm_dist.
```

Figure 6.8: Example for an ensemble predicate.

The factor F is used to express the standard deviation of the delay as a linear function of the mean. This could be a constant value based on experience or a function of factors like the current message load of the communication link.

The chosen value of δt is used to schedule the *transferEnds* event according to Definition 3.27 in Section 3.6. Before that, however, a choice is made between *transferEnds* and *transferFails* according to another distribution that represents the likelihood of failures. Additionally, the error amount ε that is introduced by *transferEnds* is sampled from a third distribution. As the preparation phase is treated analogically, the whole information transfer of a message is technically governed by six probability distributions altogether. This creates a large variety of configuration choices and allows modeling a wide range of scenarios. While setting up the distributions for a simulation from scratch can clearly become a devious task for more complex models, the SALMA framework already offers several abstractions and pattern-based solutions that can reduce the effort significantly. In the future, these may be extended with additional pattern-based configuration macros that encode domain knowledge, experience, or empirical data.

6.2.5 Axiomatization of the Information Transfer Model

While the use of state transition diagrams above helped to present a comprehensible overview of the information transfer model, it has to be formalized as situation calculus axioms that can be integrated into SALMA system models. This axiomatization is mainly based upon message entities whose state and associated content is described by a set of fluents that will be defined below. Additionally, each message carries static meta-information that is defined through a set of constants that are initialized when the message is created.

Definition 6.1 (Message meta-data)**.** Let m be a message entity. The meta-data that describes the static properties of m is defined by the following constants:

type(m):	Indicates whether m is a unicast message, a multicast source or destination message, a local sensor message or a remote sensor source message. This is encoded by the value of *type*(m) being one of unicast, multicastSrc, multicastDest, sensor, or remoteSensorSrc.
src(m):	The agent that acts as the source of the message.
dest(m):	The agent that is determined as the recipient of the message m.

locSen(m):	For a remote sensor message m, *locSen*(m) contains a pointer to the fluent that represents the local sensor view from which the transferred information originates.
srcMsg(m):	For a multicast destination message, *srcMsg*(m) points to the corresponding multicast source message.
channel(m):	The channel on which a message is transmitted. For remote sensor messages, this corresponds to the name of the remote sensor.
sensor(m):	For a local sensor message m, refers to the sensor name that is also the name of the fluent that acts as the *local sensor view* of the sensor.
srcFluent(m):	For a local sensor message, contains a reference to the source fluent that represents the sensed feature of the world.
srcRole(m):	For a channel-based message contains the role of the sender.
destRole(m):	For a channel-based message contains the role of the receiver.
oppositeRole(c, r):	For a channel c and a role r, *oppositeRole*(c, r) denotes the other role that is defined in the channel declaration of c.

The representation of messages as entities requires that they can be created and removed dynamically as effect to actions and events. Unlike traditional realizations of the situation calculus, SALMA supports this by using a special (meta-)fluent *domain*(*sort*) to store the sets of entities that manifest the current *domains* of all sorts in the model. Creation and destruction of entities can therefore be controlled through regular successor state axioms. For the sort *Msg*, which represents the type of all messages, this is given in the following definition.

Definition 6.2 (Active message domain)**.** Let *Agents* be the set of all agents that are defined in the system. The domain of the sort *Msg*, which represents the set of all messages that currently exist in the system, is defined by the following SSA:

$$
\begin{aligned}
&\mathit{domain}(\mathit{Msg}, \mathit{do}(a, s)) = D \equiv a = \mathsf{newMsg}(m) \wedge D = \mathit{domain}(\mathit{Msg}, s) \cup \{m\} \vee \\
&\quad (\exists m, \varepsilon.\ \Phi_1(a, m, \varepsilon) \\
&\qquad \wedge ((\mathit{type}(m) = \mathsf{multicastDest} \wedge D = \mathit{domain}(\mathit{Msg}, s) \backslash \{m, \mathit{srcMsg}(m)\}) \\
&\qquad\quad \vee (\mathit{type}(m) \neq \mathsf{multicastDest} \wedge D = \mathit{domain}(\mathit{Msg}, s) \backslash \{m\}))) \\
&\quad \vee (\exists m, \varepsilon.\ \Phi_2(a, m, \varepsilon) \\
&\qquad \wedge D = \mathit{domain}(\mathit{Msg}, s) \\
&\qquad\qquad \cup \{m' \mid d \in \mathit{Agents} \wedge \mathit{multicastCopy}(m', m, d) \\
&\qquad\qquad\qquad \wedge \mathit{ensemble}(\mathit{channel}(m), \mathit{src}(m), d)\}) \\
&\quad \vee ((\nexists m, \varepsilon.\ \Phi_1(a, m, \varepsilon) \vee \Phi_2(a, m, \varepsilon)) \wedge D = \mathit{domain}(\mathit{Msg}, s))
\end{aligned}
\tag{6.2}
$$

with

$$\Phi_1(a, m, \varepsilon) \equiv a = transferEnds(m, \varepsilon) \vee a = transferFails(m) \tag{6.3}$$

$$
\begin{aligned}
\Phi_2(a, m, \varepsilon) \equiv a = transferStarts(m, \varepsilon) \\
\wedge\ \mathit{type}(m) \in \{\mathsf{multicastSrc}, \mathsf{remoteSensorSrc}\}
\end{aligned}
\tag{6.4}
$$

In the definition above, the action newMsg actually represents a creation mechanism whose technical details are slightly different. This is due to the fact that when the message is created in an agent process, a new name for the message entity has to be created and returned to be used in the remainder of the procedure.

The predicate *multicastCopy* that is used in the definition above relates a source message sent on a multicast or remote sensor channel to a copy of this message for a given destination.

Definition 6.3 (Multicast copy)**.** Let m be a message that is transferred on a multicast or remote sensor channel. Then m', the copy of m that is transmitted to the destination d, is defined by the following predicate:

$$
\begin{aligned}
&\mathit{multicastCopy}(m', m, d) \equiv \mathit{channel}(m) = \mathit{channel}(m') \\
&\quad \wedge \mathit{type}(m') = \mathsf{multicastDest} \wedge \mathit{dest}(m') = d \wedge \mathit{src}(m) = \mathit{src}(m') \\
&\quad \wedge \mathit{srcMsg}(m') = m \\
&\quad \wedge \mathit{destRole}(m') = \mathit{oppositeRole}(\mathit{channel}(m), \mathit{srcRole}(m))
\end{aligned}
\tag{6.5}
$$

The life cycle model of a message is realized by two fluents that act as state flags.

Definition 6.4 (Message life cycle)**.** The state of a message is defined by two separate mutual exclusive boolean fluents *awaitingTransfer* and *transferring* that define whether it is waiting to be transferred or currently being transfered. The state *idle* in the diagrams above corresponds with both fluents being false. If *awaitingTransfer* is true and *transferring* is false, this encodes the state *preparing* from before. Finally, *transferring* being true represents the state of a message during the actual transfer process. This is realized by the following successor state axioms.

$$\begin{aligned} awaitingTransfer(m, do(a, s)) \equiv{}& a = \mathit{requestTransfer}(m) \\ & \vee ((\nexists \varepsilon.\ a = \mathit{transferStarts}(m, \varepsilon)) \wedge a \neq \mathit{transferFails}(m) \wedge \\ & \quad \mathit{awaitingTransfer}(m, s)) \end{aligned} \tag{6.6}$$

$$\begin{aligned} transferring(m, do(a, s)) \equiv{}& (\exists \varepsilon.\ a = \mathit{transferStarts}(m, \varepsilon)) \\ & \vee ((\nexists \varepsilon.\ a = \mathit{transferEnds}(m, \varepsilon)) \wedge a \neq \mathit{transferFails}(m) \wedge \\ & \quad \mathit{transferring}(m, s)) \end{aligned} \tag{6.7}$$

The action *requestTransfer* is the entry point with witch agents initiate the information transfer. Therefore, it can only be executed when the message is in the *idle* state.

$$\begin{aligned} Poss(\mathit{requestTransfer}(m), s) \equiv{}& \neg \mathit{awaitingTransfer}(m, s) \\ & \wedge \neg \mathit{transferring}(m, s) \end{aligned} \tag{6.8}$$

The actual start of the information transfer is marked by the event *transferStarts*.

$$Poss(\mathit{transferStarts}(m, \varepsilon), s) \equiv \mathit{awaitingTransfer}(m, s) \tag{6.9}$$

A message has been transferred successfully when *transferEnds* occurs. However, source messages of multicast or remote sensor transmissions are not ended explicitly this way but are removed when all of their multicast copies have arrived or failed (cf. Definition 6.2).

$$\begin{aligned} Poss(\mathit{transferEnds}(m, \varepsilon), s) \equiv{}& transferring(m, s) \\ \wedge\ \mathit{type}(m) \notin{}& \{multicastSrc, remoteSensorSrc\} \end{aligned} \tag{6.10}$$

An information transfer can fail at any time after the transfer has been requested. For multicast and remote sensor transfers, the same argument holds as for *transferEnds*.

$$\begin{aligned} & \mathit{Poss}(\mathit{transferFails}(m), s) \equiv (\mathit{awaitingTransfer}(m, s) \\ & \lor \mathit{transferring}(m, s)) \land \ \mathit{type}(m) \notin \{\mathit{multicastSrc}, \mathit{remoteSensorSrc}\} \end{aligned} \tag{6.11}$$

Besides the state of a message, its content has to be encoded in the situation calculus, too. For that, it is necessary to distinguish between channel-based information transfer processes and sensors.

Definition 6.5 (Channel-based message transmission content)**.** The content of active messages that are *channel-based* (including remote sensors) is defined by the fluent C_{trans} whose content is set when a *transferStarts* event for the message occurs. For remote sensors, the content is received from the corresponding source sensor, i.e. from the fluent that is referenced by the *locSen* property of the message. For directly sent messages, the content is received from the message constant C_{out} that is set during a `Send` statement (see Section 6.3.2).

$$\begin{aligned} \forall m, s.\ & \mathit{domain}(\mathit{Msg}, s) \implies C_{trans}(m, \mathit{do}(a, s)) = y \equiv \\ & \big(\exists \varepsilon. a = \mathit{transferStarts}(m, \varepsilon) \land (\\ & \quad (\mathit{type}(m) = \mathsf{remoteSensorSrc} \\ & \quad\quad \land \mathit{locSen}(m) = f_l \\ & \quad\quad \land y = f_l(\mathit{src}(m), s) + \varepsilon) \\ & \quad \lor (\mathit{type}(m) \neq \mathsf{remoteSensorSrc} \\ & \quad\quad \land y = C_{out}(m, s) + \varepsilon))\big) \\ & \lor \big(\nexists \varepsilon. a = \mathit{transferStarts}(m, \varepsilon) \\ & \quad\quad \land y = C_{trans}(m, s)\big) \end{aligned} \tag{6.12}$$

For multicast destination copies, it holds that their content is always equal to the content of the original message. Deviations that are inflicted on individual message paths are aggregated when the message arrives at the destination (cf. Definition 6.6).

$$\begin{aligned} \forall m. \forall s.\ & \mathit{type}(m) = \mathsf{multicastDest} \land m \in \mathit{domain}(\mathit{Msg}, s) \implies \\ & C_{trans}(m, s) = C_{trans}(\mathit{srcMsg}(m), s) \end{aligned} \tag{6.13}$$

At the receiving side of a channel-based information transfer process, messages arrive at queues from which they can be extracted in agent processes using the `Receive` statement (see Definition 6.12).

Definition 6.6 (Incoming message queue)**.** Let c be the channel on which a message m is transferred, then for each agent, an incoming message queue for the channel c is defined by the following axiom:

$$\begin{aligned}
&\mathit{queue}_{\mathit{in}}(c, \mathit{do}(a, s)) = y \equiv \\
&\quad (\exists m, \varepsilon.\ a = \mathit{transferEnds}(m, \varepsilon) \wedge \mathit{channel}(m) = c \\
&\qquad \wedge y = \{\mathtt{msg}(\mathit{src}(m), \mathit{srcRole}(m), \mathit{dest}(m), \mathit{destRole}(m), \\
&\qquad\qquad\qquad \mathit{time}(m, s), C_{\mathit{trans}}(m, s) + \varepsilon)\} \cup \mathit{queue}_{\mathit{in}}(c, s) \\
&\quad \vee ((\exists \mathit{ag}, r.\, a = \mathit{cleanQueue}(\mathit{ag}, c, r)) \\
&\qquad \wedge y = \big\{e \mid e \in \mathit{queue}_{\mathit{in}}(c, s) \wedge \mathit{dest}(e) \neq \mathit{ag} \wedge \mathit{destRole}(e) \neq r\big\}) \\
&\quad \vee ((\forall m, \varepsilon.\ a \neq \mathit{transferEnds}(m, \varepsilon)) \wedge (\forall \mathit{ag}, r.\, a \neq \mathit{cleanQueue}(\mathit{ag}, c, r)) \\
&\qquad \wedge y = \mathit{queue}_{\mathit{in}}(c, s))
\end{aligned} \tag{6.14}$$

Here, the received content of the message including the error term is given by $C_{\mathit{trans}}(m, s) + \varepsilon$. This received value and other relevant information of the message, namely source, destination, the roles of sender and receiver, and a time stamp, are encapsulated in a term with the functor `msg` and appended to the message queue. As a convenience for referring to the parts of these terms, a dot notation with the structure *msg.«field»*is used below, i.e. *msg.src*, *msg.srcRole*, *msg.dest*, and so on.

The action *cleanQueue* is performed by the receiving agent as part of the `Receive` statement (see Definition 6.12) to remove already received messages. As one would expect, *cleanQueue* is idempotent and can be performed at any time, i.e.

$$\mathit{Poss}(\mathit{cleanQueue}(\mathit{ag}, c, r), s) \equiv \top \tag{6.15}$$

For sensor-based information transfer processes, receiving agents access a fluent that acts as a local view which contains the most recent sensor value.

Definition 6.7 (Local sensor view)**.** Let *sen* be the name of a local (direct) sensor. Then the model contains a corresponding fluent with the same name that stores the result of the last measurement as defined below. The notation «sen» is used below as a placeholder for the actual sensor name.

$$\begin{aligned}\text{«sen»}(ag, do(a,s)) = y \equiv&\\ &(\exists m, \varepsilon.\ \Phi(ag,a,m,\varepsilon) \wedge y = sen_{trans}(m,s) + \varepsilon)\\ &\vee\ ((\nexists m, \varepsilon.\Phi(ag,a,m,\varepsilon)) \wedge y = \text{«sen»}(ag,s))\end{aligned} \tag{6.16}$$

where

$$\begin{aligned}\Phi(ag,a,m,\varepsilon) \equiv& a = \mathit{transferEnds}(m,\varepsilon) \wedge \mathit{sensor}(m) = \text{«sen»}\\ &\wedge\ src(m) = ag\end{aligned} \tag{6.17}$$

The fluent sen_{trans} that appeared in 6.16 stores the information that is actually transferred during sensing. Its value is a snapshot of the source fluent content at the time when the transfer starts.

Definition 6.8 (Transmitted local sensor data)**.** The information that is transferred by a local sensor is stored in the fluent sen_{trans}.

$$\begin{aligned}\forall m, s.\ m \in domain(Msg, s) \implies\ & sen_{trans}(m, do(a,s)) = y \equiv\\ &(\exists \varepsilon.\Phi(m,a,\varepsilon) \wedge f = srcFluent(m)\\ &\quad \wedge y = f(src(m), s) + \varepsilon)\\ &\vee ((\nexists \varepsilon.\Phi(m,a,\varepsilon))\ \wedge\ y = sen_{trans}(m,s))\end{aligned} \tag{6.18}$$

where

$$\Phi(m,a,\varepsilon) \equiv a = transferStarts(m,\varepsilon) \wedge type(m) = \mathsf{sensor} \tag{6.19}$$

To achieve a unified transparent view on direct and remote sensing, the most recent values that are acquired by remote sensors are made accessible through a fluent that can be treated in the same way as the fluent view of local sensors.

Definition 6.9 (Remote sensor view)**.** Let «rsen» be the name of a remote sensor and let a_s be an agent that has a type which is eligible as a source for «rsen». Then a fluent with the name «rsen» exists that contains the most recently received value. This fluent is defined by the following SSA:

$$
\begin{aligned}
&\text{«rsen»}(a_d, a_s, do(a, s)) = y \equiv \\
&\qquad (a = \mathit{updateRemoteSensor}(a_d, \text{«rsen»}) \wedge \\
&\qquad\quad \exists \mathit{msg}.\ (\mathit{msg} \in \mathit{queue}_{in}(\text{«rsen»}, s) \\
&\qquad\quad \wedge\ \mathit{msg.dest} = a_d \wedge \mathit{msg.src} = a_s \wedge y = \mathit{msg.content} \\
&\qquad\quad \wedge\ (\nexists \mathit{msg}'.\ \mathit{msg}' \in \mathit{queue}_{in}(\text{«rsen»}, s) \wedge \mathit{msg}'.\mathit{time} > \mathit{msg.time})) \\
&\qquad \vee (a \neq \mathit{updateRemoteSensor}(a_d, \text{«rsen»}) \wedge y = \text{«rsen»}(a_d, a_s, s))
\end{aligned}
\tag{6.20}
$$

Here, *msg* and *msg′* are message terms as specified in Definition 6.6. The functions *src* and *dest* are used in this context to extract the source and the destination from this term. Similarly, *content* accesses the content part of the message term and *tstamp* the message's time stamp. The action *updateRemoteSensor* is called automatically by background processes of the agents that own remote sensors (see Section 6.3.3).

6.3 Information Transfer in SALMA Models

In order to turn the generic information transfer model into a practical solution for modeling distributed cyber-physical systems, SALMA provides high-level constructs that reflect the way a modeler normally thinks about information transfer processes in a CPS. These constructs can be seen as macros that are internally mapped to situation calculus axioms, agent process fragments, and probability distributions. Altogether, this creates an instantiation of the generic schema from Section 6.2 that integrates seamlessly with the rest of the model. In the remainder of this section, the most important elements of this high-level language are introduced, and their integration into the general simulation semantics of SALMA and the information transfer model from Section 6.2 are explained.

6.3.1 Connector Declaration Macros

SALMA's high-level language support for communication and generalized sensing spans across several sections of the model. First, all *connectors*, i.e. local sensors, remote sensors, and channels are declared in the domain model. This is done with the Prolog predicates `channel`, `sensor`, and `remoteSensor`, that are shown in Figure 6.9.

For `channel`, the modeler specifies a name for the channel, two *roles* with associated agent sorts, and the channel's *mode*, i.e. whether it is a unicast or

```
channel(«name», «role1»:«sort1», «role2»:«sort2», «mode»).
sensor(«name», «ownerSort», «srcFluent»).
remoteSensor(«name», «ownerSort», «localSensor»,
          «localSensorOwnerSort»).
```

Figure 6.9: Connector declaration predicates.

```
channel(assignment, veh:vehicle, sam:plcssam, unicast).
channel(reservation, veh:vehicle, plcs:plcs, unicast).
sensor(freeSlotsL, plcs, freeSlots).
remoteSensor(freeSlotsR, sam, freeSlotsL, plcs).
```

Figure 6.10: Connector declarations in the e-mobility example.

a multicast channel. All channels are bi-directional and the roles are used to distinguish message queues.

The `sensor` declaration defines the name of the local sensor, the sort of agents that own this sensor, and the fluent that represents the actual information source. This fluent is supposed to be qualified solely by the owning agent, i.e. it must be a function of the form $Agent \times Situation \rightarrow T$ with T being an arbitrary type. Similarly, the `remoteSensor` declaration also establishes a sensor for the owning agent type. However, instead of connecting it to a fluent, it defines a link to a local sensor that is owned by another agent. Both local and remote sensor declarations add fluents of the same name to the model that provide a current view on the acquired information (see Section 6.2.2). Additionally, a time-stamp fluent is installed for each sensor that records the time of the latest measurement or remote data retrieval, respectively. Finally, the declarations for both local and remote sensors are used to automatically install background processes that hide the sensing infrastructure (see Section 6.3.3).

An example for the use of the predicates described above can be found in Figure 6.10 that contains all connector declarations from the e-mobility example.

Here, `assignment` is defined to be a channel over which agents of the sort `vehicle` can communicate directly with agents of the sort `plcssam` in order to request and receive a PLCS assignment. The other channel `reservation` is used by vehicles to request slot reservations from PLCS agents and by the latter to acknowledge or deny these requests. The sensors of type `freeSlotsL` allow PLCS agents to count the current number of free slots at their station, i.e. access the fluent `freeSlots`. This information is propagated to the SAM via *remote sensors* of type `freeSlotsR` that effectively install channels and

```
pprocreq = Procedure([
      Receive("assignment", "sam", "assignment_requests"),
      Assign(assignments, processRequests),
      Iterate(assignments, [v, p],
            Send("assignment", Term("aresp", p), "sam", v, "veh"))])
 p1 = TriggeredProcess(pprocreq,
            "message_available", [SELF, "assignment", "sam"])
```

Figure 6.11: Assignment request processing procedure in the e-mobility example.

periodic background processes at each SAM and PLCS agent which transmit and receive the content of `freeSlotsL`, respectively.

6.3.2 Specialized Process Elements For Information Transfer

With the necessary declarations in place, the communication and sensing infrastructure can be used in agent processes by means of several special statements of the SALMA process definition language. As an example, the process `p1` shown in Figure 6.11 is installed on PLCSSAM agents in the e-mobility example. It handles incoming requests from vehicles, calculates optimal assignments, and sends them back to the vehicles.

The process `p1` is executed when messages are available at the SAM's incoming message queue of the `assignment` channel. First, all available assignment requests are retrieved from the queue with a call to `Receive` which stores a message list in the variable `req`. The actual assignment selection logic is integrated by means of an external Python function `processRequests`, which is not presented here for the sake of brevity. Through the `Assign` statement, the function is called with the received request list as a parameter and the function's result is stored in the variable `assignments`. One of the most important inputs for this optimization is the number of free slots at each PLCS. This information is made available by the remote sensor `freeSlotsR` from above that transparently gathers occupancy information from all PLCS (see Section 6.3.3). The result of `processRequests`, stored in `assignments`, is a list of tuples that assign each requesting vehicle to a PLCS. The agent process iterates over this list and sends to each vehicle `v` in that list a response term (functor `aresp`) that contains the PLCS identifier which is stored in the iteration variable `p`.

Based on the definitions from Section 3.6 and Section 6.2, the process elements used above can be defined accurately within the context of SALMA's simulation semantics and the situation calculus for information transfer. The most basic case is sending messages on a unicast channel like `assignment` from the example above.

Definition 6.10 (Unicast send)**.** Let a_s be an agent and m a fresh message. Also, let c, msg, $dest$, r_s, and r_d be terms that can be evaluated at the current simulation state to a unicast channel, a viable message term, an agent, and roles defined for channel c, respectively. Furthermore, let the value of $spec(m)$ be a tuple term that encodes the content of the message meta-data defined in Definition 6.1. Then, the following rule defines the interpretation of the `Send` statement:

$$\begin{aligned}&\langle\{(pid, a_s, \mathbf{Send}(\mathbf{c}, \mathbf{msg}, \mathbf{r_s}, \mathbf{dst}, \mathbf{r_d}) \circ \sigma, \eta)\} \cup P_{run},\\ &\quad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle\\ &\longrightarrow \langle\{(pid, a_s, Act(requestTransfer(m)) \circ \sigma, \eta)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle},\\ &\qquad Act, Evt, \mathcal{F}'\rangle\end{aligned}$$

$$\begin{aligned}\text{where } \mathcal{F}' = \mathcal{F}[&domain(Msg) \mapsto [\![domain(Msg)]\!]_{\mathcal{F}} \cup \{m\},\\ &spec(m) \mapsto \langle [\![c]\!]_{\mathcal{F},\eta}, a_s, [\![r_s]\!]_{\mathcal{F},\eta}, [\![dst]\!]_{\mathcal{F},\eta}, [\![r_d]\!]_{\mathcal{F},\eta}, [\![msg]\!]_{\mathcal{F},\eta}\rangle]\end{aligned}$$

The result of the rule above, i.e. the direct interpretation of the `Send` statement, does not yet capture the state that will eventually be reached in the simulation step in which the message is sent. Instead, this resulting state has to be derived using the rules from the simulation semantics of Section 3.6 and the axioms from Section 6.2.5. For the `Send` statement, the important aspect is that the state of the created message eventually ends up in the *preparing* state ($awaitingTransfer(m) = \top$) within the same simulation step.

Lemma 6.1. *Let $A \longrightarrow^n B$ denote the proposition that B results from A through n applications of semantic rules. Then, with m, c, msg, r_s, dst, r_d, and $\mathcal{F}'$ defined as in Definition 6.10, the following holds:*

$$\begin{aligned}&\exists n \textit{ so that}\\ &\langle\{(pid, a_s, \mathbf{Send}(\mathbf{c}, \mathbf{msg}, \mathbf{r_s}, \mathbf{dst}, \mathbf{r_d}) \circ \sigma, \eta)\} \cup P_{run},\\ &\quad P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle\\ &\longrightarrow^n \langle\{(pid, a_s, \sigma, \eta)\} \cup P^n_{run}, \emptyset, P^n_{wait}, P^n_{idle}, Act^n, Evt^n, \mathcal{F}^n\rangle\end{aligned}$$

$$\textit{and} \qquad [\![awaitingTransfer(m)]\!]_{\mathcal{F}^n} = \top \wedge [\![time]\!]_{\mathcal{F}^n} = [\![time]\!]_{\mathcal{F}}$$

Proof. First, the right hand side of the rule in Definition 6.10 represents the state directly after unfolding the `Send` statement. Let this state be denoted by St. At some step r after this unfolding, the simulation engine will have processed the `Act` statement, which yields a state St^r:

$$\begin{aligned}
&\exists r \in \mathbb{N}_0.\\
&\langle\{(pid, a_s, Act(\mathit{requestTransfer}(m)) \circ \sigma, \eta)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle},\\
&\qquad Act, Evt, \mathcal{F}'\rangle \longrightarrow^r St^{r-1}\\
&\overset{\text{Def.3.23}}{\longrightarrow} \langle P^r_{run}, P^r_{act} \cup \{(pid, a_s, \sigma, \eta)\}, P^r_{wait}, P^r_{idle}, Act^r \cup \{\mathit{requestTransfer}(m)\},\\
&\quad Evt, \mathcal{F}'\rangle = St^r
\end{aligned}$$

In the derivation above, the fluent database in state Sr^r is the same as in St. This can be seen from the premises of the rules in Section 3.6, which reveal that as long as there are still running processes, the system is neither able to perform a progression step nor a time advance or a process activation.

Before the action $\mathit{requestTransfer}(m)$ is eventually applied in a progression step, there can be several transitions, including other progression steps. Eventually, in some step s after St^r, the system will be in a state St^{r+s} where it is just about to progress the action $\mathit{requestTransfer}(m)$.

$$\begin{aligned}
\exists s \in \mathbb{N}_0.\ St^r \longrightarrow^s \langle \emptyset, P^{r+s}_{act} \cup \{(pid, a_s, \sigma, \eta)\}, P^{r+s}_{wait}, P^{r+s}_{idle},\\
Act^{r+s} \cup \{\mathit{requestTransfer}(m)\}, Evt^s, \mathcal{F}^s\rangle = St^{r+s}
\end{aligned}$$

Since the entity m has been created during the `Send` statement, it is only known in the same process. Therefore, the state of m cannot have been altered yet and thus $[\![awaitingTransfer(m)]\!]_{\mathcal{F}^s} = \bot$. With (6.8) from Definition 6.4, this means $[\![poss(requestTransfer(m))]\!]_{\mathcal{F}^s} = \top$. Hence, the progression can be performed, i.e.

$$\begin{aligned}
St^{r+s} \overset{\text{Def.3.24}}{\longrightarrow} &\langle \emptyset, P^{r+s}_{act} \cup \{(pid, a_s, \sigma, \eta)\}, P^{r+s}_{wait}, P^{r+s}_{idle}, Act^{r+s}, Evt^s, \mathcal{F}^{s+1}\rangle\\
&= St^{r+s+1}
\end{aligned}$$

where

$$\mathcal{F}^{s+1} = \mathsf{progress}(\mathcal{F}^s, \mathit{requestTransfer}(m))$$

With (6.6) from Definition 3.24, it holds that

$$[\![awaitingTransfer(m)]\!]_{\mathcal{F}^{s+1}} = \top$$

The system can proceed with performing progression steps or other transitions until all actions have been applied and the sending process is re-activated:

$$\exists u \in \mathbb{N}_0 . St^{r+s+1} \longrightarrow^u \langle \emptyset, P_{act}^{r+s+u} \cup \{(pid, a_s, \sigma, \eta)\}, P_{wait}^{r+s}, P_{idle}^{r+s}, \emptyset, Evt^{s+u}, \mathcal{F}^{s+1+u} \rangle$$

$$\overset{\text{Def.3.25}}{\longrightarrow} \langle P_{run}^{r+u} \cup \{(pid, a_s, \sigma, \eta)\}, \emptyset, P_{wait}^{r+s}, P_{idle}^{r+s}, \emptyset, Evt^{s+u}, \mathcal{F}^{s+1+u} \rangle$$

Since the message entity m was created within the process pid, no other process has been able to alter the state of m. Additionally, since the state of m is idle, the *poss* axioms in Definition 6.4 forbid that any information transfer event is scheduled for m. Therefore, $[\![awaitingTransfer(m)]\!]_{\mathcal{F}^{s+1+u}} = \top$. Furthermore, no time advance was possible throughout the transitions described above. Therefore, $[\![time]\!]_{\mathcal{F}^{s+1+u}} = [\![time]\!]_{\mathcal{F}}$. With $n = r + s + u$, this proves the lemma. □

As described in Section 6.2.2, the model for multicast channel-based communication differs from the unicast case mostly in the transmission phase. In fact, sending a multicast message merely means to leave out the destination.

Definition 6.11 (Multicast send)**.** Let a_s, m, msg, and r_s be defined as in Definition 6.10. However, let c now refer to a *multicast channel.* Then

$$\langle \{(a_s, \mathbf{Send(c, msg, r_s)} \circ \sigma, \eta)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F} \rangle \longrightarrow$$
$$\langle \{(a_s, Act(requestTransfer(m)) \circ \sigma, \eta)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}' \rangle$$

$$\text{where } \mathcal{F}' = \mathcal{F}[domain(Msg) \mapsto [\![domain(Msg)]\!]_S \cup \{m\}, spec(m) \mapsto \langle [\![c]\!]_{\mathcal{F},\eta}, mcs, a_s, [\![r_s]\!]_{\mathcal{F},\eta}, [\![msg]\!]_{\mathcal{F},\eta} \rangle]$$

With an identical argumentation as above, it can be shown that the multicast message will be eventually awaiting transferral within the same simulation step.

Lemma 6.2. *With the same notation as in Lemma 6.1, it holds that*

$$\exists n \text{ so that}$$
$$\langle \{(a_s, \mathbf{Send(c, msg, r_s)} \circ \sigma, \eta)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F} \rangle$$
$$\longrightarrow^n \langle \{(a_s, \sigma, \eta)\} \cup P_{run}^n, P_{act}^n, P_{wait}^n, P_{idle}^n, Act^n, Evt^n, \mathcal{F}^n \rangle$$

$$\textit{where} \quad [\![awaitingTransfer(m)]\!]_{\mathcal{F}^n} = \top \wedge [\![time]\!]_{\mathcal{F}^n} = [\![time]\!]_{\mathcal{F}}$$

Proof. The proof works almost exactly like in Lemma 6.1 and is therefore not repeated. □

Agents that are receiving messages from unicast or multicast channels must select those messages from the channel's message queue that are sent to the receiving agent and that match the requested destination role. In fact, the `Receive` statement removes all matching messages from the queue and stores the resulting set in a variable.

Definition 6.12 (Receive)**.** Let a, refer to an agent, and let c be a term that denotes a channel in the current evaluation context η. Furthermore, let the evaluation of r_d in η refer to a role of c. Finally, let v be a variable name that is unbound in η. Then,

$$\begin{aligned}&\langle\{(a, \mathbf{Receive}(\mathbf{c}, \mathbf{r_d}, \mathbf{v}) \circ \sigma, \eta)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle \longrightarrow\\ &\langle\{(a_s, Act(cleanQueue(a_s, c, r_s)) \circ \sigma, \eta')\} \cup P_{run}, P_{act}, P_{wait}, P_{idle},\\ &\qquad Act, Evt, \mathcal{F}\rangle\end{aligned}$$

$$\begin{aligned}\text{where } \mathcal{M} &= \{mt \mid mt \in [\![queue_{in}(c)]\!]_{\mathcal{F},\eta} \wedge\\ &\qquad\qquad \exists a_s \exists r_s \exists \vartheta.\, mt = \langle a_s, r_s, a, [\![r_d]\!]_{\mathcal{F},\eta}, \vartheta\rangle\}\\ \eta' &= \eta[v \mapsto \mathcal{M}]\end{aligned}$$

Besides transferring the received messages to the current environment, the effect of the `Receive` statement is the removal of received messages from the input queue. This is shown in the following lemma.

Lemma 6.3. *With the notation and definitions from above, it holds that*

$$\begin{aligned}&\exists n \text{ so that}\\ &\langle\{(a, \mathbf{Receive}(\mathbf{c}, \mathbf{r_d}, \mathbf{v}) \circ \sigma, \eta)\} \cup P_{run}, P_{act}, P_{wait}, P_{idle}, Act, Evt, \mathcal{F}\rangle\\ &\quad\longrightarrow^n \langle\{(a_s, \sigma, \eta^n)\} \cup P^n_{run}, P^n_{act}, P^n_{wait}, P^n_{idle}, Act^n, Evt^n, \mathcal{F}^n\rangle\end{aligned}$$

$$\textit{where} \qquad [\![queue_{in}([\![c]\!]_{\mathcal{F},\eta})]\!]_{\mathcal{F}^n} \cap \mathcal{M} = \emptyset$$

Proof. At first, it can be shown with a similar proof as in Lemma 6.1 that there is an intermediate step before n, in which $cleanQueue(a_s, c, r_s)$ is applied in progression. Let $\mathcal{F}'$ denote the fluent database just before and $\mathcal{F}''$ the fluent database right after this progression step. The contents of the queues are not known at this point since they could have been altered by other information transfer processes. However, due to the successor state axiom for $queue_{in}$ from Definition 6.6, it holds that $[\![queue_{in}(c)]\!]_{\mathcal{F}''} = \big\{e \mid e \in [\![queue_{in}(c)]\!]_{\mathcal{F}'} \wedge$

$dest(e) \neq ag \wedge destRole(e) \neq r\}$. Since $dest(e) = ag \wedge destRole(e) = r$ corresponds to the condition in the definition of $\mathcal{M}$ in Definition 6.12, all messages in $\mathcal{M}$ must either be deleted in this step or must have been deleted already by another call of *cleanQueue*. This means that $[\![queue_{in}(c)]\!]_{\mathcal{F}''} \cap \mathcal{M} = \emptyset$. Afterwards, until the receiving process is reactivated in step n, the queue content could change due to other occurrences of *transferEnds* or *cleanQueue*. However, the messages in $\mathcal{M}$ cannot be added again once they are deleted. Therefore, it still holds that $[\![queue_{in}([\![c]\!]_{\mathcal{F},\eta})]\!]_{\mathcal{F}^n} \cap \mathcal{M} = \emptyset$. This concludes the proof. □

The three statements presented above are actively used within agent processes to realize the respective agent's role in message-based communication processes. Additionally, there are three further statements that are necessary to implement the information transfer processes described in Section 6.2.2:

- `Sense(c)` initiates local sensing on sensor `c` as shown in Figure 6.5.
- `TransmitRemoteSensorReading(rs)` initiates the transmission of sensor data according to the specification of remote sensor `rs` along the lines of Figure 6.6.
- `UpdateRemoteSensor(rs)` processes the received data on remote sensor `rs` and updates the remote sensor view accordingly (see Figure 6.7).

The semantical interpretations of the statements mentioned above are very similar to those presented in the Definitions 6.10, 6.11, and 6.12 and are therefore not presented in detail. Besides that, these elements would very rarely be used explicitly within agent process definitions. Instead, they are used by implicit background processes as described in Section 6.3.3.

6.3.3 Transparent Sensing Infrastructure

With the process elements introduced in the end of the last section, it would be possible to treat local and remote sensing as explicit agent tasks in the same manner as communication with other agents. However, this would lead to process definitions that mix core agent logic with infrastructure elements. In fact, it is closer to common realistic agent architectures to place sensing facilities into a separate layer and make the sensed information available in a transparent way. To achieve that, the SALMA framework automatically installs some *background processes*:

For each *local sensor*, an update process is installed at each agent that owns such a sensor. This process repeatedly executes `Sense`, i.e. initiates the sensing process. The actual *sensor view fluent* is updated after a delay that depends on the probability distributions that are set up for the specific sensor. By default, the update process is set up as *periodic* with a fixed period that is set up once in the simulation setup. However, other scheduling schemes are

```
...
If("freeSlotsL(self) > 0", [
    Act("add_reservation", [SELF, vehicle]),
    Send("reservation", Term("rresp", SELF, True), "plcs", vehicle, "veh")],
    [ # ELSE
    Send("reservation", Term("rresp", SELF, False), "plcs", vehicle, "veh")])
...
```

Figure 6.12: Excerpt from reservation processing process of PLCS agents.

also possible, e.g. to reflect adaptive sensing strategies that try to optimize energy consumption.

Similarly, a *transmission process* is installed for each remote sensor at each agent of the *remote data source type.* This process transmits the most recent value of the configured local sensor to the remote sensor data sinks in the ensemble. By default, the transmission process is configured as periodic with a fixed period that would normally be set significantly longer than for the monitored local sensor. In addition to the transmission process, a reception process is created for each *data sink agent*, i.e. each owner of a remote sensor. Here, received remote sensor data is processed as described in Figure 6.7. In contrast to the other background processes, the reception handler is installed as a triggered process that is executed as soon as new data becomes available.

With the implicit background processes in place, the modeler can access the most recent values of local and remote sensors in the same way as directly available fluents. For instance, the code fragment in Figure 6.12 is taken from the e-mobility example and shows an excerpt from the PLCS agent process that handles reservation requests by vehicles. Here, the condition `freeSlotsL(self) > 0` is used to test whether free slots are available at a PLCS, and hence to decide whether to accept or reject a request.

6.4 Statistical Model Checking for Information Transfer Processes

It was already shown in Chapter 4 that the SALMA-PSL makes it possible to refer directly to entities and agents, and to reason about their properties and relations. Since messages and connectors are also represented as entities, this means that all elements of communication and sensing processes can be examined with fine granularity. Additionally, the SALMA-PSL provides a set of specialized functions and predicates that can be used together with the fluents defined in Section 6.2.2 and Section 6.2.5 to create an intuitive way for reasoning about the content of the information transfers, e.g.:

```
P1 = forall(v:vehicle, implies(messageSent(v, assignment, ?, ?, ?, ?),
         eventually(100, currentTargetPLCS(v) \= none)))
P2 = forall(s:plcssam, forall(p:plcs, age(freeSlotsR, s, p) =< 10))
P3 = forall(s:plcssam, forall(p:plcs, abs(freeSlotsR(s, p) - freeSlotsL(p)) =< 1))
P4 = forall(p:plcs, forall(v:vehicle, implies(
         occur(add_reservation(p, v)),
         eventually(10, messageSent(p, reservation, plcs, v, vehicle,
                             rresp(p, true) )))))
```

Figure 6.13: Example SALMA-PSL properties of information transfer processes.

- $messageSent(chan, src, r_s, dst, r_d, msg)$ is a predicate that is true if, in the current time step, a message with content msg has been sent from source agent src to destination dst on channel $chan$ with the given source and destination roles r_s an r_d.
- $messageAvailable(chan, dst, r_s)$ is a predicate that is true if the incoming message queue of agent dst for channel $chan$ contains at least one message that addresses dst with role r_s.
- $src(m)$, $dest(m)$, and $con(m)$ are functions that return the source, destination, and the connector of a given message.
- $age(sen, a, [a_r])$ is a function that returns the age of the most recent value for the local or remote sensor sen of agent a. If sen is a remote sensor, then age refers to the value transmitted by the remote agent a_r.

Examples for the use of the SALMA property specification language in the context of information transfer processes can be found in Figure 6.13. The invariant `P1` requires that when any vehicle agent sends an assignment request to the SAM, it will not take longer than 100 time units until a target PLCS has been set. The question marks in the predicate `messageSent` serve as wild-card arguments for pattern matching, which achieve here that the recipient of the message, the involved roles, and the content of the message are ignored.

As other examples, `P2` and `P3` are invariants that define, for all measurements acquired by the remote sensor `freeSlotsR`, a maximum value age of 10 time units and a maximum deviation of 1 from the original sensor `freeSlotsL`.

Finally, property `P4` demonstrates how the *content* of a message can be used directly in SALMA-PSL expressions. The property, which refers to the example in Figure 6.12, states that every time a reservation is made by a PLCS agent (action `add_reservation`), a positive acknowledgment message must be sent within 10 time units. In order to test that, the content of the sent message has to be compared to the expected content according to Figure 6.12 (`rresp(p, true)`).

Altogether, while it was demonstrated before in this thesis that the ability to use a first first-order logic language greatly facilitates the expression of complex properties, it becomes apparent that these benefits are particularly valuable for properties that reason about the details of communication and sensor data propagation.

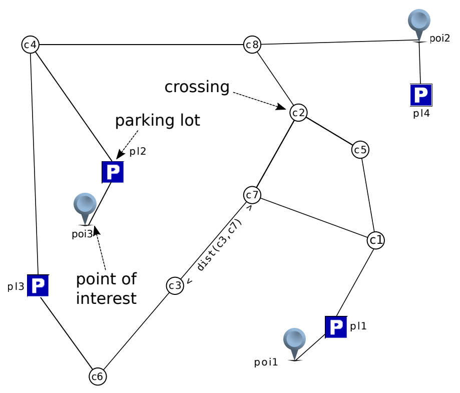
6.5 Experimental Evaluation

In order to test the presented approach and its integration in the SALMA toolkit, a reduced version of the scenario introduced in Section 6.1 was implemented. It contains only a simple mock-up version of the optimization mechanism but realizes the full communication structure.

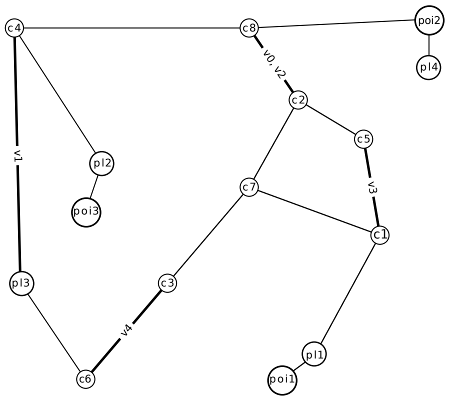
In the model that was used for the experiment, the map on which vehicles are moving around is represented by a weighted graph with three different node types: crossings, point of interests (POIs), and parking lots with charging stations (PLCS). The edges represent *roads* that lead from a start node to an end node. As usual, the lengths of these roads are represented by the weights in the graph. Figure 6.14a shows the undirected graph that was originally specified as a GraphML [BEH$^+$02] file and used to derive the map for the experiment. The undirected edges were translated to two directed edges in opposite direction with lengths that were derived from the geometrical information stored in the original file.

Vehicles are modeled as agents that move around the map to reach a certain point of interest (POI) that is randomly assigned to them at the beginning of the simulation. The vehicle agent then communicates with the super autonomous manager (SAM) and the PLCS agents according to the scheme in Figure 6.1. First it requests an assignment of a PLCS that is as close as possible to the vehicle's POI. In the current version, the SAM agent only uses a trivial optimization strategy that merely assigns the first PLCS with free slots. After the assignment has been made, the vehicle requests a reservation at the assigned PLCS. Once this reservation has been granted, the vehicle agent sets its target PLCS and calculates an optimal *route* to the assigned PLCS. To do this, a Python function was integrated as shown in Section 3.4.5 and the actual computation was performed using the graph algorithm library NetworkX [Net16].

The communication between vehicles, the SAM, and PLCS agents, is performed by agent processes like the ones shown in Figure 6.11 and Figure 6.12. In contrast, the actual vehicle movement is not performed by explicit agent processes but encoded directly in the situation calculus model of the simulation. There, the position of each vehicle is represented either by a current node or by the current road the vehicle is driving on. The current route followed



(a) Manually created weighted graph with different node types.



(b) Automatically generated visualization of one simulation step.

Figure 6.14: Graph-based map used for the e-mobility experiment.

by a vehicle is given as a list of roads that in sequence lead from the current position to the target PLCS. A more exact location on the road is actually not represented directly in the model. Instead, the road length is used as a factor for the delay with which events are scheduled that update make the vehicle move to the next route segment. This style of sparse simulation with pure event-scheduling is very well suited for the discussed example, in which the focus is set on the information transfer aspects rather than vehicle behavior or traffic.

The simulation of the e-mobility example can be run in three different modes: visualization, estimation, and hypothesis test. In visualization mode, the simulation is performed until all vehicles have arrived at their target PLCS. Meanwhile, the positions of all vehicles on the map are visualized in each step as annotations on the map. For instance, Figure 6.14b shows a step of a simulation run with five vehicles (`v0` to `v4`) where each vehicle is currently located on a road, which is shown by labeling all edges with the ids of the vehicles on the corresponding road. Additionally, textual information about the simulation state in each step is written to a log file. Figure 6.15 contains the output for the 102*nd* step of an e-mobility simulation run. Lines 2 to 4 show the actions and events that were performed in this step. It can be seen that *transferStarts* messages occurred for the messages 231 and 228, both without an error term. The messages that currently exist in the system are listed from line 5 to line 18. It turns out that both messages belong to local sensors of the type *freeSlotsL*, which is used by PLCS agents to “measure” the available capacity. Each message line in the log output contains the message specification and the *transferred content*, separated by a colon. In this case, the transferred content is set now for both sensor messages. In fact, the last four lines of the output each confirm that the transferred values coincide with

```
Step 102 (t = 116)
   Actions: [(’transferStarts’, (231, None)), (’enterNextRoad’, (’v4’,)),
            (’transferStarts’, (228, None)), (’arriveAtRoadEnd’, (’v3’,))]
      #227: msg(’freeSlotsL’, ’sensor’, ’pl4’, ()) : None
      #228: msg(’freeSlotsL’, ’sensor’, ’pl3’, ()) : 10
      #229: msg(’freeSlotsL’, ’sensor’, ’pl2’, ()) : None
      #231: msg(’freeSlotsL’, ’sensor’, ’pl1’, ()) : 10
   v4: r19(c3-c6=193) - [’r29(c6-pl3=170)’] - pl3 / None
   v0: c2 - [’r24(c2-c8=128)’, ’r6(c8-poi2=331)’, ’r32(poi2-pl4=60)’] - pl4 / None
   v2: r24(c2-c8=128) - [’r6(c8-poi2=331)’, ’r32(poi2-pl4=60)’] - pl4 / None
   v3: c5 - [’r26(c5-c1=124)’, ’r2(c1-pl1=193)’] - pl1 / None
   v1: pl3 - [’r10(pl3-c4=320)’, ’r16(c4-c8=431)’, ’r6(c8-poi2=331)’,
                  ’r32(poi2-pl4=60)’] - pl4 / None
   pl1:plcs: real = 10, local = 10, remote = 10
   pl2:plcs: real = 10, local = 10, remote = 10
   pl4:plcs: real = 10, local = 10, remote = 10
   pl3:plcs: real = 10, local = 10, remote = 10
```

Figure 6.15: Log output for one step of an e-mobility simulation run.

the true numbers of free slots of the PLCS *pl*1 and *pl*3, to which the messages belong. This conforms closely to the information transfer model for local sensing described in Figure 6.5.

The visualization and verbose logging of simulation steps can be used to do a qualitative evaluation of the model, i.e. to verify that the modeled system behaves in a plausible way that is consistent with intuitive expectations. For instance, it is easy to verify that the agents in fact exchange the intended message types, and that calculated routes actually lead to the intended destinations. Once the basic functionality of the simulation has been tested, the simulation can be executed in estimation or hypothesis testing mode. This means that multiple trials are executed and each simulation run will be stopped as soon as all registered goals are fulfilled or a registered invariant has been violated.

For the concrete experiment that is presented here, the invariant `P1` from Figure 6.13 was registered as a single property. As mentioned before, this property requires that each time a vehicle sends out an assignment request, it receives the response within a given time limit, which in turn sets the fluent `currentTargetPLCS` of the vehicle. The probability that this invariant is satisfied on a given simulation run depends mainly on the probability distributions for transmission delays and failures, and on the time limit that is specified for the `until` operator. As a first step, 5 vehicles were used and a normal distribution with a mean of 5 timesteps and a variance of 1 was set up for all message delays. Furthermore, the probability that a message transfer fails during the preparation phase was set to 0.1 and the probability that it fails during the transfer phase to 0.2. Using this fixed simulation setup, the experiment was performed iteratively while the time limit in the `until` operator of property `P1` was increased from 5 time units to 145 time units in steps of 5. For each of the 28 configurations, 50 simulation runs were performed. The

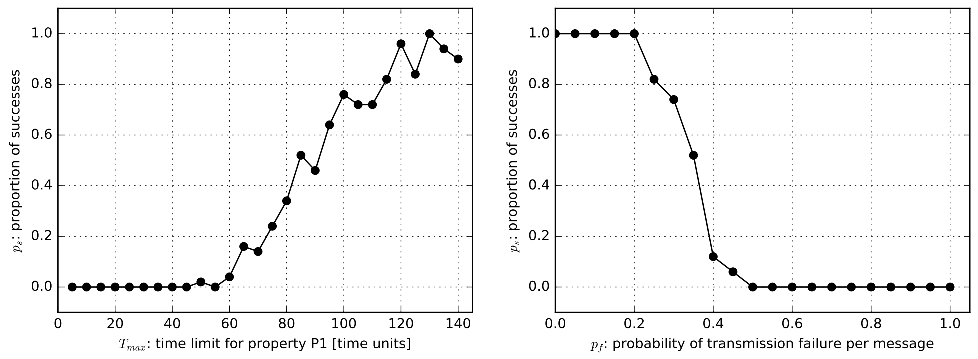


Figure 6.16: Proportion of successful simulation runs for varying time limits and transmission failure probabilities.

left graph in Figure 6.16 shows the proportion p_s of successful simulation runs for each different time limit. It can be seen that there is virtually no chance for success below 55 time units while on the other hand success seems to be almost certain for time limits over 145. However, these results clearly are only valid for the given distributions of transmission failures. To gain some insight into the effect of increasing failure rates, a second iterative experiment was performed whose results are shown in the right diagram of Figure 6.16. This time a fixed time limit of 110 time units was used while the failure probability for the message transfer phase p_f was increased from 0 to 1 in steps of 0.05. As before, 50 simulation runs were performed for each value of p_s. The results reveal that for the chosen time limit, failure probabilities below 0.2 appear to be unproblematic whereas failure probabilities of above 0.5 effectively prohibit success. The steepness of the decline between $p_f = 0.2$ and $p_f = 0.4$ is a little surprising and might be worth being analyzed further, although this is is beyond the scope of this section.

Another question that was examined is how strongly the execution times of the simulations depend on the model complexity. In particular, it can be expected that an increasing number of agents will lead to an increased number of transferred messages, which in turn increases the simulation complexity since messages are treated as entities. To reproduce this effect, the time limit was set to 500 so that every simulation run is guaranteed to succeed and the number of vehicles was increased from 1 to 30. For each vehicle number, batches of 20 simulation runs were performed and the execution times were recorded. The median values of these measured times are plotted in Figure 6.17. This curve reveals a dependence that is nearly linear, with a range between about $0.5s$ for one vehicle and about $31s$ for 30. At first sight this suggests that it would be unrealistic to extend the simulation to hundreds or even thousands of vehicles, which is not unusual for traffic simulation. Of course, supporting large numbers of agents is possible, but only if the workload is distributed by

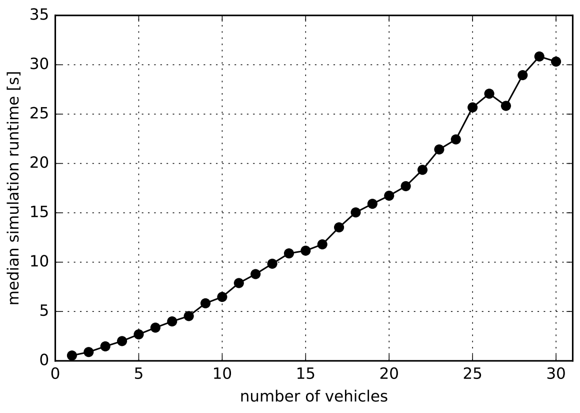


Figure 6.17: Mean simulation run times for varying vehicle numbers.

running simulations on many machines. However, even though emerging virtualization and cloud computing technologies make parallelization both simple and affordable, it is still obvious that SALMA will never be able to compete with specialized traffic simulation solutions.

On the other hand, for simulation experiments in which the focus is set on analyzing communication and sensing scenarios as in this chapter, it is questionable whether it is actually necessary to simulate that many vehicle agents with the same granularity as it was shown in the current example. Instead, it might be sufficient to choose one or a few representatives for those classes of agent behavior that differ with respect to how agents communicate or sense their environment. Other aspects of the system can often be modeled with a much higher level of abstraction with lower computational demand. For instance, in the e-mobility scenario of this example, it would be possible to add very simple agents to the map that simulate basic traffic effects. Again, SALMA intentionally leaves the choice of the right level of detail to the modeler in order to allow finding a suitable compromise between accuracy and efficiency.

It is clear that there are many parameters that could be varied in the simulation of the automatic parking lot assignment scenario, including PLCS capacity, and the probability distributions for message delays, etc. Furthermore, the Python function that realizes the PLCS assignment can be replaced to test different optimization schemes and their sensitivity to factors like delays or transmission errors. Such a detailed evaluation is clearly far beyond the scope of this thesis. However, the experiences gained during the development of the model and through experimentation show that proposed information transfer model is well applicable even for "real life" communication scenarios. In particular, the declarative high-level extensions of SALMA's modeling languages that were introduced in Section 6.3 have proven to be suitable for creating concise and readable models. For instance, the declarative part of the e-mobility example model that sets up communication and sensing infrastructure requires only about 30 lines in the style of the examples in Section 6.3.1.

6.6 Summary

This chapter presented a new logic-based approach for modeling channel-based communication, sensing, and other kinds of information transfer processes within cyber-physical multi-agent systems. High-level modeling language extensions were introduced, which can be integrated seamlessly into SALMA domain models, agent processes, or property specifications. These constructs provide a familiar viewpoint for modeling communication and sensing processes in cyber-physical systems and help to embrace their stochastic nature, like transmission delays and errors. At the same time, the information transfer model has a precise formal semantics based on the situation calculus, which means it can be integrated as a natural layer in any SALMA system model. A major advantage of this approach is that it allows fine-grained reasoning about the inner details of information transfer processes.

One of the main distinguishing features that make the presented model suitable for describing cyber-physical systems is its ability to integrate both local and remote sensing in agent processes in a unified and transparent way that hides the details of how the sensed data is propagated from its origin to the sensing agent. At the same time, the characteristics of the sensing processes can be controlled precisely by choosing appropriate probability distributions for the events that govern the message life cycle.

Accompanying the extensions for SALMA's system modeling languages, several specialized functions and predicates were added to SALMA's property specification language. They allow referring to all relevant properties of the communication and sensing processes in the simulated system model, like message payloads, queue contents, or time stamps that mark when messages are sent or received. Generally speaking, it can be seen that modeling information transfer is much facilitated by using a first-order logic language like the SALMA PSL.

The presented approach was tested during the development of a simulation model that was derived from the a case study of the EU research project ASCENS [2]. This example describes a system in which autonomous agents collaborate to find an optimal assignment of parking lots. The SALMA model that was developed for this scenario covers all aspects of information transfer that were discussed in this chapter, namely channel-based communication, local and remote sensing, and ensemble-based multicast messaging. Several simulation and statistical model checking experiments were conducted with this model and results were described in this chapter. The experience that was gained from developing the model and conducting the presented experiments shows that the approach offers great flexibility with respect to the level of

[2]`www.ascens-ist.eu`

detail and accuracy with which both the system model and corresponding requirements are formulated.

6.7 Related Work

The primary contribution of the extensions to the SALMA approach that were presented in this chapter lies in providing high-level constructs for modeling and distributed communication of agents with realistic semantics. Therefore, it makes sense to look into the related work from three main areas: (1) communication models for the situation calculus, (2) coordination languages for distributed agents, and (3) simulations of network communication in distributed systems.

From the perspective of the situation calculus, the transfer of information has traditionally been viewed from an epistemic perspective, i.e. as knowledge that agents gain through (communication) actions. Surprisingly enough, there are no systematic approaches that introduce in the frame of a situation calculus a communication model reflecting properties of real-life computer networks. In this respect, a partial approach is provided by [M$^+$95], where the epistemic model has been extended to model inter-agent communication by means of channels in a similar way as in the model described in this chapter. However, neither time nor stochastic effects are covered. In contrast to that, the approach presented in [Sch03] combines the epistemic model with time and concurrency and allows reasoning about time-related aspects like the age of measurements.

Unlike the approaches mentioned above, SALMA's method of modeling information transfer does not consider knowledge in the epistemic sense but leaves the interpretation of transferred information to the agent processes. While this perspective appears to be better suited in the particular context of cyber-physical systems, it would be possible to combine both views in a straight-forward way.

Another perspective is provided by coordination languages for distributed agents, which are typically not based on the situation calculus. Rather they provide their own process algebras to define the behavior of distributed agents. These approaches are primarily represented by SCEL [CCC$^+$14] and its indirect predecessor KLAIM [BBN$^+$03]. They feature communication based on structure knowledge exchange via tuple-spaces. The process algebra they use for specification of the behavior of agents is typically based on or at least inspired by π-calculus.

In their original form the coordination languages do not consider time and communication uncertainty, which makes them unfit for realistic modeling of network communication. However, extensions exist that feature these concerns forming a relatively large family of stochastic process algebras like TIPP [GHR93], PEPA [Hil96], EMPA [BG96, ACB10], stochastic π-calculus

[Pri95, CM13], StoKlaim [DNKL$^+$07] or a unifying framework by Nicola et al [NLLM13]. Targeting the framework of SCEL, which is arguably closest to the approach presented here, the ideas of stochastic process algebras are well integrated in [L$^+$14], where the authors introduce a stochastically timed process calculus that is centered around predicate-based communication. Similar to SALMA's information transfer model, the most detailed semantical variant they describe distinguishes between a preparation and a transmission phase and allows assigning separate probability distributions for delays and errors to each of them. However, since the semantics is based on continuous time Markov chains (CMTC), only exponential distributions can be used and delays or errors are effectively determined at the start of each phase. This can be too coarse-grained in very dynamic situations, e.g. when the movement of agents affects the likelihood of transmission errors too much to be neglected.

Targeting specifically the analysis of communication and processing in distributed systems, network simulators, e.g. OMNet++ [Var10] and ns-3 [HLR$^+$16] provide very accurate estimates. The simulators feature an agent-like approach, where the simulated network consists of a number of modules (representing end-devices and network components) mutually communicating by exchanging messages. These modules are triggered by a discrete simulator based on timing needed for message processing, communication latencies, etc. To achieve simulation of environment when agent mobility is involved, the network simulators can be integrated with mobility and traffic simulators, e.g. MATSim [HNA16] or Sumo [KEBB12].

Compared to SALMA, network simulators provide significantly more precise estimates in terms of network communication latencies. However, compared to the situation calculus and the high-level interaction patterns presented in this chapter, the low-level perspective of these network simulators make it harder to describe the system architecture and coordination structure in a concise way. Furthermore, there is no option of verification against properties formulated in temporal logic in simulators like OMNet++ or ns-3.

Chapter 7

Conclusion and Outlook

At the bottom line, it could be said that SALMA, the approach that is presented in this thesis, proposes yet a new solution for modeling, simulation, and model-based verification. More precisely, SALMA concentrates on statistical model-checking, a paradigm that aims to overcome the infamous state space explosion problem by using statistical sampling instead of exhaustive search through the state space. Although statistical model checking is a rather new concept, it is becoming more and more popular and there are now several mature tools that provide support for different combinations of modeling paradigms and statistical methods. This chapter summarizes the main points where SALMA differs from other simulation tools or statistical model checkers. Additionally, an outlook about possible improvements is given and some promising directions for future work are outlined.

7.1 Key Achievements of the Thesis

The most significant difference that SALMA makes is that it puts first order logic right into the center of its modeling approach for both the system itself and the properties that should be verified. For the system model, SALMA makes use of the situation calculus, an approach for modeling dynamic systems in first order logic that dates back several decades. Although the situation calculus has been used in the context of simulating agent-based systems, in particular as part of the programming language GoLog (see Section 2.2.4), SALMA appears to be the first approach that actually integrates it into a fully-fledged solution for discrete event simulation. As this thesis has shown, this paves the way for a very tight integration with the property specification language that SALMA uses, which is a first-order version of linear temporal logics (LTL). In fact, SALMA's support for first-order logic includes the free composition of formulas from user-defined signatures, including arbitrarily nested functions and a unification-based equality operator for terms. This makes SALMA stand out from other solutions for statistical model checking, whose property specification languages are basically propositional versions of LTL or CTL — except some limited additions like the support for quantifiers over arrays in UPPAAL (see Section 2.4).

Throughout the previous chapters, it was shown that the support for first-order logic is more than syntactical sugar. In fact, the tight integration of a logic-based system model and the property specification language creates a very high degree of flexibility that allows referring directly to all relevant facets of the system. This becomes particularly obvious in Chapter 6, which describes an extension of the basic modeling languages of SALMA that can be used to describe information transfer processes, i.e. communication between agents or using sensors to acquire information. One of the key aspects of the proposed model is that it allows capturing the stochastic nature of these processes, like occasional message failures or delays. In order to integrate these aspects in the simulated system model, they were formalized by means of a set of situation calculus axioms and several event types that govern the life-cycle of messages. Additionally, some convenient high-level primitives for communication and sensing were added to the language. The usability of the approach was demonstrated through a mid-sized example from the domain of e-mobility.

A full software stack for discrete event simulation and statistical model checking has been implemented by the author in the context of this thesis. In particular, this includes the development of efficient algorithms for the evaluation of formulas specified in SALMA's property specification language. This had to be done from scratch because SALMA's combination of logic programming and statistical model checking has not been described before. In particular, the first-order structure of the evaluated formulas together with the support for nested temporal operators require special care in order to make the evaluation efficient. The solution that was eventually developed during this thesis involves a combination of term rewriting, result caching, and an interval-based representation of evaluation states that often avoids unnecessary evaluation steps (see Chapter 5).

One key goal for the design of SALMA's modeling languages and the simulation framework is to give the modeler as much freedom as possible in choosing the right level of abstraction for each individual aspect of the model. To a large degree, this has been achieved by a close integration of SALMA's specialized modeling languages with the underlying general-purpose programming languages Python and Prolog. For instance, the agent behavior model for parking lot assignment example in Chapter 6 uses a popular graph algorithm library for calculating shortest paths through a map (see Section 6.5). The resulting route is then directly stored as a list in a fluent in the situation calculus model. There it can be analyzed and manipulated directly within axioms and property formulas using Prolog's extensive support for list and term processing. This ability to integrate details like transmitted data and to propagate it throughout all layers of the model can help significantly to avoid making forced abstractions that could conceal important effects and mechanisms of the system.

All components of SALMA's simulation and statistical model checking toolkit have been tested extensively by means of unit and integration tests. Additionally, the correctness and usability of the approach in its entirety has been validated with several complete examples, in particular the multi-robot example that was described in Chapter 3 and Chapter 4 and the parking lot assignment scenario from Chapter 6. The experience gained from developing these models and conducting experiments with them has shown that the chosen design of the modeling languages and the simulation framework strongly facilitates the creation of comprehensive, extensible, and maintainable models.

7.2 Possible Improvements and Extensions

During the development of the SALMA toolkit and by working with the examples, several points were identified where improvements of the approach would be particularly important. For most of the recognized problems, more or less concrete ideas about possible solutions emerged.

First of all, one of the most obvious problems with SALMA as an approach for discrete event simulation is the computational cost of updating the world state through the progression mechanism of the situation calculus. In fact, simulations that are developed with more low-level frameworks like Repast (see [Col03]) run much faster than SALMA simulations where in each step all *fluent instances* have to be updated by evaluating the corresponding successor state axioms. Although part of that performance drawback has to be accepted as price that has to be paid for the declarative modeling style SALMA offers, there is plenty of room for improvement. In particular, the current implementation of the progression mechanism evaluates the successor state axiom for all fluent instances and uses these axioms to decide whether a given action or event affects the value. However, it would often be possible to prune a significant part of the evaluation branches. For instance, the effect axioms (see Section 3.2.4) could be analyzed to determine whether or not any instance of a fluent might be affected by the action. Additionally, there are many actions or events that affect only one particular entity or agent that is included as an "identity" argument of the action term. In these cases, the set of fluent instances that have to be inspected could be pruned significantly. Besides these rather simple strategies, there are certainly many other more sophisticated criteria for pruning fluent instances. For example, the effects of actions or events are limited to a certain spatial vicinity. This could be exploited by a strategy that limits the selection of fluent instances based on their positions. Of course, in order to make this search efficient, a spatial index for entities and agents would have to be created using suitable spatial data structures like R-trees

[Gut84].

Although the performance of individual simulations can certainly be improved significantly by measures like those described above, it is inevitable for larger scenarios to employ parallelization. In fact, it was already mentioned before in this thesis that one of the most promising aspects of statistical model checking is that it can easily be scaled horizontally by just adding more machines that run simulations in parallel. Since these simulations are independent, this can be done without any change of the SALMA toolkit by simply aggregating the output of multiple separate simulation processes. However, in order to make the approach actually scalable to a large number of nodes, some kind of coordination infrastructure layer would have to be added that takes care of tasks like propagating simulation parameters, gathering results, or restarting crashed nodes. Due to the availability of technologies for realizing distributed systems that have emerged in the context of cloud computing and “Big Data”, realizing such services has almost become routine work. For instance, the Mesos platform [HKZ+11] provides a layer of abstraction over the resources of machines in a cluster that could be leveraged to schedule and coordinate distributed simulations in an almost completely transparent way.

Besides performance and scalability, there are several other technical aspects of the SALMA toolkit that could be improved significantly. One topic that is particularly important when statistical model checking is used to test complex properties is how to analyze and report the reasons why a property is violated during individual simulation runs. In the current version, only the properties that have failed are identified. However, a detailed error report should contain information about which subformulas have been violated and which entities or agents were responsible for these violations. This could help a lot during debugging and for forming a better understanding of the model. The main challenge is to present this data in a comprehensive way. First of all, in statistical model checking it is often not enough to look at the outcome of single simulation runs but instead these results have to be aggregated and grouped in a way so that recurring patterns can be identified.

All improvements that were outlined above are mainly technical and could be realized in a more or less straightforward manner. In addition to that, there are some issues that would require more extensive research. One of these topics is the support for nested probabilistic operators, a feature that has been widely discussed in the statistical model checking literature (see also Section 2.5 and Section 4.5). In short, what makes this difficult is that in order to evaluate a formula like $\mathsf{always}(T, P(\Phi) \geq p_{min})$, it is necessary to estimate the success probability of the sub-formula Φ in every step within T time units. If the default statistical model checking mechanism was used for that, this would require to start a new simulation experiment with multiple runs for each state. Depending on the simulated model and the structure of Φ, each of these simulations could take significant time to finish, which means that any feasible execution strategy without massive use of parallelization would be impracticable even for moderately complex models. In fact, the property evaluation algorithm would have to be extended so that it spawns a set of new parallel simulation jobs in each state and keeps track of the results. As mentioned before, there are now many powerful technologies that could be used to distribute these simulations on large clusters, which could even be provisioned dynamically on cloud computing platforms like the Amazon Elastic Compute Cloud (EC2) [Ama16]. Nevertheless, it is clear that the high computational effort for running such large numbers of simulations can make the use of statistical model checking economically unviable. One generally different approach that might help to overcome this problem could be to combine statistical model checking with methods that are able to calculate probabilities directly. This idea is followed in [YKNP06] where numerical methods for probabilistic model checking are integrated in a statistical model checker to provide a limited support for nested probabilistic operators. In the context of SALMA, adopting such a hybrid strategy would require to find a way to automatically project parts of the model into a more abstract representation like a discrete Markov chain. While this approach could certainly only be used for a very restricted class of nested subformulas, it would definitely be worthwhile to investigate further in this direction.

7.3 Outlook

It is safe to assume that discrete event simulation and statistical model checking will continue to gain importance in the near future. As described above, this is on the one hand due to the fact that simulation-based methods can be scaled almost linearly by parallelization. Nowadays, tools exist that allow the automatic provisioning and management of virtual machines on various cloud computing platforms. Researchers are now actually able to create a cluster of virtual machines in the cloud, deploy and run a distributed simulation on, aggregate the results, and tear down the cluster from their laptops with some simple commands. This makes simulation-based methods like statistical model checking feasible and attractive for use cases for which the cost and effort of using these methods at an appropriate scale used to be too high.

Although most examples presented in this thesis were taken from the domain of cyber-physical systems, the SALMA approach might also fit very well for analyzing systems that are much more common in the current IT landscape. In particular, most popular applications on the Internet have grown so much that they now depend on complex distributed system architectures that make massive use of parallelization of both computational power and storage capacity. Of course, these system structures also impose challenges regarding how to maintain data consistency even in the presence of failures. As the now famous CAP theorem [GL02] shows, compromises have to be made between guaranteeing consistency and availability for distributed systems in which network partitions can occur. However, realizing such a trade-off can be very hard. On the one hand, if guarantees for consistency are weakened, for instance towards a notion of “eventual consistency”, then having to cope with possible (temporal) inconsistency becomes a complex and error-prone responsibility of the application. On the other hand, approaches to achieve guaranteed consistency are also notoriously hard to implement in practice. For example, the Jepsen project [Kin16] regularly performs systematic tests of popular software products like distributed database systems or message brokers. In many cases, these tests revealed that data can be corrupted when partitions occur in certain situations. Since all of the tested products are used in production and these faults escaped the quality assurance processes of the respective projects, this demonstrates how important rigorous testing and simulation is.

This is where SALMA might come in. Although eventually, a pure virtual simulation cannot replace a test of the actual systems, it could be used to conduct simulations first whose results could help to steer the testing process. With its ability to model message transfer, delays, and network errors in a very flexible way and, as shown in Chapter 6, SALMA seems like a natural fit for this task. Furthermore, it allows to model data and data access operations on a semantic level that can easily be extended with additional aspects like location, mobility, or storage constraints. Altogether, SALMA’s flexibility and pragmatic approach could help introducing discrete event simulation and statistical model checking into non-academic “real-life” domains where formal methods are usually not used because they are too difficult or costly to apply. For many projects, this might be a valuable addition to the usual verification and validation techniques. Additionally, the experience gained in such environments could be an important information source for the further development of the SALMA approach and for future research regarding statistical model checking and related fields.

Appendix A

Publications of Christian Kroiß

2016

Christian Kroiß and Tomáš Bureš. Logic-based modeling of information transfer in cyber–physical multi-agent systems. *Future Generation Computer Systems*, 56:124 – 139, 2016. (Own contribution: Main author. Supported by co-author mainly during the description of the case study and the elicitation and description of related work.)

2014

Christian Kroiß and Tomáš Bureš. Logic-based modeling of information transfer in cyber-physical multi-agent systems. In *Second International Workshop on Formal Methods for Self-Adaptive Systems (FMSAS 2014)*, 2014. (Own contribution: Main author. Supported by co-author mainly during the description of the case study and the elicitation and description of related work.)

Christian Kroiß. Simulation and statistical model checking of logic-based multi-agent system models. In *8th International Conference on Agent and Multi-Agent Systems: Technologies and Applications (KES-AMSTA 2014)*, pages 151–160, 2014. (Own contribution: Sole author.)

Christian Kroiß. A statistical model checker for situation calculus based multi-agent models (extended abstract). In *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014)*, pages 1567–1568. International Foundation for Autonomous Agents and Multiagent Systems, 2014. (Own contribution: Sole author.)

2012

Andreas Schroeder, Annabelle Klarl, Philip Mayer, and Christian Kroiß. Teaching agile software development through lab courses. In *Global Engineering Education Conference (EDUCON), 2012 IEEE*, pages 1–10. IEEE, 2012. (Own contribution: Support during conceptual design and realization of exercises.)

2010

Chris Dijksterhuis, Christian Kroiß, and Dick de Waard. Adaptive driving support - information about the vehicle‘s lateral position. In Dick de Waard, Arne Axelsson, Martina Berglund, Björn Peters, and Clemens Weikert, editors, *Human Factors: A system view of human, technology and organisation*, pages 71–87. Shaker Publishing, 2010. (Own contribution: Implementation of a software component that was used for realizing adaptation mechanisms in a car driving simulation experiment.)

2009

Christian Kroiss, Nora Koch, and Alexander Knapp. UWE4JSF: A model-driven generation approach for web applications. In *Proceedings of the 9th International Conference on Web Engineering*, pages 493–496. Springer-Verlag, 2009. (Own contribution: Main author. The paper presents results of the diploma thesis.)

Gilbert Beyer, Moritz Hammer, Christian Kroiss, and Andreas Schroeder. A component-based approach for realizing user-centric adaptive systems. In *Mobile Wireless Middleware, Operating Systems, and Applications – Workshops*, pages 98–104. Springer, 2009. (Own contribution: Participation in development and description of the software framework.)

Gilbert Beyer, Christoph Mayer, Christian Kroiss, and Andreas Schroeder. Person aware advertising displays: Emotional, cognitive, physical adaptation capabilities for contact exploitation. In *Proceedings of the 1st Workshop on Pervasive Advertising at Pervasive*, 2009. (Own contribution: Participation in development and description of the software framework.)

Daniel Ruiz-González, Nora Koch, Christian Kroiss, José-Raúl Romero, and Antonio Vallecillo. Viewpoint synchronization of UWE models. In *Proc. 5th International Workshop on Model-Driven Web Engineering*, pages 46–60, 2009. (Own contribution: Implementation of model transformations.)

2008

Juan Carlos Preciado, Marino Linaje, Rober Morales-Chaparro, Fernando Sanchez-Figueroa, Gefei Zhang, Christian Kroiß, and Nora Koch. Designing rich internet applications combining UWE and RUX-method. In *Web Engineering, 2008. ICWE'08. Eighth International Conference on*, pages 148–154. IEEE, 2008. (Own contribution: Implementation of model transformations.)

2006

Christian Kroiss and Gefei Zhang. Tool supported modeling of mobile systems. In *Proceedings of the 10 th IASTED International Conference on Software Engineering and Applications*, 2006. (Own contribution: Co-author, the paper presents results of the "Fortgeschrittenenpraktikum ").

Bibliography

[AC98] Alan Agresti and Brent A Coull. Approximate is better than "exact" for interval estimation of binomial proportions. *The American Statistician*, 52(2):119–126, 1998.

[ACB10] Alessandro Aldini, Flavio Corradini, and Marco Bernardo. Stochastically timed process algebra. In *A Process Algebraic Approach to Software Architecture Design*, pages 75–124. Springer London, 2010.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[Ama16] Amazon Web Services, Inc. Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2`, May 2016.

[B$^+$13] Tomáš Bureš et al. A Life Cycle for the Development of Autonomic Systems: The e-Mobility Showcase. In *3rd Workshop on Challenges for Achieving Self-Awareness in Automatic Systems*, pages 71–76. IEEE, 2013.

[BA12] M. Ben-Ari. *Mathematical Logic for Computer Science.* Springer London, 3 edition, 2012.

[BBN$^+$03] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gianluigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The klaim project: Theory and practice. In Corrado Priami, editor, *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems*, number 2874 in Lecture Notes in Computer Science, pages 88–150. Springer Berlin Heidelberg, 2003.

[BCD01] Lawrence D Brown, T Tony Cai, and Anirban DasGupta. Interval estimation for a binomial proportion. *Statistical Science*, pages 101–117, 2001.

[BCINN04] Jerry Banks, Jon S. Carson II, Barry L. Nelson, and David M. Nicol. *Discrete-event system simulation*. Prentice Hall, Upper Saddle River, NJ, 4th edition, 2004.

[BCM92] Jr Burch, Em Clarke, and Kl McMillan. Symbolic model checking: 10 20 states and beyond. *Information and ...*, 98(2):142–170, 1992.

[BCR$^+$16] Fabio Bellifemine, Giovanni Caire, Giovanni Rimassa, Agostino Poggi, Federico Bergenti, Tiziana Trucco, Danilo Gotta, Elisabetta Cortese, Filippo Quarta, and Giosuè Vitaglione. Jade website - java agent development framework. `http://jade.tilab.com`, March 2016.

[BDG$^+$98] Jürgen Bohn, Werner Damm, Orna Grumberg, Hardi Hungar, and Karen Laster. First-order-CTL model checking. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1530 LNCS:283–295, 1998.

[BEH$^+$02] Ulrik Brandes, Markus Eiglsperger, Ivan Herman, Michael Himsolt, and M Scott Marshall. Graphml progress report structural layer proposal. In *Graph Drawing*, pages 501–512. Springer, 2002.

[BG96] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. Technical report, University of Bologna, 1996.

[BGHS04] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. *Verification, Model*, pages 44–57, 2004.

[BK00] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, Cambridge, Massachusetts, 2008.

[BKV13] Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In *Runtime Verification*, pages 59–75. Springer, 2013.

[Bla06] Paul E Black. Manhattan distance. *Dictionary of Algorithms and Data Structures*, 18:2012, 2006.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):1–64, 2011.

[BPR99] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE–A FIPA-compliant agent framework. *Proceedings of PAAM*, pages 97–108, 1999.

[BRS$^+$00] Craig Boutilier, Raymond Reiter, Mikhail Soutchanski, Sebastian Thrun, et al. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI/IAAI*, pages 355–362, 2000.

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking using SAT Solving. *Journal of Formal Methods in System Design*, 19(1):7–34, 2001.

[CCC$^+$14] Giacomo Cabri, Nicola Capodieci, Luca Cesari, Rocco De Nicola, Rosario Pugliese, Francesco Tiezzi, and Franco Zambonelli. Self-expression and dynamic attribute-based ensembles in SCEL. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, number 8802 in Lecture Notes in Computer Science, pages 147–163. Springer Berlin Heidelberg, October 2014.

[CCG02] Alessandro Cimatti, Edmund Clarke, and E Giunchiglia. Nusmv 2: An opensource tool for symbolic model checking. *Computer Aided Verification*, 2404(November):359–364, 2002.

[CE82] Edmund M Clarke and E Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic.* Springer, 1982.

[CGL94] E M Clarke, O Grumberg, and D E Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CLRS01] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2 edition, 2001.

[CM13] Luca Cardelli and Radu Mardare. Stochastic pi-calculus revisited. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theoretical Aspects of Computing – ICTAC 2013*, number 8049 in

Lecture Notes in Computer Science, pages 1–21. Springer Berlin Heidelberg, 2013.

[Col03] Nick Collier. Repast: An extensible framework for agent simulation. *The University of Chicago's Social Science Research*, 36:2003, 2003.

[CZ11] Em Clarke and Paolo Zuliani. Statistical model checking for cyber-physical systems. *Automated Technology for Verification and Analysis*, 1041377(2005):1–12, 2011.

[DEDC12] Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. *Prolog: the standard: reference manual.* Springer Science & Business Media, 2012.

[dev15] SciPy developers. The SciPy Stack specification. `http://www.scipy.org/stackspec.html`, 2015.

[DGLL00] Giuseppe De Giacomo, Yves Lespérance, and Hector J Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1):109–169, 2000.

[DNKL$^+$07] Rocco De Nicola, Joost-Pieter Katoen, Diego Latella, Michele Loreti, and Mieke Massink. Model checking mobile stochastic logic. *Theor. Comput. Sci.*, 382(1):42–70, August 2007.

[DS12] M.H. DeGroot and M.J. Schervish. *Probability and Statistics.* Addison-Wesley, 4 edition, 2012.

[DvLF93] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 3(50):3–30, 1993.

[ecl06] The ECLiPSe Constraint Programming System Website. `http://eclipseclp.org/`, March 2006.

[Fow10] Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Pearson Education, 1994.

[GHR93] Norbert Götz, Ulrich Herzog, and Michael Rettelbach. Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras. In *Performance Evaluation of Computer and Communication Systems, Joint Tutorial Papers of Performance*

'93 and Sigmetrics '93, pages 121–146, London, UK, UK, 1993. Springer-Verlag.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51, 2002.

[GLLS09] Giuseppe De Giacomo, Yves Lespérance, Hector J Levesque, and Sebastian Sardina. IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents. *MultiAgent Programming*, pages 31–72, 2009.

[Gly89] P W Glynn. A {GSPM} formalism for discrete event systems. *Proceedings of the IEEE*, 77(1):14–23, 1989.

[Goo16] Google Self-Driving Car Project. https://www.google.com/selfdrivingcar/, March 2016.

[GS05] Radu Grosu and SA Smolka. Monte carlo model checking. *Tools and Algorithms for the Construction and ...*, (631), 2005.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[GvLH$^+$96] Patrice Godefroid, J van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer Heidelberg, 1996.

[Hil96] Jane Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, New York, NY, USA, 1996.

[HJG08] Gerard J Holzmann, Rajeev Joshi, and Alex Groce. Swarm Verification. *2008 23rd IEEEACM International Conference on Automated Software Engineering*, 27(2):1–6, 2008.

[HKNP06] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: A tool for automatic verification of probabilistic systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3920 LNCS:441–444, 2006.

[HKZ$^+$11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[HLP01] K. Havelund, M. Lowry, and J. Penix. Formal analysis of a spacecraft controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, 2001.

[HLR$^+$16] Tom Henderson, Mathieu Lacage, George Riley, Mitch Watrous, Gustavo Carneiro, Tommaso Pecorella, et al. ns-3: A Discrete-Event Network Simulator. `http://www.nsnam.org/`, April 2016.

[HNA16] Andreas Horni, Kai Nagel, and Kay W. Axhausen, editors. *The Multi-Agent Transport Simulation MATSim*. Ubiquity, London, 2016.

[Hol97] Gerard J Holzmann. The Model Checker. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 23(5):279–295, 1997.

[Hol00] G.˜J. Holzmann. Logic Verification of {ANSI-C} Code with {S}pin. *Proc.\ SPIN*, pages 131–147, 2000.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[JCL09] Sumit K Jha, Edmund M Clarke, and Christopher J Langmead. A bayesian approach to model checking biological systems. *... in Systems Biology*, (2005):218–234, 2009.

[Jen96] Henrik E. Jensen. Model checking probabilistic real time systems. pages 247–261, 1996.

[Kar96] Pim Kars. The application of promela and spin in the bos project. In *Proc. Second SPIN Workshop*, 1996.

[KB16] Christian Kroiß and Tomáš Bureš. Logic-based modeling of information transfer in cyber–physical multi-agent systems. *Future Generation Computer Systems*, 56:124 – 139, 2016.

[KEBB12] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. Recent development and applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements*, 5(3&4):128–138, December 2012.

[Kin16] Kyle Kingsbury. Jepsen - distributed systems safety analysis. `http://jepsen.io/`, May 2016.

[KN07] Marta Kwiatkowska and Gethin Norman. Stochastic model checking. *Formal Methods for Performance*, 2007.

[Kro14a] Christian Kroiß. A statistical model checker for situation calculus based multi-agent models (extended abstract). In *13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014)*, pages 1567–1568, 2014.

[Kro14b] Christian Kroiß. Simulation and statistical model checking of logic-based multi-agent system models. In *8th International Conference on Agent and Multi-Agent Systems: Technologies and Applications (KES-AMSTA 2014)*, pages 151–160, 2014.

[Kro16] Christian Kroiß. The salma tolkit website - simulation and analysis of logic-based multi-agent system models. `http://www.salmatoolkit.org/`, June 2016.

[L$^+$97] Hector J Levesque et al. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1):59–83, 1997.

[L$^+$14] Diego Latella et al. Stochastically timed predicate-based communication primitives for autonomic computing. Technical report, QUANTICOL Project, 2014.

[Law14] A. Law. *Simulation Modeling and Analysis.* 2014.

[LCRP$^+$05] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.

[LDB10] Axel Legay, Benoît Delahaye, and Saddek Bensalem. Statistical model checking: An overview. In *Runtime Verification*, pages 122–135. Springer, 2010.

[Lee08] Edward A Lee. Cyber physical systems: Design challenges. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC 2008)*, pages 363–369, 2008.

[LP99] Tim Lechler and Bernd Page. Desmo-j: An object oriented discrete simulation framework in java. In *Proceedings of the 11th European Simulation Symposium*, pages 46–50, 1999.

[LPR98] Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in . . .*, 3(18), 1998.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in Lecture Notes in Computer Science, pages 62–88, August 1995.

[LR97] Fangzhen Lin and Raymond Reiter. How to Progress a Database. *Artificial Intelligence*, 92:131–167, 1997.

[M$^+$95] Daniel Marcu et al. Distributed software agents and communication in the situation calculus. In *International Workshop on Intelligent Computer Communication*, pages 69–78, 1995.

[MV03] K Muller and Tony Vignaux. Simpy: Simulating systems in python. *ONLamp. com Python Devcenter*, 2003.

[NCO$^+$13] Michael J. North, Nicholson T. Collier, Jonathan Ozik, Eric R. Tatara, Charles M. Macal, Mark Bragen, and Pam Sydelko. Complex adaptive systems modeling with repast simphony. *Complex Adaptive Systems Modeling*, 1(1):1–26, 2013.

[Net16] NetworkX developer team. NetworkX - High-productivity software for complex networks. `https://networkx.github.io/`, April 2016.

[NLLM13] Rocco de Nicola, Diego Latella, Michele Loreti, and Mieke Massink. A uniform definition of stochastic process calculi. *ACM Comput. Surv.*, 46(1):5:1–5:35, July 2013.

[PG07] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[Pid95] Michael Pidd. Object-orientation, discrete simulation and the three-phase approach. *Journal of the Operational Research Society*, pages 362–374, 1995.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[Pnu77a] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[Pnu77b] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[Pri95] C. Priami. Stochastic π-calculus. *The Computer Journal*, 38(7):578–589, January 1995.

[pri16] PRISM Manual - Statistical Model Checking. `http://www.prismmodelchecker.org/manual/RunningPRISM/StatisticalModelChecking`, 02 2016.

[PY08] M. Pezzè and M. Young. *Software testing and analysis: process, principles, and techniques.* Wiley, 2008.

[Pyt15a] Python Software Foundation. The Python Language Reference. `https://docs.python.org/3/reference/index.html`, July 2015.

[Pyt15b] Python Software Foundation. The Python Standard Library. `https://docs.python.org/3/library/index.html`, July 2015.

[Rei01] Raymond Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems.* MIT press, 2001.

[Şah05] Erol Şahin. *Swarm Robotics: SAB 2004 International Workshop, Santa Monica, CA, USA, July 17, 2004, Revised Selected Papers*, chapter Swarm Robotics: From Sources of Inspiration to Domains of Application, pages 10–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[San05] Susan M Sanchez. Work smarter, not harder: guidelines for designing simulation experiments. In *Proceedings of the 37th conference on Winter simulation*, pages 69–82. Winter Simulation Conference, 2005.

[Sch03] Richard B Scherl. Reasoning about the interaction of knowledge, time and concurrent actions in the situation calculus. In *18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 1091–1098, 2003.

[Sco98] Rs Scowen. Extended BNF-a generic base standard. *Software Engineering Standards Symposium*, 3(1):6–2, 1998.

[SL03] Richard B Scherl and Hector J Levesque. Knowledge, action, and the frame problem. *Artificial Intelligence*, 144(1):1–39, 2003.

[SLB08] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations.* Cambridge University Press, 2008.

[SS10] Joachim Schimpf and Kish Shen. ECLiPSe - from LP to CLP. *CoRR*, abs/1012.4240, 2010.

[ST01] John Slaney and Sylvie Thiébaux. *Blocks World revisited*, volume 125. 2001.

[sta15] StatsModels: Statistics in Python. `http://statsmodels.sourceforge.net`, July 2015.

[SVA04] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *Computer Aided Verification*, pages 202–215. Springer, 2004.

[SVA05a] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. *Computer Aided Verification*, 2005.

[SVA05b] Koushik Sen, Mahesh Viswanathan, and Gul Agha. VeStA: A statistical model-checker and analyzer for probabilistic systems. *QEST 2005 - Proceedings Second International Conference on the Quantitative Evaluation of SysTems*, 2005:251–252, 2005.

[Var96] M Vardi. An automata-theoretic approach to linear temporal logic. *Logics for concurrency*, 1996.

[Var10] Andras Varga. *Modeling and Tools for Network Simulation*, chapter OMNeT++, pages 35–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[VW86] M Y Vardi and P Wolper. An automata-theoretic approach to automatic program verification, 1986.

[W$^+$45] Abraham Wald et al. Sequential tests of statistical hypotheses. *Annals of Mathematical Statistics*, 16(2):117–186, 1945.

[Woo09] Michael Wooldridge. *An introduction to multiagent systems.* John Wiley & Sons, 2009.

[WSTL12] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[WTM04] Fang Wang, Sofiène Tahar, and Otmane Ait Mohamed. First-Order LTL Model Checking Using MDGs. pages 441–455, 2004.

[YKNP06] Håkan L S Younes, Marta Kwiatkowska, Gethin Norman, and David Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer*, 8(3):216–228, 2006.

[You05a] Hakan Lorens Samir Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events.* Phd thesis, Carnegie Mellon University, 2005.

[You05b] HLS Younes. Ymer: A statistical model checker. *Computer Aided Verification*, 3576:429–433, 2005.

[YS02] HLS Younes and RG Simmons. Probabilistic verification of discrete event systems using acceptance sampling. *Computer Aided Verification*, 2002.

[ZPK00] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems.* Academic press, 2000.