THE LANGUAGE DURA: A DECLARATIVE EVENT QUERY LANGUAGE FOR REACTIVE EVENT PROCESSING

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

vorgelegt von
Diplom-Informatiker
Steffen Hausmann

München, den 7. August 2014

IMPRESSUM

Verlag: epubli GmbH, Berlin, www.epubli.de
ISBN 978-3-7375-1377-7

ERSTGUTACHTER: Prof. Dr. François Bry
ZWEITGUTACHTER: Prof. Dr. Adrian Paschke
TAG DER MÜNDLICHEN PRÜFUNG: 2. Oktober 2014

I may not have gone where I intended to go,
but I think I have ended up where I needed to be.
— Douglas Adams

ABSTRACT

The development of Complex Event Processing (CEP) towards a mature and recognized technology has facilitated the implementation of a wide range of applications in areas like online marketing, logistics, finance, and manufacturing processes. Applications in these domains substantially benefit from the timely detection of higher-level knowledge that is derived by the continuous evaluation of standing complex event queries against a volatile stream of incoming event messages.

Public infrastructures like metro systems and airports are facing similar requirements, as they are increasingly equipped with various sensors and actuators generating a vast amount of events which need to be interpreted in a timely and continuous manner. However, due to the physical nature of such infrastructures, the approaches commonly applied cannot be easily transferred to monitor and control them. Nevertheless, those infrastructures require more elaborated and effective technical solutions to support human operators, as their complexity is approaching the limit that can be reasonably handled by humans, in particular during unusual incidents such as emergencies.

The work presented in this thesis is devoted to closing the gap between the requirements of modern Emergency Management (EM) and the capabilities of current CEP approaches by investigating means towards more reactive event processing, in particular with respect to interactions with external actuators.

A major contribution of this work is the design of the declarative event query language Dura. Dura unifies crucial aspects of EM in a *declarative and homogeneous* language, which so far have been commonly considered only independently from each other in CEP, if at all. Dura provides expressive *event queries* with support of flexible time windows in addition to versatile grouping and negation capabilities. The language furthermore integrates support of *multiple time lines* to leverage information provided by external simulations. Moreover, Dura supplies *stateful objects* that allow for a notion of states and to track the conditions of the infrastructure in a non-volatile yet declarative manner. Even though states are desirable for CEP, not only for EM applications, they have barely received the attention they deserve in many Event Query Languages (EQLs).

A second particular contribution of this work is the elaboration and integration of *complex actions* into Dura to realize composite reactions in response to detected situations. Their seamless combination with queries for events and states facilitates reactions that are dynamically

adapting to the evolving situation in contrast to statically defined procedures from written manuals that are often still used for EM today.

A third major contribution of this work is the elaboration of a provably correct and complete *static temporal analysis* for complex actions that is capable of identifying errors in the specification of complex actions and of ensuring desirable properties of actions prior to their execution. To this end, the analysis is built on a formal *fixpoint semantics* for actions that is suitable for coping with a priori unknown runtime effects stemming from the execution of actions by means of external actuators.

The theoretical aspects of this thesis are complemented by a prototypical *implementation* of Dura and the modeling of EM related use cases that have been elaborated in the European research project *Emergency Management in Large Infrastructures* (EMILI) in collaboration with EM experts and industry partners.

ZUSAMMENFASSUNG

Durch die Weiterentwicklung von Complex Event Processing (CEP) zu einer ausgereiften und anerkannten Technologie konnten zahlreiche Anwendungen, wie beispielsweise in den Bereichen Online Marketing, Logistik, Finanzen und Produktionsprozessen, ermöglicht werden. Derartige Anwendungen profitieren besonders von der zeitnahen Erkennung von höherwertigem Wissen, welches durch die kontinuierlich Auswertung von vorgegebenen Anfragen gegen einen flüchtigen Ereignisstrom hergeleitet wird.

Öffentliche Infrastrukturen, wie U-Bahn Netze und Flughäfen, sind ähnlichen Anforderungen ausgesetzt, da sie zunehmend mit verschiedensten Sensoren und Aktuatoren ausgestattet werden, welche eine Vielzahl von Einzelereignissen melden, die wiederum zeitnah und kontinuierlich interpretiert werden müssen. Durch die besonderen physikalischen Eigenschaften dieser Infrastrukturen können jedoch die für gewöhnlich angewandten Methoden nicht unmittelbar für ihre Überwachung und Steuerung übernommen werden. Nichtsdestotrotz, brauchen diese Infrastrukturen ausgefeiltere und effektivere technische Lösungen um menschliche Betriebsleiter zu unterstützen, da sich ihre Komplexität der Grenze nähert, die noch sinnvoll von Menschen bewältigt werden kann, insbesondere während ungewöhlichen Vorfällen wie beispielsweise Notfällen.

Die in dieser Dissertation vorgestellte Arbeit hat das Ziel die Lücke zu schließen zwischen den Anforderungen von Emergency Management (EM) und den Möglichkeiten von derzeitigen CEP Ansätzen durch die Untersuchung von Methoden zur besseren Unterstützung von Reaktivität im Kontext von Ereignisanfragesprachen, insbesondere unter Einbezug von externen Aktuatoren.

Ein wesentlicher Beitrag dieser Arbeit ist die Ausarbeitung der deklarativen Ereignissanfragesprache Dura. Dura vereint wichtige Aspekte von EM in einer *deklarativen und homogenen* Sprache, welche für gewöhnlich nur unabhängig voneinander für CEP betrachtet werden: Dura bietet ausdrucksstarke *Eventanfragen*, welche flexible Zeitfenster und zusätzlich vielseitigen Negationen und Gruppierungen unterstützen. Darüberhinaus unterstützt die Sprache *mehrere Zeitachsen*, um beispielsweise Informationen von externen Simulationen geeignet abbilden zu können. Desweiteren verfügt Dura über *Stateful Objects*, welche ein Konzept von Zuständen ermöglichen und die Bedingungen der Infrastruktur in einer nichtflüchtigen aber zugleich deklarativen Art abbilden können. Obwohl Zustände im Allgemeinen für CEP und nicht nur für EM wünschenswert sind, erfahren sie in vielen Eventanfragesprachen kaum die angemessene Beachtung.

Ein zweiter wichtiger Beitrag dieser Arbeit ist die Ausarbeitung und Integration von *komplexen Aktionen* in Dura, welche zusammengesetzte Reaktionen auf erkannte Situationen ermöglichen. Ihre nahtlose Einbettung in Event- und Zustandsanfragen ermöglicht Reaktionen, welche sich dynamisch an die sich entwickelnde Situation anpassen im Gegensatz zu statisch vorgegebenen Abläufen wie sie in Handbüchern zu finden sind und immer noch häufig für EM eingesetzt werden.

Ein dritter entscheidender Beitrag dieser Arbeit ist die Ausarbeitung einer nachweisbar korrekten und vollständigen *statischen Analyse* von komplexen Aktionen, welche Fehler in der Spezifizierung von komplexen Aktionen erkennt und wünschenswerte Eigenschaften von Reaktionen vor ihrer Ausführung sicherstellt. Dazu basiert die Analyse auf einer formalen *Fixpunktsemantik* für Aktionen, die für die im Voraus nicht bekannten Laufzeiteffekte geeignet ist, welche durch die Ausführung der Aktionen durch externe Aktuatoren zustandekommen.

Die theoretischen Aspekte der Arbeit werden durch eine prototypische *Implementierung* von Dura und der Modellierung von EM bezogenen Anwendungsfällen ergänzt, die im Zuge des europäischen Forschungsprojekt *Emergency Management in Large Infrastructures* (EMILI) in Zusammenarbeit mit Industriepartnern und EM Experten ausgearbeitet wurden.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

Steffen Hausmann and François Bry. "Towards Complex Actions for Complex Event Processing." In: *Proceedings of the International Conference on Distributed Event Based Systems*. DEBS'13. ACM, 2013, pp. 135–146

Steffen Hausmann, Simon Brodt, Marco Bettelini, and François Bry. "Dynamic Emergency Management." In: *Fachzeitschrift für Information Management & Consulting* (2 2013): *Urban Solutions*, pp. 36–47

Simon Brodt, Steffen Hausmann, and François Bry. *Refinement of the Implementation of Event Processing and ECA Rules for SITE*. Technical Report. EMILI Deliverable D4.7. University of Munich, 2012. 76 pp.

Steffen Hausmann, Simon Brodt, and François Bry. *Modularization Mechanisms for Dura*. Technical Report. EMILI Deliverable D4.6. University of Munich, 2012. 39 pp.

Simon Brodt, Steffen Hausmann, and François Bry. *Implementation*. Technical Report. EMILI Deliverable D4.5. University of Munich, 2011. 57 pp.

Steffen Hausmann. "A Uniform Approach for More Reactivity in Complex Event Processing." Ph.D. Workshop Paper at *International Conference on Distributed Event Based Systems*. DEBS'11. 2011. 6 pp.

Steffen Hausmann, Simon Brodt, and François Bry. *Dura: Concepts and Examples*. Technical Report. EMILI Deliverable D4.3. University of Munich, 2011. 58 pp.

Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. "Two Semantics for CEP, no Double Talk: Complex Event Relational Algebra and its Application to XChange^{EQ}." In: *Reasoning in Event-based Distributed Systems*. Vol. 347. Studies in Computational Intelligence. Springer, 2011, pp. 71–98

Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. "A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed." In: *Reasoning in Event-based Distributed Systems*. Vol. 347. Studies in Computational Intelligence. Springer, 2011, pp. 47–70

Simon Brodt, Steffen Hausmann, and François Bry. *Reactive Rules for Emergency Management*. Technical Report. EMILI Deliverable D4.2. University of Munich, 2010. 51 pp.

Simon Brodt, Steffen Hausmann, François Bry, Olga Poppe, and Michael Eckert. *A Survey on IT-Techniques for a Dynamic Emergency Management in Large Infrastructures*. Technical Report. EMILI Deliverable D4.1. University of Munich, 2010. 48 pp.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my supervisor François Bry. He was always open to intense discussions and gladly shared his knowledge, experience, and advice. Without his encouragement and support, in particular in difficult times, this thesis would not have been possible. I also thank Adrian Paschke for being the external reviewer of this thesis.

Beyond that, I am particular indebted to Norbert Eisinger, who substantially helped me to express and refine my thoughts in countless conversations, regardless of how busy his own schedule was. With his thorough and rigorous character he also helped to improve the descriptions in this thesis. Moreover, I would like to thank Simon Brodt for the comments and suggestions related to my work in addition to the patient explanations of his ideas and of the implementation of the Event-Mill engine.

I enjoyed the friendly atmosphere in the programming and modeling research unit at the University of Munich. The discussions and exchange with my fellow researchers was often encouraging and provided new insights. Moreover, I am thankful for the assistance of our technical staff who did their best to resolve the, sometimes hair-rising, issues with our administration. I also appreciate working with the members of the EMILI project that acknowledged and supported our research. The collaboration and discussion in the project provided inspiration and guidance which is clearly reflected in this thesis. Furthermore, the students Maximilian Scherr and Michael Mayer deserve special thanks for working on topics related to my research.

I am furthermore particular grateful for the numerous volunteers that took care of our daughter, giving me the required freedom to work on this thesis. Last but not least, I would like to thank my family, especially my wife. Getting through the demanding last couple of months would not have been feasible without their continuous encouragement and loving support.

CONTENTS

API	Application Programming Interface.
AST	Abstract Syntax Tree.
CEP	Complex Event Processing.
CI	Critical Infrastructure.
CQL	Continuous Query Language.
CSP	Constraint Satisfaction Problem.
CTP	Conditional Temporal Problem.
DTP	Disjunctive Temporal Problem.
EBNF	Extended Backus-Naur Form.
ECA	Event-Condition-Action.
EM	Emergency Management.
EMILI	Emergency Management in Large Infrastructures.
EPS	Event Processing System.
EPTS	Event Processing Technical Society.
EQL	Event Query Language.
ESB	Event Service Bus.
FCML	Facility Control Markup Language.
HMI	Human Machine Interface.
IED	Intelligent Electronic Device.
MTU	Master Terminal Unit.
PLC	Programmable Logic Controller.
RTU	Remote Terminal Unit.
SCADA	Supervisory Control and Data Acquisition.
SQL	Structured Query Language.

STP Simple Temporal Problem.

STPU Simple Temporal Problem under Uncertainty.

TSA Temporal Stream Algebra.

UML Unified Modeling Language.

XML Extensible Markup Language.

1 INTRODUCTION

1.1 MOTIVATION

Public infrastructures form an essential basis of modern communities living in large towns or cities. In particular Critical Infrastructures (CIs), devoted to areas such as water supply, energy generation and distribution, transportation and shipping, public health, and telecommunication, are indispensable to facilitate and sustain communal life in densely populated urban areas. However, public infrastructures and especially CIs are facing a steadily increasing utilization, which drives them towards their capacity limit: Public transportation systems are chronically overloaded, airports are having difficulties to cope with ever-increasing passenger and cargo volume, and power grids are struggling with increased loads, increasing complexity of the grid, and distributed energy generation by less predictable green energy sources.

In consequence, CIs are continuously being adjusted in order to improve their efficiency and reliability and at the same time increase their safety and security by upgrading them with technical equipment that facilitates a better supervision and regulation of the processes in the infrastructure. Examples thereof include the deployment of various sensors, smoke extraction and fire detection systems, modern public address systems, remotely configurable access control systems, and video surveillance systems. In addition, such equipment is being integrated with existing Supervisory Control and Data Acquisition (SCADA) systems that provide means for the operation and (manual) regulation of distributed infrastructures from a centralized control room.

In particular Emergency Management (EM) is affected by the changing conditions in CIs. Although the additional technology introduces powerful means that improve the operation of CIs, the lack of intelligent control systems prevents an effective utilization of the given means to substantially increase their safety. The added devices even cause further issues as they increase the complexity of and interdependency between systems that commonly operate in isolation according to rigid procedures, making CIs more sensitive for failures. Tragic incidents like the Daegu metro fire or the Düsseldorf airport fire have shown that the assessment of anomalies can massively fail and how bad decisions and wrongly chosen countermeasures can quickly evolve critical incidents into severe disasters affecting and threatening large numbers of individuals.

Accordingly, a new generation of control systems designed to support operators to identify, assess, and counteract emergencies and critical conditions is highly desirable and indispensable for modern CIs. Modern EM requires means to support quick and reliable situation assessment based on a holistic and integrated interpretation of relevant information across the boundaries of single subsystems. Moreover, modern EM must be based on the integration of simulations to facilitate the evaluation and validation of intended strategies and to estimate the evolving conditions within the next minutes. Furthermore, automatically executed reactions need to become more dynamic and must adapt to the global conditions instead of being executed by isolated subsystems. In addition, modern EM requires means for the formalization and autonomous execution of emergency procedures that are currently mostly available in written form from emergency manuals.

Apparently, EM can substantially benefit from technologies capable of filtering, enriching, and correlating large amounts of incoming information and able to execute (semi-)automatic high-level reactions in response to detected situations and incidents. In particular Complex Event Processing (CEP) appears to be a convenient framework for improving EM as it is today: Event queries derive higher-level knowledge in form of complex events by correlating the incoming information in a continuous and timely fashion. In addition, reactive rules triggered by complex events facilitate the timely execution of (semi-)automatic reactions that are tailored to the current conditions. In this way, CEP can facilitate elaborate situation assessments for EM as complex events provide a reasonable and concise high-level abstraction suitable and desirable for the interpretation by human operators. Moreover, time consuming routine tasks do no longer call for the attention of operators which gives them the required time to focus on strategic decisions that cannot be made by computers.

However, although CEP seems well suited to improve EM, the characteristics of CIs, in particular their physical character, and the requirements of EM introduce new challenges so that commonly applied CEP approaches cannot be easily transferred to effectively and adequately monitor and control CIs as it is required for modern EM. One major difference to applications that are commonly consulted for CEP is the physical nature of CIs and the interactions therewith: The particularities of physical actions executed by means of external actuators include the formalization of high-level goals for actions, indirect feedback on their contingent effect, little control over their execution by the Event Processing System (EPS), and in general an irreversible effect. EM is moreover inherently based on a notion of local and global states that provide the context for the interpretation of incidents and for the determination of appropriate reactions. However, such states cannot be conveniently modeled by means of complex events or facts

in a data or knowledge base. In addition, effective and reliable situation assessment by means of complex event queries requires expressive means for the formulation of queries including a generic notion of time, eg, to conveniently model the different times of events that are generated by means of external simulations, and versatile means for the specification of rich event patterns including powerful negation and aggregation capabilities.

1.2 CONTRIBUTIONS

This thesis is devoted to closing the gap between the requirements of modern EM and the capabilities of current Event Query Languages (EQLs) by investigating means towards more reactive event processing. Its central research question is:

> How can notions of states and of complex actions be properly integrated into a *coherent* and *uniform* EQL that facilitates the implementation of applications relying on composite reactions as they are for instance required for EM purposes?

Note, however, that although the work is motivated by EM, the solutions presented in this thesis naturally generalize to a broader range of application domains that require interactions of computer systems with the physical world.

FULL ACKNOWLEDGMENT OF ORTHOGONAL CONCEPTS We propose a model for the three orthogonal concepts events, stateful objects, and actions that are required to obtain a versatile and sufficient degree of reactivity. We furthermore argue that a full language level coverage of all three facets of reactivity in event processing is imperative to obtain an expressive, convenient, and easy to use high-level language that is adequate to support reactivity in event processing.

In contrast, most existing EQLs are tailored to the detection of complex events and often do not integrate notions capable of modeling stateful objects or actions. However, approaches that are lacking one of these concepts are inevitably losing expressiveness and ease of use: If at all expressible, an equivalent behavior depends on misusing available language constructs, which is error prone, inaccessible to non-experts, and prevents systematic analysis of programs at compile time.

SEAMLESS INTEGRATION IN A UNIFORM LANGUAGE A major contribution is the elaboration of the high-level EQL Dura that has been designed to interleave events, stateful objects, and actions in a uniform and seamless language by providing high-level means for an integrated language level support of the three concepts.

The design of Dura provides a unique and deep integration of all three facets of reactivity in event processing, facilitating a coherent association of concepts that are usually considered in isolation, if at all, for CEP. Moreover, Dura is based on deductive rules to derive complex events and integrates expressive reactive rules that generalize conventional Event-Condition-Action (ECA) rules. At the same time, the design of Dura strives for a clear and unambiguous semantics that is free of implicit assumptions and is based on a minimum of language constructs while maintaining a high user friendliness and ease of use.

CLEAR SEPARATION OF DIMENSIONS As pointed out by Bry and Eckert [BEo8b], a clear separation of query dimensions is highly desirable and even mandatory to obtain a high expressiveness and ease of use for EQLs. We build our work on this principle and further extend and adapt the four proposed dimensions of event queries to the concepts that are not covered by XChange^{EQ} [BEo7; Ecko8].

We thoroughly revise the four dimensions of complex events and complement them with further dimensions that divide aspects that have not been clearly separated yet. Moreover, in the spirit of the dimensions of complex events we elaborate dimensions of complex actions that identify aspects that need to be considered for the execution of composite and physical actions and which must not be combined in single monolithic operators of the language.

MODEL FOR PHYSICAL AND COMPLEX ACTIONS Physical actions are executed by external actuators interacting with the physical world and thus have particular properties that are in strong contrast to the properties of (internal) actions that are usually considered in CEP: actions require versatile means for the formalization of their goals, there is only indirect feedback about their contingent effect, the EPS has little control over their execution, and their effect is in general not reversible.

We propose a model for actions that naturally accommodates the particularities of actions as they are desirable to formalize composite workflows in physical environments. Moreover, we elaborate a notion of complex actions that fully covers the identified dimensions of complex actions, in particular temporal aspects as well as a generic notion for the status of actions, and promotes their clear separation.

STATIC ANALYSIS FOR ACTIONS The clear separation of dimensions of complex actions is desirable to obtain a high expressiveness and ease of use. However, the liberties that arise from the expressiveness facilitate the specification of incorrect or inconsistent actions, a circumstance that is avoided in other approaches by the syntactic limitations that result from an insufficient separation of dimensions.

To avoid the execution of improper actions while maintaining the expressiveness of complex actions, a provably sound and complete static analysis is proposed that verifies desirable properties of complex actions at compile time. To this end, we elaborate a fixpoint semantics for complex actions that accounts for the indefinite inherent properties of physical actions. Moreover, to sufficiently capture the sparsely available and highly heterogeneous amount of domain knowledge available on physical actions, the static analysis is designed to scale with available knowledge on actions instead of requiring a complete and precise formalization of their properties and effects.

LANGUAGE CORE AND OPERATIONAL SEMANTICS We elaborate a rather minimal language core of Dura that is sufficient to express all high-level constructs of the entire language. This is not only interesting for theoretical aspects as it provides deep insights into relevant components of the language. It also facilitates the transfer of high-level language constructs of Dura, including stateful objects and complex actions, to languages that have similar properties and capabilities as the language core of Dura.

Moreover, expressing Dura by means of a minimized language core simplifies the elaboration of a reliable and sound operational semantics as it suffices to elaborate an operational semantics for the language core. This is effectively realized by translation to the Temporal Stream Algebra (TSA) [BB12a], a variant of relational algebra adapted to the particularities of streams.

IMPLEMENTATION PROTOTYPE More technically but still worth mentioning, the theoretical foundations in this thesis are accompanied with a prototypical implementation of a runtime environment for Dura that is based on the Event-Mill runtime system capable of evaluating TSA expressions.

The prototype provides a proof of concept implementation that emphasizes and demonstrates the practicality of the language design and the underlying theoretical considerations.

1.3 ORGANIZATION

The rest of this thesis is organized as follows.

We briefly introduce the visions of modern EM in Chapter 2 as they have been identified by the EMILI project and review information technologies suitable to support the elaborated visions in Chapter 3.

Afterwards, we motivate and describe the language design of Dura in Chapter 4 followed by an informal introduction to the language itself in Chapter 5. The understanding of the language and its appli-

cation is subsequently intensified by means of an EM related use case in Chapter 6.

Following this, we address the possibilities of inconsistent specifications that may arise due to the expressiveness of Dura. To this end, a semantics of complex actions is elaborated in Chapter 7 that forms the basis of the static analysis of actions in Chapter 8. The correctness and completeness of the applied analysis is formally verified in Chapter 9.

Eventually, we provide an operational semantics for a particular language core of Dura in Chapter 10 and subsequently extend the operational semantics to Dura in Chapter 11 by elaborating transformations that express Dura programs with the means available in the language core. In addition, we briefly address the prototypical implementation of Dura and illustrate its usage in Chapter 12.

Finally, we discuss future work and research perspectives in Chapter 13 and close with a conclusion in Chapter 14 that summarizes our findings.

Part I

INTRODUCTION TO EMERGENCY MANAGEMENT

2

EMERGENCY MANAGEMENT IN CRITICAL INFRASTRUCTURES

Emergency Management (EM) is devoted to the coordination and integration of activities necessary to build, sustain, and improve the capability to mitigate against, prepare for, respond to, and recover from threatened or actual natural disasters, acts of terrorism, or other man-made disasters [IAE07]. Accordingly, EM aims at the prevention and control of emergencies at different phases which can be roughly classified into mitigation, preparedness, response, and recovery [SBR11a]: mitigation to minimize the effects of potential emergencies by means of appropriate technical and structural adaptations made to infrastructures; preparedness by training operators and personnel in addition to the development of elaborate emergency plans; response in form of immediate reactions to incidents; and the recovery from emergencies and transition back into normal operation.

2.1 A VISION FOR MODERN EMERGENCY MANAGEMENT

The European research project *Emergency Management in Large Infrastructures* (EMILI) [EMI] strived to improve the preparedness and response phase of EM in Critical Infrastructures (CIs) by leveraging and extending state of the art technology currently not adequately considered or not fully exploited for EM. Thereby the efforts were focused on the elaboration of an advanced simulation and training environment intended for thorough and realistic training of staff and on the creation of a new generation of control systems that improves the assistance of operators in critical and emergency situations.

The vision of the EMILI project includes the elaboration of fast computable simulations, the creation of an advanced training environment providing versatile and tailored user interfaces, and using Complex Event Processing (CEP) technologies to improve EM as it is applied today.

BASIC EMERGENCY MANAGEMENT PRINCIPLES Seifert, Bettelini, and Rigert [SBR11b] thoroughly describe how to effectively approach and counteract fire incidents in CIs which serves as the basis for the EMILI use cases.

Initially, potential anomalies need to be reliably identified based on the incoming information and false alarms need to be effectively detected and eliminated. To this end, uncertain alarms indicating potential anomalies, such as suspiciously high temperature readings,

may need to be manually verified. Then, in a first response, rather generic reactions are executed that prepare the affected area for the emerging incident, for instance by increasing the fresh air supply on certain platforms, but do not require detailed information about the incident or substantially interfere with the operation of the infrastructure. Thereafter, the relevant characteristics of the incident need to be assessed, eg, the exact location and size of the fire, to provide a sound basis for the decision of subsequent steps. Moreover, the operation mode of the station and affected areas needs to be adapted appropriately. When the specific characteristics on the location and the size of the fire are available, simulations are carried out and eventually specific actions are carried out, eg, to implement a certain ventilation or smoke extraction strategy, that are tailored to the detected incident and its characteristics.

In the course of the incident all these steps are continuously repeated to integrate newly obtained information and to adapt to the potentially changing conditions in the infrastructure. However, some of the reactions, for instance the adaptation of the ventilation regime, can hardly be reversed or adapted once a certain strategy has been settled upon.

PHYSICAL MODELS AMENABLE TO FAST COMPUTATIONS Simulations based on physical models are commonly used in the design and design verification phase of, eg, railroad and train tunnels, allowing for better performance, enhanced safety, and lower operational costs [Beto1; Beto8]. To this end, simulation tools are used to determine an optimal deployment and positioning of technical equipment, such as, ventilators, fire dampers, and smoke extraction systems. Moreover, they facilitate the elaboration, comparison, and verification of different evacuation strategies.

Three dimensional Computational Fluid Dynamics [Weno9; ANS; McG+13] is applied for the simulation of smoke development and propagation. It is based on numerical models for liquids and gases and provide very accurate results and insights at the cost of substantial preprocessing and postprocessing efforts for building three dimensional models of the infrastructures and of computational costs required to carry out the simulation [Hau+13].

In addition, microscopic egress models [IST; Tra] are applied to simulate different evacuation strategies by individually describing the movements of single persons taking their particular properties and behavior, eg, body size, walking speed, and reaction times, into account [KSMo5].

Although successfully applied for the elaboration of safety concepts for infrastructures, the approaches described above are carried out in the design and verification phase. The underlying computations are too time consuming to be carried out in an online fashion

when the actual boundary conditions of an emerging emergency are available. In contrast, Seifert, Bettelini, and Rigert [SBR11c] envision for EMILI the integration of fast computable simulations into the situation assessment and decision support carried out *while* the emergency takes place based on actually measured boundary conditions. To this end, they elaborate simplified graph-based models for smoke propagation and evacuation simulation that are optimized for speed rather than generality and accuracy. To provide sufficiently meaningful results the models are subsequently calibrated for individual infrastructures based on conventional simulation models and tools [Hau+13; BRS13].

SIMULATION AND TRAINING ENVIRONMENT Training is an integral part of EM in EMILI and of EM in general. The lack of sufficient training has actually been identified as one of several causes leading to undesirable and evitable escalation of incidents caused by inappropriate reactions [SB10b].

The simulation and training environment envisioned for EMILI facilitates the training and evaluation of operators by modeling infrastructures and simulating related EM scenarios. To this end, the simulation and training environment integrates simulators capable of emulating the low level events emitted by the corresponding Supervisory Control and Data Acquisition (SCADA) systems as well as an appropriate Event Processing System (EPS) to emulate a realistic behavior of the training environment. By those means, it is possible to elaborate realistic training scenarios that evolve similar to actual emergencies and take the actions applied by trainees into account. Moreover, the scenarios can be dynamically adapted by supervisors to introduce further challenges.

ADVANCED USER INTERFACE Moreover, the simulation and training environment integrates an advanced and highly sophisticated user interface intended to support human decision makers by visualizing the status and relevant events and incidents of the infrastructure. In addition, the interface accepts commands from operators interacting with the devices in the infrastructure.

The user interface is furthermore not only conceived for training purposes but also to complement or replace the interfaces of currently used SCADA systems.

COMPLEX EVENT PROCESSING Situation assessment forms the basis for the determination and implementation of appropriate and effective strategies and measures to manage emergency situations and to prevent casualties and damage to the infrastructure.

To this end, the EMILI project envisions the integration of CEP technologies into EM to facilitate automatic, quick, and reliable situation

assessment by means of complex event queries that realize the combination and correlation of information provided by individually operating sub-systems. In addition, expressive reactive rules beyond the state of the art are intended to specify high-level reactions that are capable to adapt to the observed conditions in the infrastructure and are executed by the EPS.

2.2 INCIDENTS IN CRITICAL INFRASTRUCTURES

Tragic incidents in critical infrastructures have shown how inappropriate reactions in combination with technical and human failures can escalate to severe incidents with many fatalities [Fri10]. Interestingly, Fridolf [Fri10] finds that incidents can seldom be explained by one single critical event leading to the severe consequences.

The following two incidents have substantially influenced the use cases that are considered by the EMILI project which serve as an inspiration for the design of the Event Query Language (EQL) Dura. Both incidents illustrate some of the causes related to human failure that could have easily been prevented by better training and by means of proper situation assessment and more adequate and immediate reactions.

DÜSSELDORF AIRPORT FIRE On 11 April 1996 a fire in a passenger terminal of the Düsseldorf airport killed 17 people and injured 62 [Vra+10].

The fire was caused by welding works on expansion joints of a road igniting the insulation in the ceiling of the arrival hall located directly below the road. Despite reports of sparks falling from the ceiling the called fire fighters initially struggled to determine the cause and existence of the fire and suspected an electrical failure instead. Eventually, the fire spread through the ceiling in an explosive manner developing a large volume of toxic smoke to the terminal. Seven people died in elevators that remained functional despite of the fire alarm and tragically moved and opened into the fire area. Nine people were trapped and died in a lounge insufficiently informed of the fire and of available evacuation routes because of absent staff and problems with the communication system [Vra+10; NFP].

DAEGU METRO FIRE On 18 February 2003 an arsonist set fire in a metro train in the Korean city of Daegu killing 197 people and injuring around 150 people [BSB11].

Right before the metro train entered a station the arsonist ignited a flammable liquid he carried in two milk packages. The fire spread quickly due to the lacking flame-retardant of the floor and the seat cushions. The burning train stopped in the station where many passengers were able to escape from the train. However, although the

control center was aware of smoke in the station, another train, approaching from the opposite direction, stopped alongside the burning train. Shortly after its arrival the power supply of both trains was automatically shut down preventing the intact train from leaving the station. In addition, the train driver left the intact train while keeping its doors closed effectively trapping the people in this train leading to a high number of fatalities when the fire eventually spread over from the burning train [BSB11; Wik14].

2.3 THREE CHALLENGING USE CASES

The developments in EMILI are motivated and driven by three use cases described in [SB10b; SBR11a], which have been elaborated by industry partners in collaboration with safety experts and are inspired by actual incidents as the ones described above. The use cases can be roughly classified into public CIs and technical CIs: Public CIs include metro stations and airports and involve a great number of persons that are directly threatened by critical and emergency situations and therefore the corresponding use cases focus on the detection of emergencies and support of the early self-rescue phase of passengers. In contrast, technical CIs include power grids and focus on the fast detection of incidents and the automatic determination of their actual cause giving the chance of accurate and right reactions.

The following descriptions summarize the main scenarios of the use cases from [SB10b; SBR11a] which envision how intelligent control systems can support and improve conventional EM. These scenarios have substantially influenced and inspired the design of Dura.

AIRPORT The main scenario of the airport use case considers a fire that affects the terminal of a mid-size airport [SBR11a; Vra+10]:

A fire breaks out in a baggage sorting room located in the basement of the airport terminal. The detection of the fire by means of the correlation of several fire detector sensor signals causes the automatic execution of immediate reactions intended to suppress the fire and prevent its spread. However, despite these efforts the fire eventually spreads through the ceiling to the ground floor where the passport control filled with passengers and personnel is located. It is inferred from the available sensor data that one predetermined evacuation route leading from the ground floor through the basement is already blocked by smoke and thus cannot be used for evacuation. Moreover, the data obtained by the smoke simulation reveals that the second evacuation route is likely to be blocked by smoke within the next minute. Therefore, an alternative route needs to be determined and announced to the persons within the airport. Eventually, a suitable evacuation route leading through the restricted areas of the airport is proposed by the system. By unlocking some of the doors of restricted

areas the alternative route is established and announced by means of the public address systems and emergency lighting directing people to the alternative route.

Variations of this scenario include the manual confirmation of the fire by sending guards to the respective area and the malfunctioning of doors that render further evacuation routes unavailable.

METRO The main scenario of the metro use case considers a train on fire stopping in a metro station consisting of two platforms and a mezzanine level [SBR11a; SBR11b]:

Fire breaks out in the rear part of a metro train approaching a station. When the fire is first detected and confirmed, the evacuation of the platform the train is heading to is prepared by immediate reactions and the operation mode of the station and platform is updated to exceptional and emergency, respectively. Meanwhile the main characteristics of the fire are determined, that is, its exact location, size, and smoke generation, by correlating the information from adjacent sensors. Based on the gathered information, simulations on the likely smoke propagation and passenger evacuation are carried out to choose an adequate ventilation strategy and to determine safe evacuation routes which are subsequently announced through the public displays in the station.

Variations of the scenario include evacuation routes that are unavailable due to maintenance works and additional high passenger volumes caused by a soccer match.

POWER GRID The power grid use case focuses on the detection of the exact origin of technical failures and is thus quite different from the preceding use cases as it does not directly involve the threat of humans or automatic reactions [SBR11a; Esp10]:

In power grids, technical failure, eg, short circuits, easily spread and affect large parts of the grid and can substantially damage expensive and hard to replace technical equipment. So in case of anomalies, the equipment is protected by disconnecting it from the network. However, the protection is easily triggered and also reacts to disturbance of remote equipment to be on the safe side. Moreover, the protection can fail, making it actually necessary that remote protections isolate the defective area. Therefore, the main challenge of this use case is to analyze the alarm messages to draw conclusions of the actual origin of the anomaly so that a small defective area can be isolated from the grid and the greatest possible portion of the power grid remains fully functional.

Variations of this scenario include falsely issued alarm messages and the malfunction of protective devices.

3

FOUNDATIONS OF DYNAMIC EMERGENCY MANAGEMENT

Emergency Management (EM) as it is envisioned in the EMILI project is based on the execution of dynamic reactions based on the detection on certain situation. To this end, characteristic variables and information need to be collected and processed from physical sensors and actuators located in the infrastructure and eventually communicated to the central control room for further processing by means of complex event queries.

The following introduction to Supervisory Control and Data Acquisition (SCADA) systems, Complex Event Processing (CEP), and reactivity in event processing gives an overview of the basic technologies that provide the foundations for the approach towards more reactivity in event processing as it is subject of this thesis. In particular Section 3.2 is based on joint work previously presented in [Eck+11b; Bro+10; BHB10] of which the author of this thesis is coauthor.

3.1 SUPERVISORY CONTROL AND DATA ACQUISITION

SCADA systems [BW03; SFS12] are industrial control systems specifically designed to monitor and control the processes in large and distributed industrial and public facilities from a centralized control center. They are in particular used in public transportation systems, such as, metro systems and airports, as well as in power distribution networks as they are considered within the EMILI project [SB10b; SBR11a]. In simplified terms, a SCADA system automatically controls the processes in the infrastructure on a local level by means of predefined procedures and provides interfaces for human operators to monitor and regulate the applied procedures.

From an event processing perspective a SCADA system roughly corresponds to a specialized Event Service Bus (ESB) that can be deployed to monitor and control industrial and public facilities and provides specialized equipment that is appropriate for the particular prevailing ambient conditions.

From an application level perspective the main purposes of SCADA systems are the *acquisition of data* including basic preprocessing and conversion of raw signals in addition to the exchange of information and commands between devices in the field and the control center; the *local regulation of processes* by means of predefined procedures locally carried out by appropriate devices connected to the deployed sensors and actuators; and the facilitation of *supervision and control*

of the infrastructure by humans operators in the control center on the basis of appropriate visualizations and analytical capabilities as well as interfaces for the application of manual interventions in the automatically executed procedures.

3.1.1 *Basic Components of SCADA Systems*

According to Stouffer, Falco, and Scarfone [SFS12] the basic components of a SCADA system typically include:

Remote Terminal Units (RTUs) and *Programmable Logic Controllers* (PLCs) that provide physical interfaces to sensors and actuators located in the infrastructure. Their purpose is to acquire, convert, and process the (analog) signals of sensors and to exchange commands with actuators. In particular PLCs are furthermore intended to locally monitor and control the processes of the connected equipment according to predetermined static procedures. Although RTUs serve similar purposes, PLCs are in general more versatile and flexible than special-purpose RTUs [SFS12].

Intelligent Electronic Devices (IEDs) which correspond to particular actuators with integrated sensors that operate independently and provide similar processing and communication capabilities as PLCs. IEDs are in particular directly integrated with the communication infrastructure of the SCADA system.

A *Master Terminal Units* (MTUs) in the control center, also referred to as SCADA server, which integrates the communication infrastructure of the SCADA system with user interfaces and a database storing historical data. It receives and processes the signals provided by the available RTUs, PLCs and IEDs and in exchange passes (automatically or manually) issued commands to them.

A *communication infrastructure*, classified into fieldbus networks and control networks. Fieldbus networks connect the sensors and actuators in the field with RTUs and PLCs which are in turn connected to the MTU by means of the control network. To this end, various landlines and proprietary protocols are available [BW03; GH13] whereas modern systems increasingly rely on standard Internet technologies and XML-based protocols, such as, the Facility Control Markup Language (FCML) [Bry+08].

A *Human Machine Interface* (HMI) which provides an interface between human operators in the control center and the technical equipment in the infrastructure. It displays the current condition of the entire infrastructure in a suited graphical manner and furthermore facilitates the exploration and investigation of issues related to certain components or incidents. Moreover, the HMI facilitates operators to manually enter commands and to intervene into the predefined procedures executed by means of PLCs and IEDs.

3.1.2 *Limitations with respect to Emergency Management*

SCADA systems are crucial components of any sufficiently large infrastructures. Public infrastructures as they are considered in the EMILI project are experiencing a continuously increasing load, eg, in terms of passenger volume [SB10a]. To compensate for the higher demands and to increase their efficiency and safety the infrastructures are continuously extended and the facilities gradually are upgraded with (intelligent) technical equipment including various sensors. However, the additional equipment drastically increases the amount of information that needs to be processes, both, by computers and humans. As a consequence, the capabilities of conventional SCADA systems do no longer seem to be sufficient and appropriate to adequately monitor and control modern critical infrastructures, in particular during emergency situations. They are often not meant to process and display the amount of information in a way that is suitable for the interpretation by humans.

Seifert and Bettelini [SB10b] analyze conventional SCADA systems that are currently applied to monitor and control Critical Infrastructure (CI) with respect to their limitations for EM which include:

Heterogeneous Infrastructure: Devices from different manufacturers require different communication protocols that are expensive to integrate and to maintain. Moreover, due to the heterogeneity of devices, the representation of information on measured physical values may differ in terms of structure and applied measuring units and thus require additional efforts to gain a homogeneous representation.

Insufficient Integration of Systems: The acquisition of data and regulation of devices is merely carried out locally by means of more or less isolated PLCs and IEDs whereas insights that could be gained from a holistic integration and correlation of relevant information from different sub-systems do barely affect the automatic regulation of devices. As a consequence, the algorithms deployed on PLCs and IEDs regulating the local processed can be considered as mostly static in the sense that they have only very limited capabilities to automatically adapt to the global conditions of the infrastructure.

Limited Situation Assessment: Due to their limited capabilities in terms of processing power and the poor integration of systems on a local level, the capabilities of local components to correlate information and adapt to the condition of other sub-systems is very limited. However, even in the control center where all information flows together there are often only very basic means to correlate events from different subsystems and to automatically identify the actual cause of an incident: Anticipated alarm messages often require manual interpretation by human operators what tends to be time consuming and prone to failure, in particular when a high number of alarms is

anticipated. As a consequence, important messages can easily get lost in a cluttered log of highly redundant and cascaded alarm messages.

Limited Decision Support: Emergency procedures are only available in written manuals and are hence difficult to assess by humans, in particular in emergency situations where quick and at the same time well-considered reactions are required to prevent harm from passengers, personnel, and the infrastructure itself. Moreover, the implication and effectiveness of intended steps needs to be assessed based on the experience of the operator without proper software support, such as, appropriate simulations capable of estimating the likely development of the conditions within the infrastructure in the near future and of assessing the implication of anticipated reactions before they are actually carried out.

3.2 COMPLEX EVENT PROCESSING

CEP is a technology devoted to the timely derivation of higher-level knowledge from a high-volume stream of volatile events. Thereby, CEP differs from conventional database technology in the way information is represented and queries are evaluated: In conventional databases information is stored in persistent relations that are queried in a sudden ad-hoc manner. In contrast, in CEP systems information is communicated in form of volatile event messages and queries are persisted and continuously evaluated facilitating a timely detection of query answers in form of complex events.

The core of CEP form complex event queries that specify event patterns to filter, enrich, and correlate events of the volatile and high-volume stream of incoming events. To this end, complex events, corresponding to situations that can only be recognized as patterns of several events emerging in the event stream, are specified by means of Event Query Languages (EQLs) specifically designed for the convenient and effective specification of complex events.

In [Eck+11a] we have elaborated five language styles by distinguishing different categories of EQLs based on the general concepts that are applied for the specification of complex events, namely, composition operator based languages, data stream query languages, production rules, timed automata, and logic languages. Based on these five language styles we have surveyed different EQLs by classifying them into the identified language styles and by discussing the general ideas for the specification of each style. It follows a survey and comparison of EQLs based on joint work presented in [Bro+10; BHB10; Eck+11a] that is thoroughly revised and further extended.

For a related overview that focuses on the elaboration of a common model that is applicable for all kinds of EQLs and compares different languages and systems individually according to properties of the

model, such as, their underlying data and time model and supported operators, refer to [CM12b].

COMMON NOTIONS AND CONCEPTS To provide a common basis for the comparison of different EQLs, we will briefly introduce some of the most common concepts related to complex events and complex event queries which are particular relevant for the following overview of EQLs. For a more thorough and comprehensive introduction of various notions related to CEP refer to the event processing glossary [LS11] assembled by the Event Processing Technical Society (EPTS).

Each event has a *type* and carries a *payload* of application dependent data. In the following, we restrict the payload of the events to flat attribute value pairs which can be represented as flat tuples. Moreover, with few exceptions, events are commonly associated with a *time interval* that degenerates to a time point in case of basic events and comprises the times of all events that contributed to the detection of a composite event otherwise.

Variables specified in atomic event queries are commonly used to bind or extract values from the payload of events. Furthermore, *event identifiers* preceding atomic (and sometimes even composite) event queries are often used to refer to the time of matched event instances.

In addition, EQLs often provide a notion of *relative timer events* to define a time window relative to the time of a previously matched event. They are in particular desirable as they facilitate the accumulation of events over a time window, as it is required for negation and aggregation.

Finally, the term *garbage collection* is commonly used to refer to the automatic discarding of events that are no longer relevant for the evaluation of queries in a way that remains transparent for the programmer.

ILLUSTRATIVE EXAMPLE The following four example queries are subsequently used to illustrate and compare the basic properties of each language style:

- `Q1`: detect a fire when smoke and high temperature occur in the same area within one minute
- `Q2`: detect a fire when smoke occurs followed by the detection of high temperature in the same area within one minute
- `Q3`: detect a sensor failure when the latest temperature reading of a sensor occurred more than 12 seconds ago
- `Q4`: when a temperature reading is detected compute the average temperature reported by this sensor within the last minute

In the following, these queries are formulated by means of query languages representative for each respective language style based on

occurrence of the three events `smoke`, `temp`, and `hightemp`. Thereby, the payload of all given events contains an `area` attribute referring to the location of the respective sensor. In addition, the payload of `temp` events contains a `sensor` and a `value` attribute providing a key uniquely identifying the sensor and the measured temperature, respectively.

3.2.1 *Composition Operator Based Languages*

Composition operator based languages, or event algebras as they are sometimes called [PK09], have their origin in active databases [PD99] where they are specified in the event part of Event-Condition-Action (ECA) rules to determine when rules are triggered.

In this domain, composite event queries are specified by means of composition operators that combine several atomic or composite sub-queries. To this end a myriad of operators is available whereby the most common ones are the n-ary conjunction and sequence operators and the ternary negation operator. Conjunctions and sequences simply specify that certain (complex) events need to occur in any and a particular order, respectively, often in addition with a duration that constrains the time interval between the occurrence of the events. In contrast, negations specify that a certain event must not occur between two events. Other, less common operators include periodic and aperiodic operators [Cha+94] or at least n and at most n operators [AE04]. However, despite the variety of available operators, the aggregation of events is not supported by means of composition operators.

Composition operator based languages often provide several event selection and consumption policies [GD94; ZU99; AE04] which determine which one out of a set of matching events contribute to a query, eg, the first or last one, and if events can be matched by multiple rules or multiple times by the same rule.

ILLUSTRATIVE EXAMPLE The language GEM [MS97] is used in Figure 3.1 to implement the example queries described above. GEM applies an interval based time semantics for events and runs independently from a database system.

In GEM composite queries specifying complex events are build from the following operators: `e1 & e2` matching whenever the events `e1` and `e2` occur in any order, `e1 ; e2` matching whenever `e1` occurs before `e2`, `e1 | e2` matching whenever `e1` or `e2` occurs, `{e1 ; e2} ! e3` matching whenever `e1` is followed by `e2` with no interleaving `e3`, and `e + d` matching `d` time units after `e` occurs. Moreover, identifiers preceding atomic or composite sub-queries are used in combination with so-called guards that are specified in a supplemental `when` part of a query to determine conditions on the values of attributes and the time

```
Q1: x:( s:smoke & t:hightemp )
      when (s.area == t.area) && (@x - |@x) < [1*min]

Q2: ( s:smoke ; t:hightemp ; s+[1*min] ) when (s.area == t.area)

Q3: { t1:temp ; t1+[12*sec] } ! t2:temp when (t1.sensor == t2.sensor)
```

Figure 3.1: Running Example in GEM

of matching events. Thereby `@` and `|@` are used in combination with an identifier to access the begin and end of an event.

Query `Q1` is implemented by means of a conjunction operator (`&`) and queries for `smoke` and `hightemp` events in combination with a guard constraining the area and the time window of matching events. To this end, the duration of the time interval of `x`, which comprises the time of matched `smoke` and `hightemp` events, is restricted to one minute by means of the condition `(@x - |@x) < [1*min]`. Similarly, `Q2` is realized by a sequence operator (`;`) in combination with a guard, but in contrast to `Q1`, the temporal restriction is specified by means of the relative timer event `s+[1*min]`, which is matching one minute after the occurrence of the referred `smoke` event. In case of query `Q3` a negation operator (`!`) is applied to match `temp` events that that are not followed by an appropriate `temp` event within 12 seconds. As mentioned before, aggregation is not supported by means of composition operators and thus `Q4` is missing from Figure 3.1.

LANGUAGES AND SYSTEMS Languages based on composition operators have their roots primarily in active databases [PD99]. This class of languages includes the COMPOSE language of the Ode active database [GJS92b; GJS92a; GJS93], the composite event detection language of the SAMOS active database [GD93; GD94], and Snoop [Cha+94]. However, these early event algebras apply a point based time semantic for events which suffers from severe shortcoming with respect to the semantics of composite queries as it is pointed out, eg, by [GA02]: Under point based time semantics the query `b ; (a ; c)` counter-intuitively matches event instances `a`, `b`, and `c` occurring in this particular order, as the time point of the composite event matching `(a ; c)` coincides with the time point of `c`. Moreover, these early approaches merely considered events without any payload.

In contrast, more recent composition operator based languages, usually apply interval based time semantics for events, which avoids the aforementioned anomalies. Moreover, they support events carrying a user defined payload and often run independently from a database. Those kind of languages include, eg, SnoopIB [AC05; AC06], CEDR [BC06], ruleCore [SB05], GEM [MS97], Amit [AE04], the SASE event language [WDR06], and the original event specification language of XChange [Pat05; BEP06b].

3.2.2 *Data Stream Query Languages*

Data stream query languages have been developed in the context of relational stream management systems. The specification of continuous queries resembles the specification of queries in SQL using the common `SELECT`, `FROM`, `WHERE`, and `GROUP BY` clauses. In addition those languages extend conventional SQL queries with temporal constructs to select relevant portions of streams, eg, all events that occurred within the last minute or the last n events of a stream with respect to the current time.

Data stream query languages are very suitable for aggregation of event data and usually offer a good integration with databases, sharing in particular the common basis of SQL. They are, however, not focused on detecting complex patterns of events in a stream.

ILLUSTRATIVE EXAMPLE The Continuous Query Language (CQL) [ABW06] is used in Figure 3.2 to implement the example queries. In CQL streams and (time-varying) relations are clearly distinguished. To integrate both notions CQL provides three different classes of operators, namely, relation-to-relation, stream-to-relation, and relation-to-stream operators, that either apply to relations or to streams.

A CQL query commonly consists of the following three components. To begin with, streams are converted to relations by means of stream-to-relation operators, that is, window-based operators selecting relevant portions of the stream. The core of the queries is expressed by relation-to-relation operators which are derived from the traditional SQL queries operating on relations. Eventually, the resulting relation is converted back to a stream by means of a relation-to-stream operator. Conceptually, this process is done at every point of time which implies a discrete time axis.

There are three different kinds of stream-to-relation operators that are used to convert streams to finite relations by selecting the latest porting of a stream: Time-based sliding windows, such as, `[Range 1 Minue]` and `[Now]`, to select the events obtained within the last minute or the current instant, tuple-based windows, such as, `[Rows N]`, to select the latest `N` events in the stream, and partitioned windows, such as, `[Partition By` A_1`, ...,` A_k `Rows N]`, to partition the stream according to the attributes A_i and to apply a tuple-based window of size `N` to each partition. Note that the content of a relation obtained by means of a stream-to-relation operator actually depends on the instant at which the operator is applied.

CQL comes with three relation-to-stream operators, namely, *Istream*, *Dstream*, and *Rstream*, that are applied to convert time-varying relations back to streams: *Istream*`(R)` and *Dstream*`(R)` correspond to the positive and negative delta of a relation `R`, that is, the resulting stream contains events that have been added and removed, respectively, from

```
Q1: SELECT Istream(s.area)
    FROM smoke [Range 1 Minute] s, hightemp [Range 1 Minute] t
    WHERE s.area = t.area

Q2: SELECT Istream(s.area)
    FROM smoke [Range 1 Minute] s, hightemp [Now] t
    WHERE s.area = t.area

Q4: SELECT Istream(t1.sensor, avg(t1.value))
    FROM temp [Range 1 Minute] t1, temp [Now] t2
    WHERE t1.sensor = t2.sensor
    GROUP BY t1.sensor
```

Figure 3.2: Running example in CQL

the relation R with respect to its state in the preceding instant. In addition, `Rstream(R)` simply contains each tuple that is contained in the relation R at any past instant, and thus, if applied to a relation obtained by means of a time window other than *Now*, may contain many duplicates.

In Figure 3.2 query `Q1` is realized by a join between the streams `smoke` and `hightemp` over the attribute `area`. To match events that occurred within one minute, only the last minute of each stream is considered for the conversion to a relation by means of the sliding window operator `[Range 1 Minute]`. `Q2` is realized similarly, but note that the operator applied to the `hightemp` stream is adapted to *Now* in order to accomplish that matching `hightemp` events occur after `smoke` events. To implement query `Q4` a grouping is applied to determine the average of the temperature of matching events within the last minute. Note that temporal windows realized by means of stream-to-relation operators are always backward directed and that the time of events is not explicitly contained in the schema of events. As a consequence, the following CQL query does not formalize the intension of the query `Q3` in an appropriate manner:

```
SELECT Istream(t1.sensor)
FROM temp [Now] t1
WHERE NOT EXISTS ( SELECT *
                   FROM temp [Range 12 Seconds] t2
                   WHERE t1.sensor = t2.sensor )
```

The given query matches if there is a temperature event and 12 seconds *before* its occurrence no corresponding temperature event occurred, ie, if an intermediate temperature event is missing. However, if a sensor breaks, it suddenly stops sending signals and thus the query does not match because until then, at least every 12 seconds a temperature event is sent by the sensor.

LANGUAGES AND SYSTEMS A typical example of a data stream query language is the previously introduced CQL [ABW06] of the

STREAM systems. The general ideas behind CQL apply to a number of open-source and commercial languages and systems including Esper [Esp], the CEP and continuous query component of the Oracle Fusion middleware [Orab], Microsoft StreamInsight [Mic; KGR12], and SAP Sybase Event Stream Processor [SAP; SAP13], formerly known as Coral8.

In addition, there are further SQL dialects that have been tailored to CEP, eg, [KS09], [Dem+07] and [KLG07; LGI09], which merely provide stream-to-stream operators and thus spare the conversion between relations and streams.

3.2.3 *Production Rules*

Production rules and production systems have their origin in the domain of artificial intelligence where they are used to infer knowledge by applying a set of rules to a given knowledge base, also referred to as working memory. Each rule consists of two parts specifying a condition and action, respectively. Whenever the conditions of a rule apply for the current state of the working memory, the (update) action of the rule is triggered which alters the working memory and may in turn trigger further rules.

```
when «condition» then «action»
```

Production rules are often well integrated with existing programming languages, such as, Java, and are thus very flexible and highly adaptable. For a deeper introduction to production rules in general refer to [Ber+07].

As production rules are applicable to large knowledge bases and incremental evaluation, eg, by means of the Rete [For82] algorithm and its successors TREAT [Mir87] and LEAPS [Bat94], they are also convenient to realize CEP in a very flexible way, although they are no proper EQLs in a narrower sense. To this end, events are represented by means of facts in the working memory and event queries are represented by means of rules that specify conditions for these facts and insert detected complex events into the working memory by means of appropriate actions. In this way, the programmer has much freedom but little guideline as it entails working on a low abstraction level that is somewhat different from other EQLs because it is primarily state and not event oriented. Moreover, depending on the system, the user needs to manually implement garbage collection on events that cannot contribute to queries anymore or needs to actively trigger the evaluation of rules, issues that are usually automatically taken care of by Event Processing Systems (EPSs). See however, eg, [WBG08; WGB08] for work on automatic garbage collection for production systems.

```
Q1: when
      $s: Smoke() over window:time( 1m )
      HighTemp( area == $s.area ) over window:time( 1m )
    then
      insert(new Alarm( $s.area ));

Q2: when
      $s: Smoke()
      HighTemp( this after[1m] $s && area == $s.area )
    then
      insert(new Alarm( $s.area ));

Q3: when
      $t: Temp()
      not( Temp( this after[12s] $t && sensor == $s.sensor ) )
    then
      insert(new Failure( $t.sensor ));

Q4: when
      $t: Temp()
      $avg: Number() from accumulate(
        Temp( this during[-1m, 0s] $t &&
              sensor == $t.sensor && $val : value ),
        average($val))
    then
      insert(new AvgTemp($t.sensor, $avg));
```

Figure 3.3: Running Example in Drools Fusion

ILLUSTRATIVE EXAMPLE Figure 3.3 illustrates how to implement the example queries by means of Drools Fusion [Drob], a production system tailored to CEP. In Drools all events are represented as Java objects. Every event that arrives at the EPS is converted into a Java object and inserted into the working memory.

As already mentioned, production rules consist of two parts specifying a condition and an action, respectively. The condition in the `when` part of rules combines the specification of event types to be matched with instances in the working memory with conventional Java expressions. A rule triggers whenever the specified event types can be matched with instances in the working memory satisfying the conditions on their attributes and all specified boolean expressions evaluate to true.

The actions in the `then` part are composed from Java expressions. Thereby, the particular system functions *`insert`*, *`retract`*, and *`update`* are available to alter the working memory, eg, to insert detected complex events into the working memory. However, the `then` part can as well specify method invocations facilitating arbitrary computations. Accordingly, the specification of reactions is very flexible and can influence the derivation of events on a very low level. Therefore, programmers needs to pay close attention as, eg, inadvertent updates

and retractions of events can severely impact the desired semantics of rules.

Query `Q1` is implemented by matching `Smoke` and `HighTemp` events in the working memory with coinciding `area` attributes occurring within one minute. To this end, the identifier `s` is used in the query for `HighTemp` events to constrain the value of the `area` attribute. Moreover, the time window `window:time(1m)` is applied to both event queries to match only events that occurred within the last minute. Likewise, `Q2` is realized by matching `Smoke` and `HighTemp` events, but in contrast to `Q1`, the occurrence of `HighTemp` events is constrained to be at most one minute before the occurrence of `Smoke` events by means of the condition `this after[1m] $s`. `Q3` queries the absence of `Temp` events which is realized by means of a `not` operator that matches if the respective event is not contained in the working memory. Note that thereby the temporal condition `this after[12s] $s` causes a deferral of the query evaluation which is required for the sound evaluation of the negation. The accumulation of events required to for query `Q4` is realized by means of the operators `accumulate` and `average` in addition to the appropriate (temporal) conditions on the queried events.

LANGUAGES AND SYSTEMS The first successful production rule engine is OPS [FM77], in particular in the incarnation OPS5 [For81]. Since then, many others have been developed in the research community and industry, including systems like Drools Expert [Droa], Jess [San], and IBM Operational Decision Manager [IBM], formally known as ILOG JRules.

Besides their use in business rule management systems not focused on events, production rules are also an integral part of the CEP products TIBCO Business Events [TIB] and Drools Fusion [Drob], which offer more CEP-specific features such as integrated support for temporal aspects or automatic garbage collection of events.

3.2.4 *Timed Automata*

Finite automata are convenient to model the behavior of a stateful system that reacts to events. In particular their advancement towards timed automata are applied for the modeling and verification of real-time systems [AD90] and more recently also for CEP [Pop+13].

Timed automata are based on finite automata, which correspond to directed graphs whereby nodes determine states and edges determine transitions between states. Thereby, the labels of edges specify event types to be matched and conditions on the occurrence of events including temporal conditions formalized by means of variables modeling clocks. Accordingly, states, in particular accepting states, in timed automata are reached by traversing a particular se-

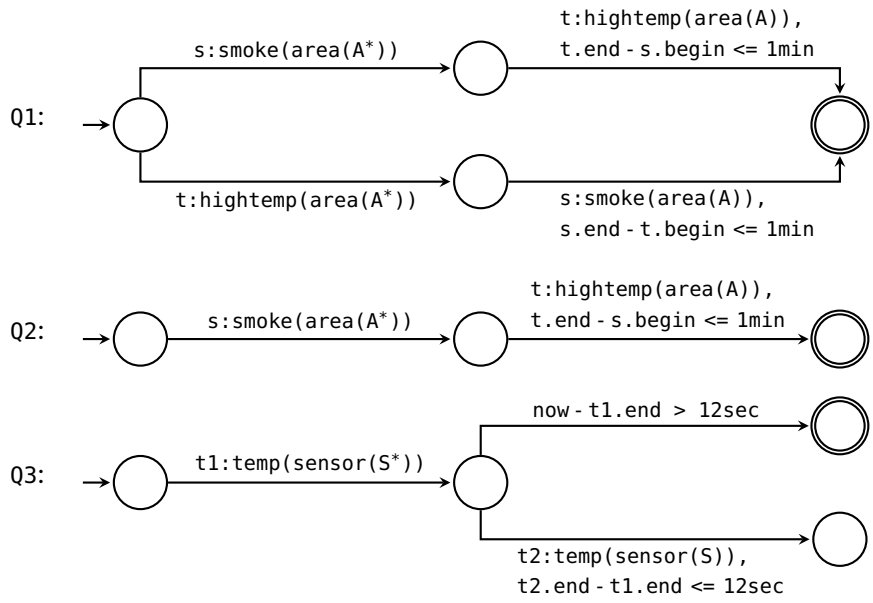


Figure 3.4: Running Example in HIT

quence of multiple events occurring over time and therefore timed automata implicitly specify complex events queries.

ILLUSTRATIVE EXAMPLE Figure 3.4 illustrates how to implement the example queries by means of Hierarchical Instantiating Timed Automaton (HIT) [Pop+13], a substantial extension of timed automata towards complex event processing.

In HIT states are modeled by means of (potentially nested) timed automata and application workflows, implicitly describing complex event queries, are captured by means of different states and event-driven transitions between them. To this end, finite automation are extended so as to support edge labels that contain, in addition to the specification of event types, conditions on the payload of events formalized by means of variables that are not limited to merely model clocks.

The implementation of query `Q1` requires two different branches in the automaton. To match `smoke` followed by `hightemp` events, the edges of the upper branch are labeled with `s:smoke(area(A*))` and `t:hightemp(area(A))`, whereby `A*` denotes a variable definition that is subsequently referenced by `A`. In addition, the temporal conditions `t.end - s.begin <= 1min` and `s.end - t.begin <= 1min` constrain the temporal distance between matching events to one minute. As the order of `smoke` and `hightemp` events is irrelevant for `Q1`, a second branch is required that specifies `smoke` and `hightemp` events in reversed order. The implementation of `Q2` simply corresponds to the upper branch of the implementation obtained for `Q1`, as in `Q2` `smoke` must occur before `hightemp`. The negation of `Q3` is realized by means of an error state that

prevents the automaton from accepting a sequence of events where consecutive `temp` events arrive on time. To this end, the label of the edge leading to the accepting state does not specify an event but merely contains the condition `now - t1.end > 12sec` that refers to the current time by means of the particular variable `now` and thus matches when the sufficient amount of time exceeded. Event accumulation, as it is required for `Q4`, cannot be expressed by means of HIT automaton but instead needs to be realized by means of an integrated rule language which is not presented here.

LANGUAGES AND SYSTEMS Timed Büchi Automata (TBA) [AD90] are an early attempt to extend automata with temporal aspects to model real-time systems. Representatives of this language style have often been developed to achieve a particular task or solve a problem that is specific to real-time distributed systems. Examples include the timed abstract state machine language for real-time system engineering [OL08], a timed automata approach to real time distributed system verification [Kra+04], timed-constrained automata for reasoning about time in concurrent systems [MMT91], and regular real-time languages [HRS98]. Recently, timed automata have been specifically tailored towards the requirements of CEP, such as, HIT [Pop+13] introduced above.

Further related languages of this language style include UML activity diagrams [OMG11b] and the Business Process Model and Notation (BPMN) [OMG11a].

3.2.5 *Logic Languages*

Logic languages express event queries in logic-style formulas whereby events are represented as facts or terms. Complex event queries are specified by means of deductive rules deriving higher level events based on the occurrence of events in the stream:

$$event \leftarrow query$$

Deductive rules are incrementally evaluated to fit the volatile nature of event streams, ie, new events are gradually derived according to the specification in the rule head and added to the event stream. Hence, deductive rules correspond to materialized views from database systems. However, note that in spite of the resemblance with production rules, deductive rules do not specify (imperative) actions that alter some kind of working memory but instead derive events in a declarative manner.

ILLUSTRATIVE EXAMPLE The language XChange^{EQ} [BE07; Eck08] adopts ideas from event calculus-like approaches [KS86], but extends and tailors them to the needs of an expressive high-level event query

```
Q1: DETECT
      fire{ area{var A} }
    ON
      and{
        event s: smoke{{ area{var A} }},
        event t: hightemp{{ area{var A} }}
      } where { {s,t} within 1min }
    END

Q2: DETECT
      fire{ area{var A} }
    ON
      and{
        event s: smoke{{ area{var A} }},
        event t: hightemp{{ area{var A} }}
      } where { s before t, {s,t} within 1min }
    END

Q3: DETECT
      failure{ sensor{var S} }
    ON
      and{
        event t: temp{{ sensor{var S} }},
        event i: timer:from-end[event t, 12sec],
        while i: not temp{{ sensor{var S} }}
      }
    END

Q4: DETECT
      avgtemp{ sensor{var S}, value{ avg(all var T) } }
    ON
      and{
        event t: temp{{ sensor{var S} }},
        event i: timer:from-end-backward[event t, 1min],
        while i: collect temp{{ sensor{var S}, value{var T} }}
      }
    END
```

Figure 3.5: Running Example in XChange^{EQ}

language. Moreover, XChange^{EQ} adopts the pattern based query approach from [BS02; FBS07] to facilitate queries of events that carry a payload of semi-structured data.

In XChange^{EQ}, events are represented as data terms which correspond to XML messages. Atomic event queries are specified by means of an incomplete query pattern preceded by the keyword *event* and an event identifier that refers to the time of matched event instances. Event queries are composed from the operators `and` and `or` in addition to supplemental `where` parts containing temporal and other relationships between events. Negation and accumulation is realized by means of relative timer events, which specify convex time windows relative to a queried event, in combination with the operators *while*, *not*, and *collect*.

Complex events are specified by means of deductive rules consisting of two distinct parts, the rule head and the rule body, that share commonly named variables and are denoted

```
DETECT «event» ON «query» END
```

In Figure 3.5, the rule corresponding to query `Q1` queries atomic events by means of the two query patterns `smoke{{ area{var A} }}` and `hightemp{{ area{var A} }}`. Thereby, the doubled curly brackets denote that the pattern is incompletely specified, that is, the payload of the two events may contain additional attributes. The queries are furthermore connected by means of a conjunction whereby the recurring definition of the variable `A` entails a join condition. In addition, the temporal condition `{s,t} within 1 min` in the separate `where` part constrains the time of matching events. Note how in this way of formulating a query a clear separation of the complementary dimensions of event queries, namely, data extraction, event composition, temporal (and other) relationships between events, and event accumulation, is established. The implementation of `Q2` almost exactly coincides with the one of `Q1`, only the condition `s before t` is added to the `where` part. The negation required for `Q3` is realized by means of *while* and *not* in combination with the relative time event `timer:from-end[event t, 12sec]`, which specifies a time window for the negation. The aggregation of events in `Q4` is realized similarly: *collect* is used instead of *not* to collect matching events specified within a time window determined by means of a relative timer event and `avg(all var T)` is used in the query head to determine the average of all collected temperature values.

LANGUAGES AND SYSTEMS An early representative of this language style is the event calculus [KS86]. While event calculus is not an event query language per se, it has been used to model event querying and reasoning tasks in logic programming languages such as Prolog [CM03] and Prova [Koz+06]. More recently, variants of the event calculus have been proposed that extend the event calculus to be better suited for CEP, eg, the interval based event calculus [Pas06; PKB07] and the event calculus for runtime reasoning (RTEC) [ASP12; Art+12].

The previously introduced rule based language XChange^{EQ} [BE07; Eck08] strives for high-level language constructs tailored to the requirements of CEP and a clear separation of orthogonal dimensions of complex events. Further examples of this language style include Reaction RuleML [Pas+12], which combines derivation rules, reaction rules, and other rule types such as integrity constraints into the general framework of logic programming, the rule language TESLA [CM10; CM12a] which, although being declarative, provides versatile event consumption capabilities, and the unnamed language proposed in [Ani+10; Ani+12b; Ani+12a].

3.2.6 *Summary*

Composition operators allow an intuitive specification of event patterns. Particularly temporal relationships and negation are well-supported. Event instance selection and consumption are features that are not very common in other approaches. However, composition operator based languages have very limited aggregation capabilities and often are tightly coupled to databases.

Data stream query languages are very suitable for aggregation of event data and offer a good integration with databases. However, specifying complex event patterns that rely on negation and expressive temporal windows is often cumbersome or even impossible. Due to their similarity to the well-known language SQL, data stream languages, with substantial extensions compensating for the mentioned shortcomings, are often popular in commercial and open-source CEP products.

Production systems offer a very flexible way to accomplish complex event processing and are very well integrated with existing programming languages. However, the great flexibility comes at the price of working on a low abstraction level as production systems are not primarily intended for CEP. Thus, although successfully applied in the industry, working with production system requires expert knowledge and careful authoring of rules.

Though timed automata provide an intuitive visualization of complex events their expressiveness is limited. They do not directly support aggregation. Negation and even composition of events is cumbersome. To overcome these deficiencies timed automata are usually integrated with languages of other styles.

Logic languages have strong formal foundations, allow an intuitive and convenient specification of complex temporal conditions and account for event data. Moreover, rule based languages can be naturally integrated with conventional reasoning capabilities, eg, to facilitate semantic extensions based on ontologies [Sto+11; TRP12].

Combination of different language styles in one approach allows to benefit from their strengths. This is the main reason why hybrid approaches are most successful in the industry. Hybrid approaches include the introduction of pattern matching into data stream query languages as in Oracle CEP [Orab] and Esper [Esp], the use of composition operators on top of data stream queries [GAC06; CA08], the addition of composition operators to production rules [WBG08], and the combination of production rules and state machines, eg, in TIBCO Business Events [TIB].

One of the main benefits of CEP over conventional database technology is the continuous evaluation of queries and the resulting timely detection of valuable situations by means of complex events. In this way, CEP contributes to substantially improve the reaction time to application relevant incidents and thus facilitates the implementation of event driven applications that rely on time critical reactions. The desire to further increase the effectiveness of time critical reactions and to seize opportunities by mitigating or eliminating undesired future events has even lead to the development of proactive event processing models [EE11; EEF12].

Despite the benefits of the timely execution of reactions based on the detection of complex events, the actual execution of actions is often neglected in current EQLs. The dominant event processing model does not directly include a notion of actions [EN10; CM12b]. Instead, event sinks are used to model actuators executing actions on the reception of certain events that are derived by the EPS and forwarded to them by means of an event based communication infrastructure, such as, an ESB. This is an effective and in many cases sufficient model facilitating many relevant applications that are bases on rather simple reactions or reactions that are merely triggered by events and are executed by dedicated systems not relying on a deep integration with event queries. Another benefit of this model is the abstract view on actions that excludes issues not directly related to the efficient detection of complex events which is still one of the main interests in the community. However, the model effectively prevents the desirable integration of events and actions required to obtain complex actions as they are desirable for EM.

Nevertheless, some EQL provide further means to model actions that interleave the detection of events and the specification of actions more directly, for instance, by incorporating remote procedure calls or by providing an integration of EQLs with general purpose languages. However, in particular in research, these features are less commonly considered for EQL.

3.3.1 *Remote Procedure Calls*

A pragmatic and straight forward way to integrate reactions into EQL is realized by facilitating the specification of remote procedure calls. This can either be realized by integrating the specification of method invocations directly into the EQL, eg, by means of reactive rules specifying the respective method in their head, or by means of an appropriate configuration of the EPSs that is external to the applied query language.

By means of method calls or procedure calls, respectively, it is merely specified which method is invoked when certain events occur but the reaction is specified in a different (often imperative) host language and the actual execution is delegated to another process either on the same or on a remote machine. In addition to the invocation of methods of an imperative language it is also feasible to integrate other (Web based) protocols that enable for instance support of Web services [PKB07; Esp14].

Accordingly, event queries initiating the reaction and programs specifying the reaction are strictly separated and all information that is relevant for the execution of the reaction needs to be passed to the corresponding method on its invocation. And although the basic idea applies to all kinds of languages styles, the specific shape slightly differs.

DATA STREAM LANGUAGES SQL based EQLs can facilitate the specification of procedures in the `SELECT` part of continuous queries with the semantics that the corresponding procedures are invoked whenever the respective query matches.

For instance, the continuous query language provided by Esper [Esp] facilitates the specification of static Java methods in the `SELECT` part of queries.[1]

```
SELECT Reactions.immediateAlarm(area) FROM alarm
```

Whenever an `alarm` event occurs, the query calls the static Java method `immeditateAlarm` and injects the `area` attribute of the matched event as a parameter to the method.

PRODUCTION RULES Production systems used for CEP are often well integrated with conventional general purpose languages (see Section 3.2.3). In addition to the methods intended to alter the working memory, further user defined methods can be included in the head of production rules. This even facilitates the direct specification of simple sequences of actions in the EQL.

For instance, Drools Fusion [Drob] is based on Java and thus facilitates the invocation of user defined static Java methods. Thereby, Java objects stored in the working memory as well as particular attributes can be passed to the invoked procedures.

```
when
  $a: Alarm()
then
  Reactions.immediateAlarm($a.area)
```

1 We assume here and in the following examples that the corresponding method is initially properly imported.

Whenever an `Alarm` is added to the working memory the static Java method `immeditateAlarm` is invoked with the corresponding value of the `area` attribute from the `Alarm` object as its formal parameter.

LOGIC LANGUAGES The rule based nature of logic languages nicely integrates with reactive rules specifying method calls in their head. In particular in the domain of active databases, so called ECA rules are used to obtain reactive behavior and are introduced in more detail in Section 3.3.3.

For instance, XChange^{EQ} [Eck08] integrates reactive rules denoted `RAISE «message and recipient» ON «event query» END`. The recipient, specified by means of the keyword `to`, also determines the transport mechanism which includes calling Java methods and issuing SOAP message requests [W3C07] via HTTP [IET14].

```
RAISE
  to(recipient="Reactions.immediateAlarm", transport="java") {
    area{ var A }
  }
ON
  event e: alarm{{ area{var A} }}
END
```

Whenever an alarm occurs, the XML document or node, respectively, constructed in the rule head is forwarded to the recipient which causes the invocation of the static Java method `immediateAlarm` in this example.

3.3.2 *Integration with Imperative Languages*

Another way to couple event queries with general purpose languages is to integrate event queries into general purpose languages by means of appropriate APIs that facilitate the specification of queries and the reception of query answers in a suited manner. The basic idea is that event queries are directly specified in the source code and answers are provided by means of (infinite) collections that can be iterated over or by subscribing a callback function that is invoked when new events to the corresponding event queries are available. In this way, event queries and the actual reaction, not just the invoked methods, are specified in conjunction and are not distributed over several files or even systems.

A generic interface that facilitates the integration of (database) queries with languages that are based on the .NET framework, is provided by LINQ [PR07]. It is in particular suited to integrate event queries from StreamInsight with C# [KGR12].

```
var query = from e in alarm select e.area;

foreach (long area in query) {
  ...
}
```

Other EQLs and EPSs, eg, Esper [Esp], provide proprietary APIs that facilitate a similar integration of queries and programs. However, in any case, the lack of a deep integration remains as the information that is available to influence the behavior of the reaction only comprises the information that is stored in the payload of the event.

3.3.3 *Event-Condition-Action Rules*

ECA rules originate from active databases where they are used to anticipate and react to changes of the database. They consist of three distinct parts, whereby the action is executed when the event query matches and in the same instant the condition specified by means of a database query holds.

```
ON «event query» IF «database query» DO «action»
```

Due to their origin in databases the action of ECA rules is often concerned with (composite) updates of database relations by means of nested transactions [DHL91; BN09]. However, ECA rules have been successfully applied to model composite workflows by means of rules [KEP00; Bry+06].

Several extensions of ECA rules have been proposed facilitating the specification of more flexible rules. Knolmayer, Endl, and Pfahrer [KEP00] propose ECAA rules that specify additional alternative actions that are executed when the condition in the C part does not hold. Moreover, ECA-LP [PKB07; Pas06] provides an extension of ECA rules by preconditions, postconditions, and alternative actions which are homogeneously integrated with derivation rules in addition to constructs facilitating the asynchronous communication of event messages. Furthermore, in [Beh+06] the process algebra CCS [Mil82], an abstraction of communication between different agents or processes, is incorporated into the action part of ECA rules to facilitate reasoning and verification of reactions formulated by means of CCS.

In addition, database updates specified in the action part of ECA rules have been recently adapted to the requirements of streams by means of a stream-oriented transactional model [WRE11]. Furthermore, generalizations of ECA rules towards reactive rules for the Web have been proposed, for instance by the language XChange [BEP06a; BEP06b], to facilitate updates to data on the Web as a reaction to composite events that are exchanged between interacting Web-based applications.

Besides their relevance for reactivity in event processing, reactive rules, in particular ECA rules, have been extensively studied in the database community. For a more thorough introduction of reactive rules in general and reaction rules on the Web refer to [PK09] and [Ber+07], respectively.

Part II

THE LANGUAGE DURA

4

FOUNDATIONS AND LANGUAGE DESIGN

Before we introduce Dura in Chapter 5, we elaborate a high-level overview of its main building blocks and summarize the basic ideas of its underlying concepts.

Dura is conceived as a uniform and declarative language with a clear separation of dimensions, a lightweight set of well aligned language constructs, and a deep integration of events, stateful objects, and actions. To this end, the language applies a rule-based language paradigm that is well suited for the formalization of complex event queries and naturally integrates with reactive rules to facilitate reactions to detected events. Moreover, Dura uses a pattern based approach for querying and constructing of (complex) events by means of declarative rules.

Dura uniformly integrates the orthogonal concepts events, stateful objects, and actions on a common language level to realize expressive complex event queries and versatile reactive behavior. Moreover, the language design establishes a clear separation of dimensions, an aspect of Event Query Languages (EQLs) that has been prominently identified by Bry and Eckert [BE07]. The design of Dura furthermore avoids implicit assumptions that are hard coded into core language constructs but instead strives for lightweight and orthogonal operators that can be combined in a convenient and flexible manner. It furthermore recognizes time as a first class citizen and facilitates the specification of multiple user defined time lines and versatile temporal constraints that are specified by generic formulas rather than a limited set of temporal operators.

Dura is built on the foundations of the EQL XChange^{EQ} [BE07; Eck08] and consequently some of the fundamental aspects that have proven beneficial for the design of XChange^{EQ}, in particular its rule based nature and the clear separation of dimensions, are adopted and integrated into the design of Dura.

An early draft of Dura is presented in [HBB11] and various aspects related to Dura and its design are discussed in [BHB11; HBB12; BHB12] of which the author of this thesis is either lead author or coauthor.

4.1 DECLARATIVE RULE-BASED REASONING OVER STREAMS

Rule-based languages have their origin in mathematical logic and were applied in artificial intelligence and logic programming before they have been adopted to facilitate reasoning over a stream of events.

Rule-based languages provide an expressive and convenient foundation for EQLs as they promote formal declarative semantics, integrate reasoning capabilities, enable support of ontologies, and facilitate the natural integration with reactive rules [CM10; Ani+10; BE07].

4.1.1 *Reasoning with Rules*

Logic programming is based on the formalism of mathematical logic which has been adopted by general and special purpose languages, such as the well known rule-based language Prolog [CM03] and the related database query language Datalog [CGT89].

A logic program is a set of logical clauses that are classified into facts and deductive rules [CGT89]: Facts are asserting some kind of application knowledge by means of predicates and (composite) terms whereas rules correspond to generic statements formalizing logical consequences that can be drawn from the available knowledge. Deductive rules consist of two parts, the rule body specifying a proposition by means of a (often conjunctive) formula and the rule head specifying the inferences that can be made in terms of derived facts when the proposition in the body holds:

$$\textit{head} \leftarrow \textit{body}$$

Thereby, the conclusions made by several rules are chained together, that is, the predicate specified in the head of a rule can occur in the body of another (or even the same) rule, to enable a better abstraction between rules and to facilitate recursive programs.

Rules can be interpreted as logical clauses reading "if *body* then *head*" which entails a natural and convenient declarative semantics of logic programs. There are, however, other equivalent but substantially different approaches to define the semantics of logic programs [AHV95], which is considered as a very elegant property of logic programming in general.

EXAMPLE The following program computes the reachability relation `t` of a graph `r`, also referred to as transitive closure of `r`.[1]

```
t(X,Y) ← r(X,Y)
t(X,Z) ← t(X,Y) ∧ t(Y,Z)
```

The first rule formalizes that `Y` is reachable from `X` if there is an edge from `X` to `Y` in the graph. Moreover, the second rule states that `Z` is reachable from `X`, if `Y` is reachable from `X` and `Z` is reachable from `Y`. Note how the second rule is recursive as it uses rule chaining to query predicates in its body that also occur in its head. Given the additional set of facts

1 We apply the conventions of Prolog here, which distinguish relations and terms by a small first letter and variables by a capital first letter.

```
r(a,b)
r(b,c)
r(c,d)
```

formalizing a graph by means of the binary relation symbol `r` and the constants `a`, `b`, `c`, and `d`, the program from above determines that `d` is reachable from `b`, that is, `t(b,d)` holds: According to the first rule `c` is reachable from `b` and `d` is reachable from `c`, that is, `t(b,c)` and `t(c,d)` hold. Thus, according to the second rule, `d` is reachable from `b`, that is, `t(b,d)` holds.

RELEVANCE FOR EVENT PROCESSING By applying incremental evaluation on top of a dynamic knowledge base, logic programming is in principle suited for Complex Event Processing (CEP): Informally speaking, facts can be used to model events that are subsequently added to the knowledge base and rules can be used to specify complex event queries that derive complex events whenever their body is satisfied.

Note, however, that this just illustrates the basic idea. Rule-based languages for CEP furthermore need to integrate sufficient support of temporal aspects and the evaluation of queries needs to be substantially adapted to the volatile, continuous, and unbounded nature of streams.

4.1.2 *Data Model*

Depending on the origin and purpose of specific EQLs, the expressiveness of the underlying data model can substantially differ [CM12b]. Events considered by early approaches from active databases do not carry any additional data whereas modern high-level languages support events carrying XML data. In between there are approaches that support tuples and key/value pairs, respectively, and records that have an inherent structure but are not capable of modeling graphs or generic trees.

The data that is relevant in the domain of Emergency Management (EM) is rather simple as it mostly comprises physical parameters provided by sensors and actuators. The structure of the payload of relevant events thus mainly resembles flat tuples [SB10b; SBR11a]. Nevertheless, it is highly desirable for programmers to support some kind of structure in the payload of events, in particular for the specification of derived events. To this end, we strive for a pragmatic compromise between the generic self-describing structure of XML that is desirable for humans and the simplicity of flat tuples that facilitates an efficient storage of events in a database.

Events are represented in Dura by means of Xcerpt terms [BS02] that have were proven to be effective for rule based event query languages, such as, XChange [BEP06b] and XChange^{EQ} [BE07]. The syn-

tactical representation of Xcerpt terms is similar to the one used in mathematical logic: a prefix written label followed by a list of child nodes enclosed with brackets.

```
graph{
  edge{ from{a}, to{b} },
  edge{ from{b}, to{c} },
  edge{ from{c}, to{d} }
}
```

Figure 4.1: An Xcerpt data term

However, data terms that are considered for Dura are restricted to a proper subset of valid Xcerpt data terms. In particular data terms representing graph based semi-structured data, such as XML data, are omitted as they are not further required for the representation of events as discussed above: Data terms that are considered for Dura need to adhere to a fixed schema that is limited in width and depth, so that data terms correspond in their expressiveness to records rather than to generic graphs. Moreover, equally labeled siblings are not permitted and ordered sub-terms (distinguished by square brackets in Xcerpt) cannot be specified.

4.1.3 *Pattern-based Queries*

Xcerpt applies a pattern-based approach to query data terms. Xcerpt query terms specify patterns that resemble the respective queried data terms that are furthermore enriched with variables indicating where data should be extracted. This way of querying data resembles the query-by-example paradigm introduced for the equally named query language QBE [Zlo77].

Consider for instance the following Xcerpt query term

```
graph{{ edge{ from{var F}, to{var T} } }}
```

which resembles the data term from Figure 4.1. The given query term contains the variables `F` and `T` to extract the values at the respective positions and uses double curled brackets to specify that the query is incomplete in width, that is, it matches `graph` terms with at least one `edge` child term instead of exactly one `edge` child term. Accordingly, the given query term matches the data term from Figure 4.1 three times, once for each `edge`, yielding three different bindings for the two variables `F` and `T`, namely, $\{\mathsf{F} \mapsto \mathtt{a}, \mathsf{T} \mapsto \mathtt{b}\}$, $\{\mathsf{F} \mapsto \mathtt{b}, \mathsf{T} \mapsto \mathtt{c}\}$, and $\{\mathsf{F} \mapsto \mathtt{c}, \mathsf{T} \mapsto \mathtt{d}\}$.

Xcerpt query terms can include further advanced query constructs that are specifically designed to cope with the semi-structured and

graph based nature of data terms. This includes for instance constructs to formulate queries that are incomplete in depth, to facilitate negated sub-term queries, or to specify restrictions on the values that are to be bound by variables. For a thorough introduction of those and further constructs refer to [BS02; Sch04].

However, because only a subset of Xcerpt data terms is considered for Dura, constructs that are related to the rich and semi-structured nature of data terms are omitted form Dura query terms. In result, query terms merely correspond to data terms that are enriched with variables. Other constructs are not required and thus not supported, in particular the order of sub-terms cannot be qualified and queries are always considered to be incomplete in width.

In this way, the desirable pattern based nature of queries is maintained and backward compatibility to Xcerpt is (almost completely) ensured while data terms can be efficiently mapped to tuples of a relational database.[2]

4.2 Full Acknowledgment of Orthogonal Concepts

Dura is designed to combine aspects that are related to complex event queries with the specification of reactions that are realized by means of external actuators. To this end, Dura provides uniform integration of the three orthogonal concepts events, stateful objects, and actions: volatile events communicate observed changes of relevant parameters; stateful objects are the persistent counterpart to volatile events that facilitate a notion of state; and actions realize interactions with external components.

Thereby, the integration of all three notions is desirable to obtain a convenient model for reactivity in event processing. This has been similarly found by Schmidt, Anicic, and Stühmer [SAS08] who propose contexts as an convenient abstraction for programmers to specify reactions that are tailored to a certain situation.

Events Events are commonly conceived as indication of incidents or changes of parameters that are relevant for some application logic [LS11]. In particular the detection of complex events corresponds to the recognition of relevant high-level situations that are of interest for the considered application that cannot be identified by looking at single events in isolation.

Stateful objects Stateful objects are intended to persist information in a non-volatile manner as it is desirable to model states. To this end, stateful objects represent data that can be updated in a non-destructive manner. Thereby, queries against stateful objects yield an

2 Merely double curled brackets need to be substituted for single curled brackets to transform a Dura query term into an equivalent Xcerpt query term.

answer immediately after the corresponding data begins to be valid whereas event queries yield an answer after the corresponding event is detected, that is, after it ended.

ACTIONS Actions are executed in response to the detection of a complex event and with respect to the current situation that is represented by means of stateful objects. Thereby, they intentionally cause observable side effects either by means of external actuators that affect physical values or by internally changing the values of stateful objects.

4.3 DEEP INTEGRATION IN A UNIFORM LANGUAGE

Events, states, and actions are clearly intended for different purposes and it is thus desirable to keep them separate on a language level. Yet, adequate and expressive event detection and reactions can only be realized by means of a combination of all three concepts: The appropriate detection of events and the execution of suitable reactions depends on the current context represented by means of stateful objects; in turn, updates of stateful objects are caused by the detection of events and realized by means of internal actions; events queries are required for the specification of composite actions to determine whether their intended high-level goal has been achieved; and alternative execution branches in composite reactions are guarded by queries for events and stateful objects.

To account for those kinds of dependencies, the design of Dura strives for a deep integration of all three concepts in a uniform language: Event queries can naturally integrate queries for stateful objects and complex actions can use event queries to specify the status of their execution and to facilitate conditions for different execution branches beyond temporal dependencies. In this way, the different aspects of events, stateful objects, and actions can effectively be combined to realize expressive complex event queries and complex actions. In addition, the integration of queries for events and stateful objects gives rise to a natural extension of Event-Condition-Action (ECA) rules that eliminates the inflexible separation of the event and condition part not adequate for queries to dynamic stateful objects.

In contrast, approaches lacking a deep integration of events, states, and actions loose expressiveness. If, for instance, reactions are realized by means of remote procedure calls, the information gathered by the Event Processing System (EPS) in form of complex events and stateful objects is not directly accessible to the entity executing the action. As a consequence, the status of the execution of actions formalized by means of complex event queries cannot be easily incorporated into the execution of composite reactions.

4.4 TIME AS A FIRST CLASS CITIZEN

Another fundamental basis for the design of Dura is to avoid hard coded defaults that cannot be adapted by programmers according to their specific needs. This most prominently includes an adjustable time of events that is furthermore included in their payload. By contrast, in many other languages, the time of events is an extrinsic part of events that is solely governed by the EPS and not accessible to external components.

In Dura, the time of events is stored as an attribute in the payload of events. As a consequence, the time of derived events, which is for convenience implicitly determined by the EPS, can be adjusted to the requirements of the programmers by providing an alternative more suited value. Moreover, events are not limited to carry a single time in their payload and thus events can be associated with multiple times representing different time lines. This is in particular useful to incorporate events generated by simulators that need to distinguish multiple times, eg, the reception time of the event by the EPS and the simulation time indicating when the event is deemed to occur according to the simulation. In addition, the time of events is naturally available for external components that receive derived complex events.

4.5 EXPLICIT SPECIFICATION OVER IMPLICIT ASSUMPTIONS

The design of Dura favors explicit specifications over implicit assumptions to obtain a clear and accessible semantics for queries and rules, respectively. However, this results in the deliberate omission of language constructs that are prominently featured in other approaches.

Some EQLs, such as the rule-based language TESLA [CM10], provide constructs for event selection, eg, to select the most recent event matching some kind of condition. Although this seems convenient to realize particular use cases, it is often neglected that events, in particular derived events, may occur at the exactly same instant and that there is hence no unique last or first event. This problem is circumvented in TESLA by additional conditions that select the events with the highest key in case of ambiguities. This, however, substantially jeopardizes the considerable formal semantics of TESLA, as the determination of the key is subject to runtime effects that are abstracted away in the formal semantics.

Note that the functionality of selecting the most recent event matching some kind of conditions can indeed be realized by means of appropriate rules in Dura. However, instead of providing syntactic sugar with integrated implicit assumptions, programmers are deliberately forced to explicitly specify the intended behavior by means of appropriate conditions and rules.

In the work on XChange^{EQ} [BEo7; BEo8b; Ecko8] different aspects of EQL, so-called dimensions, have been identified that need to be adequately covered to facilitate the specification of complex events by means of expressive query patterns. Those dimensions are data extraction, event composition, temporal and other dependencies, and event accumulation. Moreover, Bry and Eckert [BEo8b] stress that beyond a clear coverage of the identified dimensions, a clear separation of the dimensions is inevitable to obtain a high expressiveness and ease of use that cannot be accomplished by monolithic operators combining aspects of different dimensions.

In the spirit of their work, the clear separation of concerns is retained for the design of Dura. To this end, the language model of XChange^{EQ} is substantially extended to additionally distinguish stateful objects and complex actions that are not acknowledgment and integrated as crucial concepts in XChange^{EQ}. Moreover, the four identified dimensions of event queries are thoroughly revised and extended to a total of six dimensions separating aspects that are combined in the primal four dimensions more clearly.

4.6.1 *Dimensions of Complex Events*

The dimensions of complex events cover different aspects relevant for the specification of complex events by means of complex event queries. The six dimensions that are retained for Dura are data extraction, event composition, temporal and other dependencies, grouping and aggregation, value definition, and event construction.

DATA EXTRACTION Events carry a payload of data that provides application dependent information. Naturally, the derivation of complex events and the initiation of actions greatly depends on the carried data.

Accordingly, relevant portions of the carried data need to be extracted from the event and bound to variables to facilitate the specifications of appropriate conditions, eg, by relating the extracted values to values obtained from other events.

EVENT COMPOSITION Although the occurrence of single events arguably provides relevant information, it usually takes the occurrence or absence of multiple events that are in some way related to describe a valuable high-level situation. Accordingly, means to compose higher-level events from patterns of multiple events are a mandatory integral part of versatile EQLs.

Note, however, that if a clear separation of dimensions is granted, three operators for conjunction, disjunction, and negation of events

are sufficient for the specification of arbitrary event patterns. Other common operators, eg, operators specifying sequences of events, are realized by means of additional temporal conditions that are orthogonal to the composition of events.

TEMPORAL AND OTHER DEPENDENCIES Time is an important aspect of stream based applications. Due to the volatile and unbounded nature of streams, the occurrence of events within a composite query pattern is often sensitive to timing and order and thus the temporal dimension of complex events deserves particular attention.

Accordingly, in addition to the composition of several events within one query, EQLs require means to establish temporal dependencies between events contributing to the detection of a complex event. Moreover, dependencies on the payload of events are required to specify filters or to correlate several events.

GROUPING AND AGGREGATION It is highly desirable to provide means that facilitate the derivation of an abstracted and concise summary encompassing the information of multiple events of the same type. To this end, events need to be separated into different groups, eg, according to their time, so that the values of events of each group can be cumulated into a single aggregated value.

Aggregation is orthogonal to event composition as it is intended to combine the information of multiple events into a single aggregated value whereas using event composition each matching event contributes to one particular query answer.

EVENT CONSTRUCTION The evaluation of event queries eventually leads to the detection of complex events which entails the construction of a (potentially virtual) complex event. To this end, the attributes of derived events need to be determined by means of fixed values, ie, literals, or derived values.

VALUE DEFINITION Derived values determined by means of expressions composed from variables are required for the construction of higher-level knowledge that combines multiple values in a non-trivial manner.

To facilitate the specification of expressive sub-queries the definition of derived values based on the attributes of queried events needs to be clearly separated from other dimensions. This also applies for the determination of aggregated values.

4.6.2 *Dimensions of Stateful Objects*

As it is subsequently elaborated in Section 5.3, it is highly desirable to abandon the strict separation of reactive rules, in particular ECA

rules, into a separate event and condition part. Instead it is desirable to deeply integrate queries for events and stateful objects in a uniform manner.

Due to the uniform representation of events and stateful objects in Dura and the integration of queries thereof, the dimensions of complex events and stateful objects are almost identical and do not require any further adaptations. Merely the construction of events does not have an equivalent representation for stateful objects as views on stateful objects are currently not considered in Dura.

4.6.3 *Dimensions of Complex Actions*

Naturally, the dimensions of declarative event queries and imperative complex actions substantially differ. There are at least five dimensions of complex actions that are based on physical actions executed by external actuators, namely, action invocation, action composition, temporal dependencies, temporal assertions, and execution status.

To obtain complex actions that are tailored to physical actions and provide a high expressiveness and ease of use, a good coverage of these dimensions is desirable. Moreover, the observations that suggest a clear separation of dimensions of complex events apply likewise for complex actions.

ACTION INVOCATION Action invocation is the counterpart to data extractions for events. However, instead of querying events and extracting relevant data from their payload, values that have been previously bound to variables need to be injected to actions to determine their formal parameters.

ACTION COMPOSITION High-level reactions are implemented by means of (atomic or complex) sub-actions that are executed in combination to achieve a certain higher level goal that cannot be achieved by single and more basic actions. The composition of actions is therefore crucial to obtain useful reactions that exceed the capabilities of single actuators.

Similar to the composition of events, the composition of actions requires merely a few constructs. In fact, provided that a clear separation of concerns is implemented, only one construct for the basic collection of several actions is required to sufficiently cover this dimension. Other operators, such as, operators specifying a sequence of actions, are realized in Dura by means of a combination of composition and temporal dependencies.

TEMPORAL DEPENDENCIES Temporal dependencies are crucial to specify the timing between several physical actions that are executed in combination. In contrast to internal actions, composite actions that

are intended to realize a certain effect in the physical world often require a certain and precise timing entailing a particular order between actions as otherwise unintended side effects may occur.

To be suitable for external actions with contingent results, temporal dependencies must furthermore discriminate between success and failure of actions to allow different reactions based on the result of preceding actions.

TEMPORAL ASSERTIONS Temporal assertions state temporal conditions between actions that are expected to be satisfied when the action is actually executed according to its temporal dependencies. However, in contrast to temporal dependencies, these conditions have no effect on the execution of the composite action. It is merely verified at compile time that the conditions will be satisfied when the action is executed

The separation of temporal dependencies and temporal assertions strongly contrasts with approaches that specify constraints on actions that should be satisfied by an intelligent scheduling of actions.

EXECUTION STATUS Physical actions are means to an end that are executed to indirectly affect some physical properties. However, feedback on their success often cannot be inferred from the feedback that is provided by the corresponding actuator, in particular in case of complex actions that are intended to achieve some high-level goal. Instead, the achievement of their goal needs to be verified by means of sensors that are capable of measuring whether the desired effect has been achieved.

Accordingly, means for the specification of the status of actions must be expressive enough to specify generic event patterns that infer the status of actions from related sensor messages.

5 SYNTAX AND INFORMAL SEMANTICS

The language model of Dura is based on the three orthogonal concepts events, stateful object, and actions that are clearly distinguished in the language: volatile events to model changes of parameters, stateful objects to model persistent data that is modified in a declarative non-destructive manner, and actions to realize intended side effects. At the heart of Dura are declarative composite event queries integrating queries for events and stateful objects and imperative action specifications integrating event queries and actions.

The design of Dura strives for a clear separation of concerns which manifests in expressive sub-queries that are composed from well conceived orthogonal operators covering the relevant aspects of event queries. Moreover, the systematic separation of concerns promotes expressive sub-actions that integrate composite event queries so that the specification of complex actions becomes independent from reactive rules which are merely required to trigger actions.

In addition, Dura applies deductive rules to specify complex events that are derived based on composite queries matching the stream of events. They serve as an convenient and natural abstraction mechanism, eg, to facilitate hierarchies of events. Moreover, Dura applies reactive rules to initiate (complex) actions in response to detected events. They mark the transition between the declarative world of events and stateful objects and the imperative world of actions.

This chapter elaborates the syntax and semantics of Dura by means of informal examples. A formally precise grammar of Dura is contained in Appendix A.

5.1 COMPLEX EVENTS

According to Luckham and Schulte [LS11], an event is anything that happens or is contemplated as happening and hence events are ubiquitous in modern critical infrastructures. Events are generated for instance by various physical sensors and actuators scattered throughout the entire infrastructure so as to observe and control it. Moreover, events are caused by humans, eg, by operators who enter information they observe into the system and by passengers over emergency phones, and by fast computable simulations that generate simulation events which estimate the likely development of the conditions within the next couple of minutes.

Events are a substantial foundation for Emergency Management (EM) as they describe the current conditions of the infrastructure and

thus serve as an important basis for reactions and decisions. However, the sheer number of raw events that are caused by the equipment of the infrastructure has reached the limit that can be reasonably handled by humans, in particular during emergencies. Operators need concise and reliable interpretations of the current situation in the infrastructure that is derived from the incoming sensor readings and can be captured without having to consider all events that actually occurred. Therefore, modern EM needs means that filter, correlate, and interpret the incoming data to derive a representation of the ongoings in a way that is suitable for human operators.

Accordingly, Complex Event Processing (CEP) can substantially improve EM as it is today, as it provides means to detect high level situations within the incoming stream of events by means of complex event queries that are evaluated in a continuous and timely fashion.

5.1.1 *Representation of Events*

Each event has a unique type, carries a payload of data, and is associated with a key uniquely identifying each event instance and a time interval over a continuous time domain, the so-called reception time. Moreover, each event type is characterized by a schema describing the type and structure of its payload.

The payload of events resembles records which can be efficiently stored and processed inside conventional databases but yet provide the desirable flexibility to structure the data in a way suited for EM purposes. Although other Event Query Languages (EQLs) are capable of querying semi-structured data provided by generic XML messages, it is not mandatory to cover the full flexibility of XML to suite the events considered by EM [SB10b; Vra+10; SB10a]. The structure of those events is much more basic than events that are typically observed, eg, in distributed and loosely coupled web applications that provide content which exploits the expressiveness of XML.

Nevertheless, events and event queries are expressed in Dura by means of Xcerpt terms as they have been proposed in [BS02] to represent and query semi-structured data. However, as the structure of events considered in Dura is restricted to records as it is discussed in Section 4.1.2, constructs that account for the rich structure of generic terms are omitted. In this way, a pattern-based query approach that is proven for rule [BS02; BEP06b] and event query languages [BE07] is suitable for Dura while an efficient representation of events is preserved.

EXAMPLE For convenience, events are represented in a linearized form that is similar to the representation of terms in mathematical logic or logic programming. The label of a node is written prefix followed by a (unordered) list of its children enclosed with curly braces.

```
temp{ area{31}, value{12.7}, sensor-id{8191} }
```

The preceding data term represents an event reporting about the temperature within a certain area. The event is of type `temp` and has the three atomic attributes `area`, `value`, and `sensor-id`.

TIMES OF EVENTS By default, each event is associated with a time interval over a continuous time domain, the so-called reception time.

The reception time of basic events corresponds to the instant the event is received by the Event Processing System (EPS). Accordingly, the reception time has the same begin and end and thus degenerates to a time point. For derived events, the reception time comprises the reception time intervals of all events that contributed to its detection, that is, the begin and end of derived events corresponds to the least beginning and largest ending of all positively queried events, respectively.

In addition to the reception time which is implicitly added to all events, events can carry any number of application dependent, user defined, times. Other relevant times are for instance the detection time and the simulation time. The detection time corresponds to the instant an event is actually derived and thus becomes visible for external subscribers whereas the simulation time describes when a simulation event is deemed to occur according to the simulation. How to incorporate user defined times into the schema of events and how to adapt the default values for existing times is further elaborated in Section 5.2.5.

EVENT SCHEMA The event schema specifies for each event type the structure and fashion of their payload.

Listing 5.1: Schema of `temp` Events

```
temp{ area{long}, value{double}, sensor-id{long} }
```

Naturally, the schema is specified by means of terms that contain types in place where the actual data of the event will be located and resemble tree-like structures. The leaves of a schema term correspond to basic types and the internal nodes specify labels that describe the data they contain. Thereby, labels of siblings need to remain distinct. There are nine different basic types available in Dura, namely *int*, *long*, *float*, *double*, *boolean*, *string*, *duration*, *timestamp*, and *identifier*.

As already mentioned, this representation corresponds to the representation of Xcerpt terms [FBS07; BS02]. But in contrast to generic Xcerpt terms that resembles a graph structure, references to nodes as well as equally labeled or ordered siblings are not permitted in Dura.

Accordingly, events and their schemas are represented by terms that correspond to trees with a fixed width and depth.

IMPLICIT ATTRIBUTES Each event is associated with a unique key and its reception time. To be easily accessible by common event queries and external components that operate on derived events, both properties are represented in the payload of events. To this end, the schema of events is implicitly extended with two attributes, namely, the atomic attribute `id{`*`identifier`*`}` for its key and the composite attribute `reception-time{ begin{`*`timestamp`*`}, end{`*`timestamp`*`} }` for its reception time.

Accordingly, the schema of the temperature event from above actually corresponds to the schema in Listing 5.2 which additionally contains the implicit attributes of the event.

Listing 5.2: Schema of `temp` Events with Implicit Attributes

```
temp{
  id{identifier},
  reception-time{ begin{timestamp}, end{timestamp} },
  area{long}, value{double}, sensor-id{long}
}
```

Note that the values for the key and the reception time attribute are implicitly determined by the EPS, unless they are explicitly specified by the programmer as it is discussed in Section 5.2.2. Moreover, for the sake of simplicity, implicit attributes are often omitted from the following examples.

5.1.2 *Atomic Event Queries*

Events are queried by means of a pattern based approach known from existing query languages, that is, the query pattern resembles the data of the event and variables are specified in the pattern where data should be extracted. To this end, event queries are expressed by means of simplified Xcerpt query terms [FBS07] that do not contain formulas such as negation or optionality.

Event queries are specified by means of query patterns which are preceded by the keyword *`event`* followed by an alphanumeric name, the so-called event identifier. The event identifier is often used in composite queries to concisely refer, eg, to the reception time of the queried event. Query patterns do not need to be completely specified, that is, attributes that are not relevant for a certain query can be omitted. Moreover, attributes in the query pattern can be specified in an arbitrary order that does not need to correspond to the positioning of the attributes in the schema.

EXAMPLE The query in Listing 5.3 matches temperature events and extracts the values of the area and the temperature from their payload.

Listing 5.3: An Atomic Event Query

```
event e: temp{ value{var T}, area{var A} }
```

When applied to a stream consisting of the temperature event

```
temp{ area{31}, value{12.7}, sensor-id{8191} }
```

the given query yields a single substitution binding `A` to `31` and `T` to `12.7`.

IDENTIFYING EVENTS There are actually three different ways to identify or refer to events which must not be confused: The *type*, eg, `temp`, corresponds to a name that identifies a class of events. The alphanumeric *identifier*, eg, `e`, identifies instances of matched events within the lexical context of a query. And finally, the *key* identifies event instances within the entire event stream.

Accordingly, an *identifier* corresponds to a static reference whereas a *key* corresponds to a dynamic reference. Note that hence two (syntactically) different event identifiers, eg, `e` and `f`, may actually refer to the same event instance at runtime.

REFERRING TO ATTRIBUTES OF EVENTS The attributes of events can be referenced in two different manners: either by means of variable definitions specified in the query term of an atomic event query or by means of a path that combines an event identifier with labels separated by periods pointing to the respective attribute.

For instance, in Listing 5.3 the attribute `value` of the `temp` event can either be referred by means of the variable `T` or by means of the path `e.value`. Accordingly, the notions variables and paths are also referred to as references.

Note that due to the restrictions that apply for the structure of events, all attributes are actually distinguished by a unique path. Therefore, advanced path query languages, such as, XPath [W3C10], are not required here.

5.1.3 *Composite Event Queries*

In general, the main goal of CEP is to extract higher level knowledge from a stream of basic events that cannot be recognized by just looking at single basic events. To this end, the occurrence or absence of several events in the stream needs to be correlated to recognize valuable situations or events. Accordingly, the capability to relate several queries in a composite query is crucial for every EQL.

Several event queries are combined by means of the operators `and`, `or`, and `not`. Due to a clear separation of query dimensions which has been proposed in [BE07; Eck08], these operators are merely combining several queries into a composite query. They do not, however, specify temporal and other dependencies between events and the data they carry, as such dependencies can only be specified in a separate `where` part that supplements composite and simple queries.

CONJUNCTIONS Conjunctive event queries are specified in Dura like in XChange^{EQ} by means of the n-ary operator `and` that contains multiple simple or composite event queries. They resemble joins from traditional databases that are incrementally evaluated and generate new variable bindings whenever further matching events occur in the event stream.

A conjunctive query matches whenever all contained event queries match the stream of events. Thereby, variable definitions that occur in different sub-queries and specify the same name imply a join between the attributes they refer to. Event queries in Dura are furthermore purely declarative and there is no consumption or absorbance of events that are matched by a query. In consequence, one event of the stream can be matched by multiple queries and beyond that it can be matched in several different manners by a same composite query.

Conjunctive event queries in Dura may be temporally unrestricted, as opposed to many other EQLs which require that all matching events occur in a predefined and finite time window. Although the evaluation of temporally unrestricted queries may require an indefinite amount of memory which may cause problems at runtime, they have shown to be valuable if applied carefully. For instance, queries that aggregate values between the occurrence of two events, eg, the begin and end of an emergency, can hardly be restricted by a finite time window, as the duration of the emergency is unknown in advance. Furthermore, queries that determine stateful information usually also cannot be restricted in such a manner, which is thoroughly elaborated in Section 11.2.

Listing 5.4: A Conjunctive Event Query

```
and{
  event e: temp{ area{var A}, value{var T} }
  event f: smoke{ area{var A}, conc{var C} }
}
```

The query in Listing 5.4 is a valid conjunctive query, although it does not relate the time of the contributing sub-queries. It matches whenever a temperature and a smoke event occur in the same area. Notice the join between the area attribute of both events which is caused by the use of the variable `A` in both sub-queries.

Note that, due to the temporally unrestricted nature of the given query, temperature and smoke events that occurred apart for any length of time, eg, several days or weeks, will still match the query. This seems undesirable for this particular kind of query as one is only interested in events that occur in quick succession. However, due to the clear separation of query dimensions, the conjunction specifies merely the composition of events but does not imply any temporal dependencies between events which is further addressed in the next Section 5.1.4.

DISJUNCTIONS On a syntactical level, disjunctive queries closely resemble conjunctive queries. They are specified by means of the n-ary operator `or` and are as well subject to incremental evaluation. But in contrast to conjunctive queries, disjunctive queries match the stream of events whenever a single sub-query matches the stream. Accordingly, the sub-queries of disjunctive queries are independent from each other and in consequence commonly named variables do *not* specify a join condition.

Listing 5.5: A Disjunctive Event Query

```
or{
  event e: uncertain-fire-alarm{ area{var A} },
  event f: potential-overcrowding{ area{var A} },
  event g: sos-telephone-call{ area{var A} }
}
```

The query from Listing 5.5 matches when an `uncertain-fire-alarm`, `potential-overcrowding`, or `sos-telephone-call` event occurs.

NEGATION Negated queries are applied to query the absence of events in the stream and are specified by means of the `not` operator. They are always used in conjunction with conjunctive queries and thus resemble anti-semi-joins known from traditional database systems [AHV95]. Accordingly, the negations that are specified with this operator does not correspond to the classical negation, but instead it corresponds to negation as failure [She85] which is commonly applied in logic programming and deductive databases.

Negated queries are always specified as part of a conjunctive query, whereby the conjunctive query only matches when its non-negated sub-queries match and at the same time its negated sub-queries do *not* match. Similar to conjunctive queries without negations, variable definitions specifying the same name imply a join condition so that the values of the attributes they refer to need to be equal in order to result in a valid match of the entire query. Note that composite even queries composed from conjunctions and disjunctions can be negated.

Listing 5.6: A (Non-Evaluable) Negated Event Query

```
and{
  event e: uncertain-fire-alarm{ area{var A} },
  not event f: fire-alarm{ area{var A} }
}
```

The query in Listing 5.6 contains a negated query for fire alarms. It matches uncertain fire alarms that did not eventually evolve into a proper fire alarm. As, by design, the composition operators of Dura do not constraint the time of matching events, the negated query matches fire alarms that occur at an arbitrary time.

However, due to the conceptually infinite streams, temporally unbounded negated queries, as the one from Listing 5.6, cannot be evaluated correctly in finite time, as at any point only a finite prefix of the entire stream has been obtained by the EPS. This issue can be resolved by adding appropriate temporal conditions to the query which is further discussed in Section 5.1.4 and Section 5.2.5.

5.1.4 *Temporal and other Conditions*

Due to the volatile and temporal nature of events and the conceptually unbounded event streams, temporal conditions have a specific role for complex event queries. Queries are usually only interested in current portions of an event stream to derive valuable insights in an event driven manner. As a consequence, event queries need means to restrict the time of events to match only situations that consider current events.

Temporal conditions between events are specified by means of generic formulas built from potentially nested conjunctions, disjunctions, and negations of inequations that relate time points of events in a temporal fashion. This allows to specify arbitrary temporal dependencies between events and the programmer is not restricted to a set of predefined and hard coded operators. Of cause, there is also syntactic sugar in Dura to specify often used relations in a concise manner. However, all of those relations can be equally specified by means of conventional formulas.

Conditions on the data are specified in a similar manner but instead of time points the values extracted by means of variables are related. In this way, the values of several events can be correlated or events can be filtered according to the data they carry.

All (temporal and other) conditions that are relevant for a certain (composite) query are specified in a separate `where` part that is amended to the query. Thereby the collection of the `where`, `let`, and `group by` parts, which will be subsequently introduced, associated with a query is also referred to as query supplement.

EXAMPLE Temporal and other conditions are specified in Dura by means of (in)equations in a separate `where` part that is associated either with an atomic or a composite event query.

Listing 5.7: Conditions of a Query

```
and{
  event e: temp{ area{var A}, value{var T} }
  event f: smoke{ area{var A}, conc{var C} }
} where {
  end(e) <= end(f) + 2min, end(f) <= end(e) + 2min,
  or{ var T > 400, var C > 0.2 }
}
```

The conditions in Listing 5.7 constrain the values of the variables `T` and `C` to exceed 400 degrees or 20 percent. Moreover, they specify that matching temperature and smoke events must occur within two minutes.

ATOMIC EXPRESSIONS Atomic expressions are built from literals, such as, strings (`"celsius"`), numbers (`23`, `42.5`), durations (`5ms`, `1min 17sec`), constants (*const* `n`), paths (`e.value`), and variables and identifiers (*var* `A`, *event* `e`). For the sake of readability, the prefixes of constants, variables, and identifiers can be omitted in Dura expressions.

Moreover, the functions `system-time.now()` and `sequence.next()` are available which yield the current time at their evaluation and the value of a monotonically increasing sequence of long values, respectively.

INTERVAL EXPRESSIONS Time intervals are specified by means of the value constructor `[«expression»,«expression»]` which takes as an argument two expressions of type timestamp. Moreover, time intervals can be obtained by means of the function `time` which takes as an argument an event identifier and returns the reception time of the corresponding event.

The functions `begin` and `end` operate on time intervals and return their corresponding upper and lower boundaries. For convenience, both functions can also be applied to event identifiers whereby they return the begin and end of the reception time of the corresponding event. This convenience notation was actually already used in Listing 5.7. In addition, Dura provides functions on intervals that resemble the operators that are used to construct relative timer events in XChange^{EQ} [Eck+11b; Eck08; BE07]. These functions, specified in Table 5.1, take as parameter a duration and a time interval or an event identifier and compute a new time interval. However, in Dura these functions are just syntactic sugar for more basic expressions and can thus be omitted from the language, whereas the corresponding operators of XChange^{EQ} are a fundamental part of the language.

`extend(e, d)`	≡	`[begin(e), end(e) + d]`
`shorten(e, d)`	≡	`[begin(e), end(e) - d]`
`extend-begin(e, d)`	≡	`[begin(e) - d, end(e)]`
`shorten-begin(e, d)`	≡	`[begin(e) + d, end(e)]`
`shift-forward(e, d)`	≡	`[begin(e) + d, end(e) + d]`
`shift-backward(e, d)`	≡	`[begin(e) - d, end(e) - d]`
`from-end(e, d)`	≡	`[end(e), end(e) + d]`
`from-end-backward(e, d)`	≡	`[end(e) - d, end(e)]`
`from-begin(e, d)`	≡	`[begin(e), begin(e) + d]`
`from-begin-backward(e, d)`	≡	`[begin(e) - d, begin(e)]`

Table 5.1: Translation of Interval Functions to Expressions

COMPOSITE EXPRESSIONS Composite expressions are built by applying arithmetic functions, such as, the binary and infix written functions `+`, `-`, `*`, `/`, and `**` or the n-ary functions `least` and `greatest`, to atomic or composite expressions.

Moreover, Dura provides the binary functions `floor` and `ceil` which take as arguments a timestamp and a duration and yield a timestamp that is rounded to the nearest value above or below the given one according to the precision specified by the given duration. For instance, the expression `floor(ts, 1min)` omits the seconds and milliseconds from the timestamp `ts`.

In addition, the type of expressions can be adapted by means of unary prefix written cast functions whose name corresponds to the type the expression should be cast to. Eg, the expression *`long`*`("1729")` yields the long value `1729`.

Other than those functions, in particular user defined functions, cannot be used in Dura to form (composite) expressions. Moreover, expressions are statically typed and their correct typing is verified at compile time.

FORMULAS Formulas are build from (composite) expressions by means of the relations `!=`, `<`, `<=`, `=`, `>`, and `>=`. Furthermore, composite formulas are obtained by composing several formulas by means of the n-ary sentential connectives `and` and `or` and the unary connective `not`. Note that thereby the negation corresponds to the classical negation from mathematical logic in contrast to the negation used in event queries which corresponds to negation as failure from logic programming.

Formulas are specified in the `where` part of queries that is either associated with atomic queries or with composite queries. For convenience, the keyword `and` can be omitted from the outer most formula.

QUALITATIVE TEMPORAL RELATIONS Allen's thirteen relations [All83] identify thirteen distinct relations on temporal intervals that are convenient for domains where temporal information is imprecise and relative. They are commonly used in EQLs to specify qualitative temporal dependencies between events.

Allen's relations are usually specified in conjunction with event identifiers to constrain the time of the corresponding events. They are, however, only syntactic sugar for composite formulas built from more basic inequations. Table 5.2 contains Allen's relations and their corresponding representation as basic formulas. However, the inverses of the given relations, namely, *after*, *contains*, *overlapped-by*, *met-by*, *started-by*, and *finished-by*, are omitted as they are directly related to the given relations, eg, `e` *after* `f` corresponds to `f` *before* `e`.

`e` *before* `f`	≡	`end(e) < begin(f)`
`e` *meets* `f`	≡	`end(e) = begin(f)`
`e` *overlaps* `f`	≡	`and{ begin(e) < begin(f), begin(f) < end(e), end(e) < end(f) }`
`e` *during* `f`	≡	`and{ begin(f) < begin(e), end(e) < end(f) }`
`e` *starts* `f`	≡	`and{ begin(e) = begin(f), end(e) < end(f) }`
`e` *finishes* `f`	≡	`and{ end(e) = end(f), begin(e) < begin(f) }`
`e` *equals* `f`	≡	`and{ begin(e) = begin(f), end(e) = end(f) }`

Table 5.2: Translation of Allen's Relations to Formulas

Note that the relations can be used in arbitrary combination of other relations and sentential connectives. In particular, even disjunctions of relations can be specified in Dura.

QUANTITATIVE TEMPORAL RELATIONS Besides the qualitative relations from above, Dura provides two quantitative relations that restrict the time window in which matching events may occur, namely, *within* and *apart-by*. Within is an $(n+1)$-ary relation taking as arguments $n \geqslant 1$ event identifiers and a duration. It specifies that the given events occur within a given time window denoted by means of a non-negative duration. Apart is a ternary relation that takes as arguments two events and a durations. It specifies that the given events occur at least the given duration apart from each other.

Similar to Allen's relations, the quantitative relations are just syntactic sugar, that is, the relation `{`e_1`, ..., `e_n`}` *within* `d` translates to

```
greatest(end(e_1), ..., end(e_n))
  - least(begin(e_1), ..., begin(e_n)) <= d
```

and `{e, f}` *apart-by* `d` translates to

```
or{ begin(f) - end(e) >= d, begin(e) - end(f) >= d }
```

FLEXIBILITY OF TEMPORAL CONDITIONS Because temporal conditions are specified by means of generic expressions and formulas that incorporate the time of queried events, Dura provides a great flexibility for the specification of temporal conditions. Temporal conditions can, for instance, specify time windows that lie in the future

```
where { e during shift-forward(f, 2min) }
```

are non-convex

```
where {
  e during extend(f, 2min),
  not{ e during from-end(f, 1min) }
}
```

or are defined relative to multiple events

```
where { end(f)-1min < begin(e), end(e) < end(g)+2min }
```

In addition, temporal conditions cover all thirteen of Allen's relations and beyond that they can also specify arbitrary combinations thereof by means of conjunctions, disjunctions, and negations.

5.1.5 *Data Definition*

Besides the filtering of values in the `where` part of queries, new values can be defined in a distinct `let` part that is associated with queries. Thereby, multiple `where` and `let` parts can be associated with the same query to realize a filtering of newly derived values.

The `let` part contains one or more variable definitions and has the following structure

```
let { «variable» = «expression», ... }
```

whereby the names of newly defined variables must be unused.

EXAMPLE In the query in Listing 5.8 the temperature readings provide the temperature in degrees Fahrenheit instead of degrees Celsius. To filter events that report about a temperature above 400 degrees Celsius, the provided value is first of all converted to degrees Celsius and bound to the variable `Tcel` by means of a `let` part and subsequently filtered to match only values above 400 degrees Celsius.

Listing 5.8: Data Definition

```
event e: temp-value{ area{var A}, value{var Tfah} }
  let { var Tcel = (var Tfah - 32)/1.8 }
  where { var Tcel > 400 }
```

Note that in this particular example the query can be simplified by omitting the `let` part and by adapting the formula in its `where` part to

`(var Tfah - 32)/1.8 > 400`. However, in more elaborated examples defining new variables is desirable, as the bound values can be reused in various parts of the query, eg, in the `where` part of other sub-queries and in the query head.

5.1.6 *Grouping and Aggregation*

Grouping and aggregation are highly desirable for EM and for CEP in general. By their means, values obtained from several events collected, eg, in a certain time window, can be combined using aggregation functions such as average, minimum, maximum, etc. This is particularly useful to realize a smoothing of noisy sensor readings and to obtain concise and abstract indicators for the conditions of the infrastructure, eg, the moving average of the temperature within the last couple of minutes, that is suitable for the interpretation by humans in contrast to the vast number of individual events which contributed to the average value.

```
group by { «variable, identifier, or path», ... }
aggregate { «variable» = «aggregation expression», ... }
```

The aggregation of events is realized by means of `group by` and `aggregate` clauses that are amended to a conjunctive query. Thereby, the `aggregate` is optional whereas it can only be specified in conjunction with a `group by`. The `group by` specifies how the matching events are separated into groups and the `aggregate` specifies how the values of events contained in one group are aggregated into a single value. To this end, expressions in the aggregate part can contain the additional aggregation functions `min`, `max`, `avg`, `sum`, and `count`.

Similar to negations, grouping is not monotonic and requires further temporal restrictions that are restricting the occurrence of the queried events prior to their grouping to obtain a valid query. To this end, it is sufficient to bound above the end of the reception time of all events whose event identifier or reception time is not specified in the `group by` clause. Note that this criterion covers the most common case, whereas further and more generic criteria, in particular those with respect to user defined times, are discussed in Section 5.2.5.

EXAMPLE Consider the following query in Listing 5.9. When an alarm occurs, the query determines the average temperature `Tavg` within the last minute before the alarm. Note that thereby the end of the reception time of temperature events is bounded above, namely, relative to the end of the reception time of the alarm event.

To this end, alarm events are joined with temperature events that occurred one minute before the alarm. Subsequently, the events are grouped by the alarm event, that is, for each individual alarm event there is one group that contains all temperature events which joined

Listing 5.9: Grouping and Aggregation of Events

```
and{
  event e: alarm{ area{var A} },
  event f: temp{ area{var A}, value{var T} }
} where { f during from-end-backward(e, 1min) }
  group by { event e } aggregate { var Tavg = avg(var T) }
```

with the alarm. And finally, the temperature values within each group are aggregated to a single value, namely, the average temperature.

Without the grouping in the query of Listing 5.9, each pair of alarm and temperature events would result in a match of the composite query. However, because of the grouping over alarm events, the query matches only once for each alarm event independent of the number of matching temperature events.

HAVING CLAUSES FOR GROUPINGS Groupings can be arbitrarily combined with conditions and data definitions as long as the visibility of variables is respected, which is further formalized in Section 5.2.1. In this way, conditions can constrain aggregated values and thus resembles the having clause of the query language for relational databases SQL.

Listing 5.10: Realizing a Having Clause

```
and{
  event e: alarm{ area{var A} },
  event f: temp{ area{var A}, value{var T} }
} where { f during from-end-backward(e, 1min) }
  group by { event e } aggregate { var Tavg = avg(var T) }
  where{ var Tavg > 100 }
```

The grouping in Listing 5.9 is already preceded by a `where` part, which specifies a temporal condition on temperature events. Moreover, the grouping can also be followed by arbitrary many `where` and `let` parts. With the given adaptations, the query from Listing 5.10 matches only alarm events that occur in an area with an average temperature `Tavg` that exceeds 100 degrees.

GROUPING OVER TIMES RATHER THAN EVENTS The group by clause is not restricted to event identifiers but can also contain variables. In addition, recall that for valid groupings it is sufficient to bound above the reception time of all events whose event identifier or reception time is not specified in the group by clause.

These circumstances can be exploited in combination to obtain a grouping over a tumbling window in a rather unique way, which does not rely on a separate language construct for tumbling windows. Consider the query in Listing 5.11. As it is composed only from the

atomic query for `temp` events and the group by clause contains the variable `Min` which depends on the reception time of the queried `temp` event, the grouping is indeed valid.

Listing 5.11: Realizing a Tumbling Window

```
event e: temp{ area{var A}, value{var T} }
  let { var Min = ceil(end(e), 1min) }
  group by { var Min, var A } aggregate { var Tavg = avg(T) }
```

Because of the grouping over the variables `Min` and `A` the average temperature `Tavg` is determined for all temperature events that occurred in the same minute and in the same area. Accordingly, the grouping over `Min` has the same effect as a grouping over a tumbling window with the size of one minute.

APPLYING MULTIPLE GROUPINGS The ability to combine groupings with conditions and data definitions even includes the possibility to specify multiple groupings for one conjunctive query. At least as long as the visibility of variables and the temporal restrictions for groupings are respected. This is a powerful means to concisely formulate sophisticated queries.

Note, however, that the visibility of variables need to be respected. As a consequence, the sets specified after different `group by` parts are commonly in a subset relationship.

Listing 5.12: Query with Multiple Groupings

```
and{
  event e: alarm{},
  event f: temp{ area{var A}, value{var T} }
} where { f during from-end(e, 1min) }
  group by { event e, var A } aggregate { var Tavg = avg(T) }
  where { var Tavg > 100 }
  group by { event e } aggregate { var Acount = count(var A) }
```

When an alarm occurs, the query from Listing 5.12 determines the number of areas `Acount` where the average temperature exceeds the threshold of 50 degrees within the next minute. To this end, the average temperature of each area is determined by means of a grouping and subsequently the number of areas that exceed the threshold of 50 degrees is obtained by means of another grouping.

5.1.7 *Existential Queries*

By design of Dura, events are not consumed or absorbed during the evaluation of Dura queries and thus in a conjunctive query one event can be joined with several other events generating as many answers

as there are matching partners. For some queries, however, it is only important to determine whether there is at least one matching partner for an event whereas the exact number of matching events is irrelevant.

Querying the existence of events is realized in Dura by means of a `group by` without an `aggregate` part. Thereby the grouping is used to accumulate several events instead of generating a match of the conjunctive query for every pair of matching events.

EXAMPLE To roughly estimate the characteristics of a fire it is reasonable to observe how quickly the temperature rises in a certain area after the detection of the fire alarm. To this end, it is sufficient to determine whether the temperature exceeded a certain threshold within a given time.

Listing 5.13: An Existential Query

```
and{
  event e: fire-alarm{ area{var A} }
  event f: temp{ area{var A}, value{var T} }
} where { f during from-end(e, 2min), var T > 400 }
  group by { event e }
```

The query in Listing 5.13 determines whether the temperature exceeded 400 degrees within 2 minutes after a fire alarm. It matches, however, at most once for every fire alarm, as the query is grouped by the fire alarm. In this way, it only matters whether there are any temperature events but it is irrelevant how many there actually are. Thus, by means of the grouping the query for temperature events can be considered to be existentially quantified.

5.2 DEDUCTIVE AND REACTIVE RULES

Deductive and reactive rules form the foundation for the specification of complex events and for the initiation of reactions based on the occurrence of events.

Deductive rules specify a complex event in their head that is derived whenever the event query in their body matches the stream of events. They hence correspond to views known from database systems and are free of side effects. Deductive rules provide a convenient and natural abstraction mechanism for the definition of higher-level complex events. By contrast, reactive rules specify a (complex) action in their head that is executed whenever the event query in their body matches. Accordingly, reactive rules make the transition from purely declarative deductive rules and event queries to imperative actions that intentionally entail side effects that can in turn be observed by means of event queries.

However, similar to rules applied in logic programming, some restrictions apply to the formalization of rules. First of all, rules (and interestingly even queries) need to be range restricted to guarantee a sound evaluation. Second of all, for the incremental evaluation of deductive rules over a unbounded stream of events further temporal conditions on queries containing non-monotonic operators, such as, negation and aggregation, need to be satisfied. Finally, there apply restrictions on user-defined reception times of derived events, in particular on events that are derived by recursive rules.

5.2.1 *Range Restriction of Queries and Rules*

To obtain sound queries and rules, variables used in the supplement of queries and in rule heads need to be properly defined so as to provide concrete values during the evaluation of queries at runtime. The intuition behind "properly defined" variables is formalized in the following by means of the notions polarity and range restriction.

Obviously, both notions have been inspired by the respective notions used in mathematical logic [Bry+07]. However, be aware that they substantially differ in some aspects due to the additional constructs that are available in Dura.

POLARITY OF REFERENCES The polarity of references in Dura queries is inductively defined as follows. A variable `V`, identifier `i`, or path `p` has *positive polarity* in the query

event `e: t`	if `i` coincides with `e`, the prefix of `p` coincides with `e`, or `V` occurs in the query term `t`
`q` **`where`** `c`	if it has positive polarity in the sub-query `q`
`q` **`let`** `l`	if it has positive polarity in the sub-query `q` or occurs on the left hand side of an assignment in `l`
`q` **`group by`** `g` **`aggregate`** `a`	if it occurs in `g` or on the left hand side of an assignment in `a`
`and`$\{q_1,\ldots,q_k\}$	if it has positive polarity in any sub-query q_i
`or`$\{q_1,\ldots,q_k\}$	if it has positive polarity in all sub-queries $q_1,\ldots,q_k$

Note that the operator `not` does intentionally not occur in the inductive definition of polarity. As a consequence, in contrast to the polarity as it is defined for variables in mathematical logic, variables that are nested in multiple negations cannot have positive polarity. This particular deviation between the two notions is because Dura applies negation as failure whereas mathematical logic applies classical negation.

RANGE RESTRICTION OF QUERIES The *range restriction* of queries is defined inductively as follows. The query

`event e: t`	is range restricted (RR)
`q` **`where`** `c`	is RR if `q` is RR and all references of `c` have positive polarity in `q`
`q` **`let`** `l`	is RR if `q` is RR and all references on the right hand side of assignments in `l` have positive polarity in `q`
`q` **`group by`** `g` **`aggregate`** `a`	is RR if `q` is RR and references on the right hand side of assignments in `a` and in `g` have positive polarity in `q`
`not``{ q }` **`where`** `c`	is RR if `q` is RR and each reference of `c` has positive polarity in its comprising query or in `q`
`and`$\{q_1,\ldots,q_k\}$	is RR if all sub-queries $q_1, \ldots, q_k$ are RR
`or`$\{q_1,\ldots,q_k\}$	is RR if all sub-queries $q_1, \ldots, q_k$ are RR

RANGE RESTRICTION OF RULES In anticipation of the notions, deductive and reactive rules

`DETECT` `c` **`ON`** `q` **`END`** and **`ON`** `q` **`DO`** `a` **`END`**

are *range restricted*, if all references that occur in the construct term `c` or the action `a`, respectively, have positive polarity in `q` and if furthermore the query `q` is range restricted.

Interestingly, the range restriction of rules depends, in addition to the polarity of variables in the rule head, on the range restriction of the query in the body. This discrepancy to range restriction as it is usually defined for rules considered in mathematical logic and logic programming is required to cover the constructs that are only available in Dura.

EXAMPLE In Listing 5.14, the variable `A` and the identifier `f` have positive polarity in the atomic sub-query of the negation whereas thereof only the variable `A` has positive polarity in the conjunctive query. In addition, the identifier `e` has positive polarity in the disjunctive and conjunctive query, respectively.

The rule is, however, not range restricted, because the condition `{e,f}` *`within`* `30` contains the identifier `f` which does not have positive polarity in the conjunctive query.

WEAK RANGE RESTRICTION For convenience, it seems desirable to apply a slightly more liberal variation of range restriction, which facilitates a more unrestricted replacement of conditions referring to negated queries. The following definition of weak range restriction generalizes the one from above and enables, eg, to specify conditions referring to negated queries not only in the supplement of the nega-

Listing 5.14: Polarity and Range Restiction

```
DETECT
  pre-alarm{ area{var A} }
ON
  and{
    or{
      event e: temp{ area{var A}, value{var T} }
      event e: smoke{ area{var A}, conc{var C} }
    }
    not event f: alarm{ id{var Id}, area{var A} }
  } where { var T > 100, var C > 0.1, {e,f} within 1min }
END
```

tion but also in the supplement of queries containing the negation. It does not increase the expressiveness of Dura, though. In fact, weakly range restricted rules can be converted to semantically equivalent range restricted rules. For details refer to Section 10.3.6.

A reference has *weak positive polarity* in the query

not{ q }	if it has positive polarity in q
and{$q_1,\ldots,q_k$}	if it has (weak) positive polarity in any sub-query q_i
or{$q_1,\ldots,q_k$}	if it has positive polarity in one sub-query q_i or weak positive polarity in all sub-queries $q_1,\ldots,q_k$

Except for the different term, the definition of range restriction and *weak range restriction* for queries and rules only differs in one case that applies for the range restriction for queries. The query

q **where** c	is weakly RR if q is weakly RR and all references of a conjunct of c with weak positive polarity in q refer to the same atomic query

EXAMPLE In Listing 5.14, the variables T, C, and Id as well as the identifier f have weak positive polarity but no positive polarity in the conjunctive query. Moreover, the rule and accordingly all of its sub-queries are weakly range restricted.

Note that in the following it is sufficient if rules and queries are weakly range restricted. Obtaining rules that are range restricted in the strict sense is only relevant for the normalization of rules described in Section 10.3.

5.2.2 *Deductive Rules*

Deductive rules derive higher level events based on the occurrence of events in the stream. They consist of two distinct parts, the rule head and the rule body, or consequent and antecedent as they are

also called [Bry+07], that share commonly named variables and are denoted

```
DETECT «event» ON «query» END
```

whereby the body contains an event query and the head specifies a complex event by means of a construct term. Note that the strict separation between body and head contributes to the separation of concerns as it separates the querying of events from the construction of new events, two aspects that are indeed clearly orthogonal.

Deductive rules are incrementally evaluated to fit the volatile nature of event streams, that is, new events are gradually derived according to the specification in the rule head and added to the event stream while events matching the query in the body of the rule occur. Hence, deductive rules correspond to materialized views from database systems. As the name suggests, those kind of rules are declarative and thus queried events are not subject to consumption or absorbance. Moreover, rules are evaluated in a monotonic fashion and thus events that have been derived cannot be updated and there is no notion of premature events that corresponds to preliminary results and may need to be reverted if further events arrive in the system.

RECEPTION TIME OF DERIVED EVENTS The reception time of derived events is implicitly determined by the EPS unless the reception time is explicitly adapted by the programmer as described below.

By default, the reception time of derived events is determined be the smallest interval that comprises the reception time of all positively queried atomic events, that is, all atomic events whose event identifier have positive polarity in the query in the body of the rule. More formally, the begin and end of the reception time is determined by `least(begin(`e_1`,...,`e_k`))` and `greatest(begin(`e_1`,...,`e_k`))`, respectively, whereby $e_1, \ldots, e_k$ are the identifiers with positive polarity in the query of the rule body.

EXAMPLE Among other things, deductive rules are convenient for schema mediation and the enrichment of basic events. In Listing 5.15 the first rule derives `temp` events from generic `sensor-msg` events which describe their content by means of a rather unintuitive number in their payload instead of a meaningful event type and provide the temperature in degrees Kelvin. The second rule transforms temperature readings from `temp-value` events carrying the temperature in degrees Fahrenheit into `temp` events that specify the temperature in degrees Celsius.

Naturally, in large infrastructures, there are several different kinds of temperature sensors with different message formats and different properties they provide. Therefore, a homogeneous representation is highly desirable as rules can rely on one single event type to obtain

Listing 5.15: Schema Mediation

```
CONST TEMP_SENSOR_ID = 29
CONST SMOKE_SENSOR_ID = 59

DETECT
  temp{ area{var A}, value{Tkel-273.15}, sensor-id{var S} }
ON
  event e: sensor-msg{
    type{const TEMP_SENSOR_ID},
    area{var A}, value{var Tkel}, sensor{var S}
  }
END

DETECT
  temp{ area{var A}, value{var Tcel}, sensor-id{var S} }
ON
  event e: temp-value{ aid{var A}, val{var Tfah}, sid{var S} }
    let { var Tcel = (var Tfah - 32)/1.8 }
END
```

the temperature and do not need to be aware of all different event types and formats that are available. Moreover, using a homogeneous representation improves the maintainability of rules, as new sensors with different formats can be easily integrated by simply adding a rule that transforms events of the new format into `temp` events.

EXAMPLE Deductive rules can also be used to define a hierarchy of events to obtain various representations of the same event with different levels of abstraction. The rules in Listing 5.16, for instance, specify that uncertain fire alarms, potential overcrowding, and calls made by emergency telephones are all a kind of uncertain alarm whereas uncertain alarms and certain alarms are alarms.

Note that the event identifier of all atomic queries intentionally coincides with `e`. Therefore, `e` is positive with in the disjunctive query and the reception time can be determined as described above.

Depending on the requirements, programmers can thus write very generic rules that are just querying alarms, or, if desired, query more specific events to formulate more rules that apply only in the specific situation.

ADAPTING THE RECEPTION TIME OF EVENTS By default, derived events are associated with a reception time that depends on the times of the events it has been derived from. More precisely, the default reception time of derived events corresponds to the smallest time interval comprising all time intervals of positively queried events that contributed to its derivation, that is, the begin and end of derived events corresponds to the least beginning and largest ending of all positively queried events, respectively.

Listing 5.16: A Hierarchy of Events

```
DETECT
  uncertain-alarm{ area{var A} }
ON
  or{
    event e: uncertain-fire-alarm{ area{var A} },
    event e: potential-overcrowding{ area{var A} },
    event e: sos-telephone-call{ area{var A} }
  }
END

DETECT
  alarm{ area{var A} }
ON
  or{
    event e: uncertain-alarm{ area{var A} },
    event e: certain-alarm{ area{var A} }
  }
END
```

Although the reception time of events is implicitly determined by the system, programmers can specify an alternative definition as long as the preconditions discussed in Section 5.2.5 are satisfied. In very simplified terms, it is sufficient if the adapted time is specified relative to the end of all reception times of events positively queried in the body of the rule.

EXAMPLE One situation where adapting the time of derived events seems desirable is the following. Temperature sensors usually emit temperature values in regular intervals, eg, every 15 seconds. Thus, if more than one minute has passed after the last `temp` event has been obtained by the system one can conclude that the corresponding sensor is broken. However, although the default time determined by the EPS is usually sensible for derived events, in this particular case it seems desirable to adapt the time for `broken-temp-sensor` events to corresponds to the first time a `temp` event remains absent. This is realized in Listing 5.17 by explicitly overwriting the end of the reception time with `end(e) + 15sec`.

STRUCTURE OF RULES Recall that CEP is indeed about the detection of higher level events in a continuous and timely fashion and *not* about expressing arbitrary computations. Accordingly, EQLs are commonly restriction to hierarchical programs [Cla78; She85] to obtain an efficient incremental evaluation of queries that suits the characteristics of unbounded streams.

In addition to commonly applied hierarchical rules, Dura even supports some limited form of recursive rules. Recursive rules are valid if they ensure that recursively derived events make some kind of tem-

Listing 5.17: User Defined Reception Time

```
DETECT
  broken-temp-sensor{
    sensor-id{var S},
    reception-time{ end{ end(e)+15sec } }
  }
ON
  and{
    event e: temp{ sensor-id{var S} }
    not event f: temp{ sensor-id{var S} }
  } where { f during from-end(e, 1min) }
END
```

poral progress, that is, if the end of the reception time of derived events exceeds the end of the reception time of the recursively queried events by a constant duration. However, be aware that recursive rules are intended for very specific purposes, eg, to internally realize updates of stateful data which leads to cyclic dependencies between rules, and it is usually sufficient for programmers to rely on hierarchical rules in Dura. Nevertheless, further details on recursive rules are discussed in Section 5.2.4.

5.2.3 *Reactive Rules*

The execution of reactions to detected situations is realized in Dura by means of reactive rules. Reactive rules make the transition from the purely declarative world of events and stateful objects the imperative world of actions.

Similar to deductive rules, reactive rules consist of a distinct head and body part. Their body contains an event query specifying when the rule triggers whereas their head contains a potentially composite action specifying the corresponding reaction to the queried events. In Dura reactive rules are denoted

```
ON «event query» DO «action» END
```

Whenever the event query in the body matches, the action in the head is initiated. Thereby, the execution of so-called internal and external actions is discriminated which is further described in Section 5.4.

Be aware that, although reactive rules of Dura resemble productions used in production and expert systems [BFC86; Wat85; DK75], in particular if they alter stateful objects which is further addressed in Section 5.3, there is no automatic conflict resolution between rules. Accordingly, the action in the rule head is executed as often as the event query in the body matches the stream of events.

EXAMPLE When an alarm occurs, eg, due to the outbreak of a fire, there are various actions that need to be taken. Some of them are spe-

cific for certain situations and need approval of the human operator whereas others are very generic, eg, the shutdown of the heat and ventilation system, and can thus be triggered by the system without further approval. This behavior is implemented by the reactive rule in Listing 5.18. Whenever an alarm occurs, the immediate actions which can be executed without further approval, are initiated by the system.

Listing 5.18: A Reactive Rule

```
ON
  event e: alarm{ area{var A} }
DO
  action a: immediate-actions{ area{var A} }
END
```

Note that if there are several alarms for the same area, eg, if there are multiple justifications that lead to the derivation of several `alarm` events within a short amount of time, the immediate actions are executed multiple times. It is thus desirable to adapt the rule, eg, by means of negated or existentially quantified queries, to prevent the repeated execution of immediate actions.

REQUEST TIME OF TRIGGERED ACTIONS Naturally, the time an action is requested for execution by the EPS depends on the time the event query in the body of the reactive rule matches. More precisely, it corresponds to the end of the reception time that would be determined in case of a declarative rule.

Accordingly, in Listing 5.18 the time the `immediate-actions` action is requested by the system simply corresponds to the end of the reception time of the `alarm` event. Note, however, that in case of physical actions which are executed by external actuators the actual begin of the action may be deferred because of system inherent latency effects. This aspect of physical actions is elaborated in detail in Section 5.4 which comprehensively covers various aspects of complex actions in Dura.

5.2.4 *Recursive Rules*

Dura supports a limited form of recursive rules. However, be aware that Dura is an event *query* language intended for complex event processing. Therefore, recursive rules are, by design, not intended to provide means for carrying out arbitrary computations as it is desirable for general purpose languages like, for instance, Java and C#. In fact, the EM use cases described in [SB10b; SBR11a] can be sufficiently modeled by means of the recursion capabilities that are provided by Dura.

For rules that derive an event that is (directly or indirectly) queried in their body, it must be satisfied that the end of the reception time of

the derived event exceeds the end of the reception time of the recursively queried event by an arbitrary small but predetermined constant duration. In other words, the reception time of events derived by recursive rules needs to make temporal progress with respect to the queried event whereby the amount of the temporal progress needs to be deducible from the temporal conditions of the query. Note that this condition relates to the stratification of rules [ABW88] and can be seen as temporal stratification.

EXAMPLE The capabilities of recursive rules are exploited in Listing 5.19 to count the number of persons within a room based on `enter` and `exit` events that report in regular intervals how many persons have entered and exited an area.

Listing 5.19: A Recursive Rule

```
DETECT
  person-count{
    area{var A}, count{var Cnew},
    reception-time{ end{ end(e)+10min } }
  }
ON
  and{
    event e: person-count{ area{var A}, count{var C} },
    event f: enter{ area{var A}, amount{var In} },
    event g: exit{ area{var A}, amount{var Out} }
  } where { f during from-end(e, 10min),
            g during from-end(e, 10min) }
    group by { event e, var C, var A }
    aggregate { var Cin = sum(In), var Cout = sum(Out) }
    let { var Cnew = var C + var Cin - var Cout }
END
```

The basic idea is the following, a new `person-count` event is derived every 10 minutes by collecting `enter` and `exit` events within this timespan and by adding their cumulated values to the previous known amount of persons in the area obtained by the (recursive) query to `person-count` events. Note that the end of the reception time of the derived `person-count` event is determined by `end(e)+10min`, thus the time difference between two successive `person-count` events that have been derived by this rule amounts to 10 minutes and therefore the conditions for recursive rules are satisfied.

Note that for the sake of simplicity, we made the implicit assumption that `enter` and `exit` events are issued at least once every 10 minutes. If there is not at least one enter and one exit event within every 10 minutes, the query in the body of the rule of Listing 5.19 does not match and thus the rule does not detect any further events. For a more robust implementation, which does not suffer from this issue, three further rules need to be added to the program which cover the cases when either an enter or an exit event occurs or none of them oc-

curs. An extension that resolves this issue more elegantly is suggested in Chapter 13, though.

EXAMPLE Recursive rules are not restricted to positive event queries but also allow for "recursion through negation" as long as the temporal requirements on derived events are satisfied. This ability is used in Listing 5.20 to imply smoke from the presence of high temperature readings. However, the given examples is elaborated for didactic reasons and has arguable relevance for the considered EM use cases.

Listing 5.20: A Recursive Rule with Negation

```
DETECT
  smoke{ ... }
ON
  and{
    event e: temp{ area{var A}, value{var T} }
    not event f: smoke{ area{var A} }
  } where {
    var T > 400,
    f during shorten(from-end-backward(e, 2min), 5sec)
  }
END
```

Note how the interval function `shorten` causes the required temporal distance between the reception times of queried and derived `smoke` events: Unless the reception time of derived `smoke` events is manually adapted it coincides with the reception time of the queried `temp` event. Moreover, the temporal condition entails `end(f) < end(e) - 5sec`. It is hence satisfied that the end of the reception time of derived `smoke` events exceeds the end of the reception time of queried `smoke` events by at least five seconds. Accordingly, the temporal requirements on recursively derived events are satisfied.

If, however, the temporal condition of the rule is instead adapted to `f during from-end-backward(e, 2min)` it merely entails `end(f) < end(e)` and thus the end of the reception time of derived and queried `temp` events can become arbitrary close. Therefore, there is no predetermined temporal distance and thus the adapted conditions do not satisfy the temporal requirements on recursively derived events.

5.2.5 *Progressing Attributes: A Generalization of Time*

The incremental evaluation of Dura queries is based on the Temporal Stream Algebra (TSA) which has been proposed by Brodt and Bry [BB12a]. In the following, we will informally summarize aspects of TSA that are relevant for the specification of correct Dura rules by means of informal examples. For the formal details refer to [BB12a].

PROGRESSING ATTRIBUTES Brodt and Bry [BB12a] propose a generalization of time called progressing attributes. Progressing attributes are specific attributes of event streams. In simplified terms, for a progressing attribute a holds that for any upper bound b on the values of the attribute a there is a time p so that the values of *all* future events exceed the bound in this attribute.

By interpreting the values of a and the bound b as functions $a(t)$ and $b(t)$, respectively, the notion of progressing attributes can be equivalently formulated by means of the Landau notation [Cor+01]: a is a progressing attribute if for all constant functions $b(t)$ holds $a(t) \in \Omega(b(t))$.

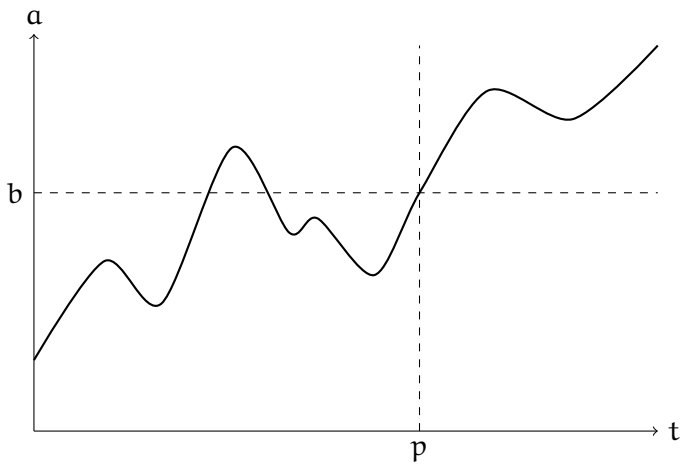


Figure 5.1: Illustration of a Progressing Attribute

The reception time of basic events is an example for a progressing attribute: As time advances the reception time of newly obtained basic events grows and eventually all future events will be associated with a reception time that exceeds any predetermined time bound b. Be aware that the notion of progressing attributes does not require that the values are monotonically increasing, as Figure 5.1 illustrates.[1]

PROGRESSING ATTRIBUTES IN DURA Each progressing attribute of an event can be interpreted as an independent time line or time model. Accordingly events can be associated with an arbitrary number of independent time lines. Note, however, that it is statically verified whether attributes distinguished as progressing attributes are determined by suited values as it is described in the following.

Programmers can distinguish attributes as progressing attributes in the schema of events by means of the predicate `progressing` specified in a `where` part amended to the schema as it is illustrated in Listing 5.21. In addition to the designation of progressing attributes,

1 Note that the representation of the graph is rather simplified as it would be discrete with realistic data.

the `where` part can also contain constraints that restrict the values of attributes.

Listing 5.21: A Schema of Events with Progressing Attributes

```
temp{
  id{identifier},
  reception-time{ begin{timestamp}, end{timestamp} },
  area{long}, value{double}, sensor-id{long}
} where {
  progressing(id), progressing(reception-time.end),
  reception-time.begin <= reception-time.end
}
```

By default, the attributes `id` and `reception-time.end`, which are implicit associated with every event, are distinguished as progressing attributes. Moreover, the begin of the reception time is constrained to be lower or equal to its end by means of the temporal formula `reception-time.begin <= reception-time.end`. In the schema of temperature events in Listing 5.21 these implicit properties and conditions on attributes are explicitly specified.

PROGRESS INFORMATION For the evaluation of rules by means of TSA, it is not sufficient to know that there exists a point in time `p` from which on all subsequent events exceed a given bound `b` in a certain attribute. In addition, suited progress information is required that facilitates to determine whether the time `p` has actually passed. Accordingly, expressions determining values of progressing attributes need to provide progress information.

With respect to the rule

```
DETECT e ON q END
```

an expression provides progress information for a progressing attribute of `e`, if it maintains progress information from `sequence.next()` or `system-time.now()` or if it maintains progress information from all events queried in `q`. Thereby, maintaining progress information is defined inductively as follows. Let `t`, $t_1, \ldots, t_n$ determine expressions and `d` a duration, then

1. `t` coinciding with a reference maintains progress information if it refers to a progressing attributes of an event queried by `q`,
2. `t + d` maintains progress information of `t`,
3. `greatest(`$t_1, \ldots, t_n$`)` maintains progress information of *all* terms $t_1, \ldots, t_n$, and
4. `floor(t, d)` and `ceil(t, d)` maintain progress information of `t`.

In addition, if the condition $t_1\ \theta\ t_2$ with $\theta \in \{<, <=, =\}$ is satisfied for the expressions t_1 and t_2, eg, if it is explicitly specified in the `where` part of `q` or in the schema of an event queried by `q`, then t_2 maintains progress information of t_1 and of t_2.

INCREMENTAL EVALUATION AND GARBAGE COLLECTION Incremental evaluation and garbage collection are two crucial properties of CEP. Event streams are conceptually unbounded and therefore queries can only be evaluated in an incremental fashion, as the stream is never completely obtained. Moreover, the unbounded stream cannot be completely persisted and thus any machine will run short of storage without appropriate garbage collection of obsolete events. Naturally, TSA supports both, at least if certain conditions on progressing attributes are met.

To facilitate the incremental evaluation of a rule

```
DETECT e ON q END
```

an expression determining a progressing attribute of `e` needs to establish an upper bound for a progressing attribute of each event queried in `q`. Similarly, to facilitate garbage collection of an event queried in `q` an expression determining a progressing attribute of `e` needs to establish a lower bound for a progressing attribute of the queried event.

Note that conditions that enable the incremental evaluation of rules are mandatory. By contrast, garbage collection of queried events is optional and thus conditions enabling garbage collection may be omitted.[2] Rules not facilitating garbage collection on an event should be used as rarely as possible and with extreme caution, though.

EXAMPLE Consider the rule deriving `ce` events in Listing 5.22. By default, the `id` attribute of derived `ce` events is implicitly determined by `sequence.next()` and the end of the reception time of derived `ce` events is implicitly determined by `greatest(end(e), end(f))`.

Listing 5.22: Simplified Rule with Negation

```
DETECT
  ce{ ... }
ON
  and{
    event e: ...,
    event f: ...,
    not event g: ...
  }
END
```

2 Note that in some cases such kind of conditions simply do not exists, eg, for negations over finite but arbitrarily large time windows.

Note that hence, the rule cannot be incrementally evaluated, as none of the progressing attributes of `ce` is determined by an expression that maintains progress information from all queried events, that is, maintains progress information from `e`, `f`, and `g`. This issue is commonly resolved for rules with negation by adding a temporal constraint to the `where` part that specifies an upper bound on the end of the negatively queried event which is relative to a positively queried event, eg, `g during from-end(e, 20sec)`.

And indeed, this condition is equivalent to

```
and{ end(e) < begin(g), end(g) < end(e) + 20sec }
```

which furthermore entails

```
end(g) - 20sec < greatest(end(e), end(f))
```

Therefore, the expression `greatest(end(e), end(f))` now also maintains progress information of `g` and furthermore specifies an upper bound on a progressing attribute of every queried event so that the query can be incrementally evaluated.

If furthermore the temporal condition `{e,f} within 1min` is added to the `where` part of the query, then garbage collection of all three events becomes feasible. The temporal condition is equivalent to

```
greatest(end(e), end(f)) - least(begin(e),begin(f)) <= 1min
```

as the condition `reception-time.begin <= reception-time.end` implicit added to the schema of every event, it furthermore entails

```
and{ greatest(end(e), end(f)) <= end(e) + 1min,
     greatest(end(e), end(f)) <= end(f) + 1min }
```

In combination with the condition `g during from-end(e, 20sec)` from above it even entails

```
greatest(end(e), end(f)) < end(g) + 1min
```

and thus the expression `greatest(end(e), end(f))` maintains progress information of all queried events and entails a lower bound on progressing attributes of all queried events, thus garbage collection for `e`, `f`, and `g` events is feasible.

5.3 STATEFUL OBJECTS

Stateful objects are data terms that can be altered over time. They resemble facts in a knowledge base or tuples in a database with the difference that updates of their values are carried out in a non-destructive and declarative way. To this end, updates merely mark the redundant values of stateful objects as obsolete so that they remain in the system and can be obtained by appropriate queries. In

this way, queries for stateful objects are not limited to obtain the most recent value, but can also refer to former values of stateful objects.

Stateful objects are particularly useful to model the varying properties of (physical) objects in a non-volatile manner. This includes, eg, the global operation mode of the infrastructure as well as the current status of external devices. Moreover, static information, such as, topology information, can be represented by means of static stateful objects, that is, stateful objects that cannot be modified at runtime.

Stateful objects are carefully designed to complement the volatile nature of events and at the same time suit the temporal aspects of event processing. Other EQLs often lack a notion that is capable of modeling stateful information or merely provide an integration with common databases or knowledge bases, which lack a notion of time.

5.3.1 *Representation of Stateful Objects*

The representation of stateful objects closely resembles the representation of events, at least at the language level that is visible to programmers. Each stateful object has a type and a user defined payload that is specified by means of Xcerpt terms conforming to the Dura specific restrictions.

EXAMPLE A particularly relevant stateful object represents the operation mode that specifies the conditions of each area, ie, whether it is in *normal*, *exceptional*, or *emergency* condition. According to the current operation mode, the reactions to certain events may substantially differ as, eg, the emerging overcrowding of a metro station requires immediate actions in case of emergencies whereas no or at least other actions are taken when the station is in normal operation mode during rush hour.

Listing 5.23: Schema of a Stateful Object

```
operation-mode{ area{long}, mode{int} }
```

Throughout the following examples we presume that the constants `OPM_NORMAL = 0`, `OPM_EXCEPTIONAL = 1`, and `OPM_EMERGENCY = 2` are available. In this way, the `mode` is efficiently represented as an attribute of type int whereas the name of the constants are convenient for programmers. Moreover, operation modes obtain a natural order and can thus be conveniently compared by means of inequations.

INTERNAL REPRESENTATION OF STATEFUL OBJECTS Whenever a stateful object is modified, its current value is not overwritten but instead merely marked as obsolete and remains, at least conceptually, in the system. To this end, a stateful object is actually determined by

a series of successive values whereby each value is associated with a unique key and only valid within a certain time interval. However, be aware that the upper bound of the time interval remains undetermined until the moment the corresponding value becomes obsolete.[3]

Accordingly, the representation of values, which in combination determine a stateful object, resembles the representation of fluents from the event calculus [MS02; Sha99]. Fluents are properties in a logic based formalism that are true if they have been initiated by an action and remain true until they have been terminated by another action. In fact, the same idea is used to internally represent stateful objects. But despite the close resemblance of fluents and stateful objects, the focus of the event calculus and Dura substantially differs. The event calculus aims at reasoning about temporal properties of (temporal) systems whereas Dura is an event query language which aims at detecting and reacting to high level situations that emerge at runtime.

STATIC STATEFUL OBJECTS Stateful objects are particularly useful to model varying properties that describe the state of the infrastructure. However, some information, such as runtime independent domain knowledge is only rarely updated or does not need to be updated at runtime at all. This kind of information is represented in Dura by means of static or constant stateful objects, that is, stateful objects whose values are initiated once and cannot be updated at runtime. Accordingly, the initial values of a stateful object are always valid and thus queries can be formulated more concisely as temporal restrictions are not required.

Listing 5.24: Schema of a Static Stateful Object

```
static
sensor{ sensor-id{long}, area{long}, type{long} }
```

Static stateful objects are distinguished in their schema by means of the keyword `static`. Listing 5.24 describes, for instance, the schema of the static stateful object `sensor` which represents a mapping between the unique identifiers specified in the payload of raw sensor events and the areas the sensors are actually located in.

INITIAL VALUES OF STATEFUL OBJECTS Initial values of stateful objects can be provided during the initialization phase of the EPS. This is in particular necessary for, but not limited to, the initialization of static stateful objects that cannot be modified at runtime.

The initialization of stateful objects is further illustrated in Section 12.3.

3 As a consequence of this representation and in contrast to events, the valid time is not explicitly represented in the schema of stateful objects.

5.3.2 *Atomic Stateful Object Queries*

Stateful objects are queried in a way that resembles atomic event queries, that is, by means of query patterns with variables in place of the pattern where relevant data should be extracted.

Each pattern is preceded by the keyword `state` and an alphanumeric identifier that is subsequently used to constrain the time of queried stateful objects, more precisely, to constrain the time of matching values determining a stateful object.

As the payload of stateful objects can be modified, it is indeed important to restrict the time of the query, as it otherwise matches all values of the stateful object regardless of when they have been valid. Note that this also includes future values of the stateful object, but similar to temporally unrestricted joins of events, the query naturally does not produce answers until the data of the stateful object actually becomes available.

EXAMPLE The atomic query in Listing 5.25 matches the values of the stateful object `operation-mode` and binds the variables `A` and `M` to the area and its corresponding operation mode. By design, this query matches the values of the stateful object regardless of when they are valid, unless the time of the query is restricted appropriately. As a consequence, the query may indeed match repeatedly and provide multiple bindings for the operation mode which are valid at different times.

Listing 5.25: An Atomic State Query

```
state s: operation-mode{ area{var A}, mode{var M} }
```

Be aware that a query of a stateful object can result in multiple variable bindings, even if the query time is restricted to a single instant. For instance, in Listing 5.25 the stateful object `operation-mode` is queried and as there are usually several areas in a metro station and each area has its own operation mode, the query of the stateful object results in multiple bindings, one for each area.

5.3.3 *Integration with Event Queries*

A major advantage of Dura over many other EQLs is its deep integration of queries for events and stateful objects. Other EQLs often provide either a notion of events or a notion similar to stateful objects but do not integrate the two of them into a homogeneous language. EQLs based on the event calculus [ASP12; Paso5], for instance use fluents to emulate events whereas EQLs based on production systems [Drob; TIB] use updates to the knowledge base to emulate events.

However, we argue that the two notions should be kept separate as they cover different aspects of CEP. Accordingly, the notions of events and stateful objects are clearly separated in Dura. As a consequence, queries for stateful objects, which merely obtain the value of a stateful object at a certain time, need to be specified in conjunction with event queries in the body of rules because deductive and reactive rules are intended to detect or react to observable changes which are represented in Dura solely by means of events.

This contrasts with production systems where queries for facts are specified in the body of rules and rules are triggered when the query matches the facts that are currently available in the knowledge base. However, production systems do not have a notion of events and thus do not need to distinguish events from facts.

EXAMPLE Listing 5.26 contains a composite query integrating queries for events and stateful objects. As with pure composite event queries, shared variables are implicitly joined and variables with positive polarity can be specified in the supplement of a query or in the head of a rule. Accordingly, the query matches when an alarm occurs and whenever the corresponding area is in emergency operation mode.

Listing 5.26: A Composite Query with Stateful Objects

```
and{
  event e: alarm{ area{var A} },
  state s: operation-mode{ area{var A}, mode{OPM_EMERGENCY} }
}
```

Note that the join between the event and the stateful object is temporally unbounded and thus the query for the operation mode matches not only areas that are currently, that is, when the alarm event occurs, in emergency mode, but matches whenever the operation mode of an area is *emergency*. Consequently, the composite query matches whenever the operation mode has been emergency in the past and it matches again when the operation mode is updated to emergency in the future.

CONSTRAINING THE QUERY TIME To obtain the values of a stateful object at a certain time, the query time of the stateful object needs to be restricted appropriately. To this end, the identifier introduced in a query of stateful objects is used in the `where` part of composite queries to constrain the time when the values of stateful objects are actually queried.

There are two predicates that are intended to be used with stateful objects, namely, *valid-at* and *valid-during*. The predicate *valid-at* constrains the query time of a stateful object to a certain time point whereas the predicate *valid-during* is used to obtain values that have

been valid for some time during a given time interval. Both predicates are actually syntactic sugar for the formulas from Table 5.3 whereby

`s` *`valid-at`* `tp`	≡	`and{cre(s) < tp, tp <= term(s)}`
`s` *`valid-during`* `ti`	≡	`and{cre(s) < end(ti), begin(ti) <= term(s)}`

Table 5.3: Translation of Relations for Stateful Objects to Formulas

`cre` refers to the time a matching value of a stateful object became valid and `term` refers to the time the value has been marked as invalid again. Note that, `cre` and `term` are short for created and terminated which is related to the objects that are used to internally represent values of stateful objects.

EXAMPLE Usually the time constraining the values of a stateful objects refers to the past. However, it is also sound to refer to times in the future. In that case the evaluation of the composite query is, similar to the evaluation of negation and grouping, deferred until the given time has passed and the requested data is actually available and the query can be evaluated correctly.

Listing 5.27: Constrained Stateful Object Query Time

```
and{
  event e: alarm{ area{var A} },
  state s: operation-mode{ area{var A}, mode{OPM_EMERGENCY} }
} where { s valid-during from-end-backward(e, 2min) }
```

The query in Listing 5.27 matches whenever an alert is detected and the operation mode of the corresponding area has been *emergency* during the last two minutes before the end of the alarm. Note that whenever the alarm occurs, the composite query is instantaneously evaluated, as the query for the stateful object refers to values that have been valid two minutes before the occurrence of the event, that is, to values that are already known to the EPS.

However, the given query can indeed be modified so that the stateful object is queried after the detection of the event, eg, by adapting its `where` part to

```
where { s valid-during from-end(e, 2min) }
```

In that case, the query cannot be answered immediately after the detection of the alarm event anymore, as it depends on the values of the stateful objects within two minutes after the detection of the event. Therefore, the evaluation of the query of the stateful object, and hence of the complete composite query, is deferred until the values of the stateful object are determined and the query of the stateful object can be correctly answered.

NEGATION OF QUERIES Queries of stateful objects can be negated, though similar restrictions as for negated event queries apply. Whenever the query of a stateful object is negated, the query time of the stateful object needs to be bounded above with respect to a progressing attribute of a query that has positive polarity in the common comprising query, eg, with respect to the reception time of a positively queried event. Recall that this restriction is mandatory as temporally unrestricted negated queries over streams conceptually cannot be evaluated in an incremental fashion. It is not a restriction that is specific for Dura.

Listing 5.28: A Negated Stateful Object Query

```
and{
  event e: alarm{ area{var A} },
  not state s: operation-mode{ area{var A}, mode{OPM_EMERGENCY} }
} where { s valid-during from-end-backward(e, 2min) }
```

The query in Listing 5.28 matches whenever an alarm occurs and within two minutes before the end of the alarm the operation mode of the corresponding area has never been set to *emergency*. Note that the validity of matching stateful objects is temporally bound above by the end of the reception time associated with the `alarm` event. Hence, the negation of the stateful object is sound and the query can be evaluated right after the detection of the alarm.

IMPACT ON THE RECEPTION TIME OF EVENTS By default, the reception time of events derived by declarative rules depends on the times of event that have positive polarity in the rule body. In presence of queries for stateful objects, the reception time of derived events additionally incorporates the query time of stateful objects that have positive polarity in the rule body. More precisely, the end of the reception time is at least as large as the point in time when a query for a stateful object matches. For instance, the end of the reception time of events derived by means of the query from Listing 5.27 is at least as large as `greatest(cre(s), end(e)-2min)`.

Note however, that the valid time of queried stateful objects is not incorporated into the reception time of derived events. The end of the valid time may be undetermined when a query for a stateful object matches or is arbitrary large in case of static stateful objects, which cannot be terminated or updated. Accordingly, incorporating the valid time of stateful objects into the reception time of derived events would cause rather undesirable effects that are not compatible with the volatile nature of events.

5.3.4 *Modifying Stateful Objects*

Stateful objects are modified by means of internal actions that are triggered by reactive rules. For each stateful object there are internal update actions available that are implicitly provided by the definition of the stateful object. However, as Dura does not support the retraction of already derived events, only currently valid values of stateful objects can be modified. More precisely, values of a stateful objects can be updated only if they are valid when the corresponding action is requested by means of a reactive rule.

Naturally, unless a reactive rule triggers the modification of a stateful object its values remain unchanged, although this circumstance is not explicitly indicated by means of rules. This aspect relates to the so-called frame problem [Sha99], but Dura is, in contrast to approaches like the event calculus [KS86], not intended to perform reasoning on the effect of actions.

ACTION SCHEMA The type and the schema of the corresponding actions are derived from the type and the schema of the stateful objects they are related to.

```
«name»$update{
  query{identifier},
  set{ «explicit attributes» }
}
```

The update action of a stateful object has the name `«name»$update` and two attributes `query` and `set`. The `query` attribute refers to a key that identifies the currently valid value of the stateful object that should be updated. The `set` attribute is a composite attribute containing all explicit attributes of the stateful object, which specifies the substitute values.

EXAMPLE The body of reactive rules that are intended to update a stateful object usually includes a query of the corresponding stateful object. The query is required to determine the key that identifies the currently valid values and which need to be specified in the `query` attribute of the update action.

The reactive rule in Listing 5.29 updates the operation mode of an area to exceptional whenever a pre-alarm is detected in this area and the operation mode is not already set to exceptional or to a higher mode. Recall that the `mode` is of type long and that `OPM_EXCEPTIONAL` is a constant associated with the value `2`.

Note that by constraining the time of the stateful object query to `end(e)`, the time the update is requested coincides with the time the values of the stateful object are queried. As a consequence, matched values that are distinguished with the key `id(s)` are guaranteed to

Listing 5.29: Updating a Stateful Object

```
ON
  and{
    event e: pre-alarm{ area{var A} },
    state s: operation-mode{ area{var A}, mode{var M} }
  } where { s valid-at end(e), var M < OPM_EXCEPTIONAL }
DO
  action a: operation-mode$update{
    query{id(s)}, set{ area{var A}, mode{OPM_EXCEPTIONAL} }
  }
END
```

be valid at the time the update is requested and therefore updates triggered by this rule are always successful.

SIMULTANEOUS UPDATES During runtime it may happen that the values of a stateful object are modified in the same instant, eg, when events representing different justifications of the same incident trigger a reactive rule. In Dura this kind of race condition is resolved by considering *all* updates requested in the same instance as successful. As a consequence, multiple updated values are concurrently valid.

Although this approach is direct and straight forward from an operational point of view, note that at least no information is lost and no kind of undetermined state arises, its implications seem quite undesirable in practice. Eg, an area that is in normal and emergency mode at the same time is not really meaningful. Details on how to handle this kine of ambiguity are described in Section 5.3.8.

5.3.5 *Creating and Terminating Values*

Besides updating values of a stateful objects, new values can be created and the validity of existing values can be terminated by means of the actions `«name»$create` and `«name»$terminate`.

The schema of the `«name»$create` action coincides with the explicit attributes of the corresponding stateful object. Whenever it is executed, a new value, carrying parameters specified for the action in its payload, becomes a valid representation of the corresponding stateful object. The opposed effect is achieved by means of the action `«name»$terminate` which expects as parameter a key that refers to a currently valid value of a stateful object. When successfully executed, the referenced values is rendered invalid.

EXAMPLE Although rarely necessary, areas can be added to and removed from the stateful object operation mode at runtime by means of the following actions:

```
operation-mode$create{ area{long}, mode{long} }
operation-mode$terminate{ query{identifier} }
```

The definition of these actions is implicitly entailed by definition of the stateful object operation mode and can be used just like regular user defined actions.

5.3.6 *Querying State Changes*

Detecting and reacting to updates of stateful objects is as important as querying the current state. For instance, the update of the operation mode to emergency entails various immediate reactions that need to be timely carried out. However, by design, the body of a rule always needs to contain at least one event query and cannot solely be composed from (one or several) queries of stateful objects. Consequently, querying changes of stateful objects in a manner that resembles queries in production system, where facts are specified in the body of rules and rules are eventually triggered when the given queries match the knowledge base, is not possible.

In Dura, state changes are indicated by means of special events that are automatically derived by the system. Whenever a stateful object has been successfully updated, created, or terminated an event referring to the affected values of the stateful object is derived by the system. Similar to actions on stateful objects, the name and schema of these kind of events is related to the stateful object they report about.

EVENT SCHEMAS For each stateful object, there are three events related to its modification, namely `«name»$updated`, `«name»$created`, and `«name»$terminated`, with the following schemas:

```
«name»$created{ payload{ «attributes» } }
«name»$terminated{ payload{ «attributes» } }
«name»$updated{
  old-payload{ «attributes» }, new-payload{ «attributes» }
}
```

Thereby `«attributes»` includes the explicit attributes of the stateful object `«name»` in addition to the attribute `id{identifier}`, which contains the key that identifies the value that has been created, terminated, or updated.

EXAMPLE The event query in Listing 5.30 matches whenever the operation mode of an area is escalated.

Note that this query can be effectively used to specify a filter on alarm events. When a fire breaks out, there are usually various justifications, observed by independent sensors, that lead to multiple detections of basically the same alarm. Accordingly, there are several alarm events that are basically caused by the same origin. And thus,

Listing 5.30: Querying State Changes

```
event e: operation-mode$updated{
  new-payload{ area{var A}, operation-mode{var Mnew} }
  old-payload{ area{var A}, operation-mode{var Mold} }
} where { var Mnew > var Mold }
```

executing immediate actions for each of them is undesirable and can cause severe consequences as the actions may interfere in an unintended manner.

By contrast, the operation mode is only rarely escalated. In particularly, it is only escalated to emergency once, regardless of the number of derived alarm events. Accordingly, triggering the immediate actions merely on the escalation of the operation mode, eg, to *emergency*, prevents the repeated execution of immediate actions.

5.3.7 *State Based Processing*

Stateful objects are well suited to model states. Instead of modeling the varying properties of concrete physical objects they can as well model properties of abstract objects, eg, the state of an entire metro station.

Moreover, the possibility to integrate queries of events and stateful objects in rules that derive events and trigger reactions enables some kind of state based processing. For instance, composite queries considering the state of the infrastructure can be used for the derivation of higher level events that are specific for the current situation, an aspect that is highly desirable for EM. In addition, queries in the body of reactive rules may also refer to the values of stateful objects. Thus, the system may propose or initiate different reactions depending on the current state and hence provide more appropriate and specific reactions.

Recall that queries of stateful objects are not limited to query the current state but can also obtain values of past states. Accordingly, rules are not limited to incorporate the current state, but can also refer to the values of past states to derive appropriate events.

EXAMPLE Recall the rule from Listing 5.29. Whenever a pre-alarm is detected, the operation mode of the corresponding area is set to *exceptional*. Although this behavior seems sound, it may be further elaborated to provide better results in border cases.

During emergencies, areas in exceptional operation mode may be easily overlooked or ignored by operators as they have to focus on those areas with major problems. As a result, a pre-alarm and its implications may not be noticed by the operators. This is sound as long as the pre-alarm does not eventually evolve into a proper alarm.

However, if the operation mode of the corresponding area has shown other problems in the past, that is, its operation mode has not been normal within the last, eg, two minutes, it is likely that there is indeed a serious issue in the corresponding area. And thus the operation mode of the area needs to be escalated to emergency.

Listing 5.31: A State Dependent Query

```
ON
  and{
    event e: pre-alarm{ area{var A} },
    state s: operation-mode{ area{var A}, mode{var M} }
    not state t: operation-mode{ area{var A}, mode{OPM_NORMAL} }
  } where { s valid-at end(e), M < OPM_EMERGENCY,
            t valid-during from-end-backward(e, 2min) }
DO
  action a: operation-mode$update{
    query{id(s)}, set{ area{var A}, mode{OPM_EMERGENCY} }
  }
END
```

Accordingly, the rule from Listing 5.29 is supplemented with the one contained in Listing 5.31. It controls, in addition to the current operation mode, whether the operation mode of the corresponding area has not been normal up until two minutes before the occurrence of the pre-alarm, and, if so, escalates the operation mode to emergency directly. In this way, pre-alarms are treated more seriously depending on the values of past states.

Note that the rules in Listing 5.29 and Listing 5.31 are conflicting in the sense that they may simultaneously trigger and cause two updates of the operation mode which eventually results in the operation modes exceptional and emergency being valid for the same area at the same time which is clearly undesirable. This aspect of stateful objects and means to avoid it are covered in more detail in the following.

5.3.8 *Resolving Simultaneous Updates*

When two reactive rules are requesting updates of the same stateful object in the same instant, both updates are successful and as a result both updated values are valid at the same time. For instance, updating the operation mode for an area in the same moment may render the area in normal and emergency mode at the same time. Although this behavior provides a clear and robust semantic from an operational point of view, note that no information is lost and no kind of undetermined unforeseeable state arises, the unusual implications on a semantical level need to be taken care of.

There are basically two different ways in which this issue can be addressed, namely avoiding conflicting updates and specifying selection criterion that choose a suited result from the conflicting values.

AVOIDING CONFLICTING QUERIES The queries in the body of conflicting rules can always be adapted so that simultaneous updates of the same stateful object are semantically impossible. This approach is not specific for Dura and applies also for other languages.

Listing 5.32: Adapted Query from Listing 5.29

```
and{
  event e: pre-alarm{ area{var A} },
  state s: operation-mode{ area{var A}, mode{var M} }
  state t: operation-mode{ area{var A}, mode{OPM_NORMAL} }
} where { s valid-at end(e), var M < OPM_EXCEPTIONAL,
          t valid-during from-end-backward(e, 2min) }
```

Listing 5.32 shows one possibility of how to resolve the conflict between the rules of Listing 5.31 and Listing 5.29. This is achieved by extending the query from Listing 5.29 with a positive query for a normal operation mode. Note that the two composite queries only differ in whether the query for the stateful object `t` is negated or not and therefore it is guaranteed that only one of them matches.

Although this is a very generic way to resolve conflicts, it requires complete knowledge of the entire program and is not very robust with respect to modifications of the program. When rules are adapted or new rules are added to a program, new conflicts may be introduced that need to be resolved again. In particular for large programs this can be a very challenging and error prone task. Moreover, programmers can easily introduce errors to the rules that unintentionally alter the semantics of the program or do not completely eliminate the possibility of simultaneous updates.

CHOOSING AMONG CONFLICTING VALUES Conflicting updates can also be resolved by looking at the parameters of all conflicting updates and selecting the most suited one according to its parameters. For instance, when multiple updates of the operation mode are concurrently requested, it seems reasonable from an EM perspective to determine the result by the highest mode that is specified among all conflicting updates. To this end, the schema of stateful objects can be associated with a `on conflict select` part that specifies a selection criterion that applies in case of conflicting updates to the respective stateful object.

When an `operation-mode$update` action is requested, the attributes of the updated value, including its implicit key, are determined by the parameters of the `update` action. In case of the stateful object `operation-mode` the update action determines the attributes `area`, `mode`,

and `id` of the updated value. The `on conflict select` part then specifies which `update` prevails in case multiple `update` actions are concurrently issued. To this end, it it refers to the attributes that are determined by the `update` action and specifies an order among update actions by means of the operators `max` and `min`.

Listing 5.33: Stateful Object Schema with Conflict Resolution

```
operation-mode{ area{long}, mode{long} }
on conflict select { max mode, min id }
```

The selection criteria in Listing 5.33 specifies that the `update` that determines the highest `mode` prevails. Moreover, if there are several concurrent actions updating the stateful object to the same `mode` the one that determines the highest `id` prevails. As the `id` is unique, it is guaranteed that only one action is actually successful and thus that merely a single updated value is created.

Be aware that the `id` attribute is more or less arbitrarily determined. But as the selection criterion ensures that all conflicting `update` actions determine the same mode it is not further relevant which of them is actually successful as, except for the `id`, their result does not differ.

Using the actual data to resolve conflicts has turned out to be both practical and convenient to resolve conflicting updates. In fact, we learned from the EMILI use cases [SB10b; SBR11a] that in general a conservative assessment of the current situation is preferred over a too optimistic situation assessment in the domain of EM. Accordingly, choosing the highest of all conflicting operation modes is a suitable approach to obtain an unambiguous operation mode for each area.

5.3.9 *A Generalization of ECA Rules*

Event-Condition-Action (ECA) rules have been introduced in the context of active databases [PD99] where they are commonly applied to anticipate and react to changes of the database. Conventional ECA rules consists of three distinct parts, namely an event, condition, and action part.

```
ON «event» IF «condition» DO «action»
```

The specified action is executed when the event query matches and in the same instant the condition holds. Thereby, shared variables are used to exchange variable bindings between the different parts of the rule. However, each rule part is evaluated independently from the others and there are no means to couple the distinct parts more closely and to affect the time of their evaluation.

Composite queries and reactive rules in Dura are convenient to express ECA rules. And due to the homogeneous integration of queries of events and stateful objects, reactive rules offer more flexibility and expressiveness over traditional ECA rules.

ECA RULES IN DURA ECA rules are expressed by means of reactive rules. The event and condition part of ECA rules are formulated by means of a composite query containing queries of events and stateful objects, respectively. As a consequence, the event and condition part are no longer separated and hence the resulting rules are more flexible and expressive than common ECA rules as, eg, temporal conditions on stateful objects can be specified relative to the time of queried events.

Listing 5.34: An ECA Rule Equivalent in Dura

```
ON
  and{
    event e: alert{ area{var A} },
    state s: operation-mode{ area{var A}, mode{OPM_EMERGENCY} }
  } where { s valid-at end(e) }
DO
  action a: ...
END
```

The reactive rule from Listing 5.34 resembles a conventional ECA rule that executes an action `a` if an alert occurs and the operation mode of the corresponding area is set to emergency in the instance the alert is observed. Note that the time of the queried stateful object is constrained in the `where` part of the query to correspond to the end of the alert event. Thus, the valid time of matched values corresponds to the time the condition of a conventional ECA rule is evaluated.

A GENERALIZATION OF ECA RULES The temporal conditions on stateful objects offer substantial control over the time when stateful objects are queried. Hence, the time of the evaluation of the condition part can be adapted to the needs of the programmer and is not fixed to the time the event query matches as in case of ECA rules. Moreover, multiple stateful objects can be queried that are related to the time of different events.

The rule in Listing 5.35 triggers an action whenever an alarm event occurs and the operation mode of the corresponding area has been emergency some time *during* the two minutes before the alarm.

5.3.10 *Processing Static Data*

Static data is desirable to represent information that is known in advance and does not change at runtime. This includes static domain

Listing 5.35: A Generalized ECA Rule

```
ON
  and{
    event e: alert{ area{var A} },
    state s: operation-mode{ area{var A}, mode{OPM_EMERGENCY} }
  } where { s valid-during from-end-backward(e, 2min) }
DO
  action a: ...
END
```

knowledge that does not need to be updated, or at least only rarely needs updates during certain maintenance periods. Common examples from the use cases described in [SB10b; SBR11a] are, for instance, topology information such as the structure of metro stations or the topology of a power distribution network, and mappings that associate the identifier of a sensor with its type and position in the infrastructure, etc.

EXAMPLE Static data is valuable, for instance, to enrich the information provided by raw events directly received from sensors. The rule in Listing 5.36 uses the domain knowledge contained in the static object `sensor` to convert a raw sensor event carrying only an identifier and a value into a higher-level event with a proper and meaningful type that also reports the area the corresponding sensor is located in.

Listing 5.36: A Query of a Static Stateful Object

```
DETECT
  temp{ area{var A}, value{var T}, sensor-id{var S} }
ON
  and{
    event e: value-update{ sensor{var S}, value{var T} },
    state s: sensor{ sensor-id{var S}, area{var A}, type{29} }
  }
END
```

As the stateful object is static, there is no need to constrain the valid time for matching values. In fact, constraining the valid time is not possible for static stateful objects at all.

NEGATION AND GROUPING OF STATIC DATA As static stateful objects cannot be modified, the "history" of their values is always completely determined at runtime. In consequence, negated queries of stateful objects or queries containing grouping over stateful objects do not need to be temporally bounded as queries do not need to be deferred because all their values are completely known by the EPS and thus even unbounded queries can be completely and correctly evaluated.

Complex actions are a central notion to realize composite high-level reactions in Dura. They are intended to specify composite reactions based on physical actions executed by external actuators. To this end, they aim to specify the desired interactions between different actuators in a way that is suitable for the physical nature of actions as it is required for EM purposes. They are, however, not meant to implement arbitrary computations as they are realized, eg, by means of imperative procedures in general purpose languages.

Complex actions are composed from basic actions by means of temporal dependencies and specifications of their execution result Moreover, complex actions benefit from a homogeneous integration into the event query language Dura that provides convenient access to high level event queries and queries for stateful objects.

5.4.1 *Properties of Physical Actions*

Physical infrastructures are equipped with various actuators that are scattered over the entire infrastructure and operated from a central control room. To this end, the actuators are attached to a Supervisory Control and Data Acquisition (SCADA) system that connects the actuators and the control room so that commands and information can be exchanged. SCADA systems are very specialized systems that are optimized for the operation of technical facilities [DS99; Boy09]. On a more abstract level, they resemble a specialized Event Service Bus (ESB) that establishes a communication between physical sensors and actuators and a central control room. Beyond that, modern SCADA systems provide limited filtering and automation capabilities and use open XML based protocols, such as the Facility Control Markup Language (FCML) [Bry+08].

The execution of physical actions is a very basic requirement for the supervision and operation of physical infrastructures. However, physical actions are not directly executed by the EPS but instead their execution is merely requested by the system and the actual execution is realized by external actuators. To this end, request messages are derived by the system and subsequently delivered to the actuators by means of a SCADA or ESB where the corresponding action is actually executed. This particularity implies some rather unusual properties of physical actions that are accounted for in Dura to obtain a suitable and convenient model of actions in the language.

TIMES OF ACTIONS Physical actions are executed by external actuators and thus the time an action is requested within the EPS and the time it is actually executed by the actuator may substantially differ.

Likewise, the time an action is completed and the time the according information is available in the EPS may differ as well.

For instance, when the query of a rule triggering an external action is satisfied, firstly the corresponding action request needs to be derived by the system. Subsequently, the request needs to be delivered to the actuator and finally the actuator needs to trigger the execution of the corresponding action. All three steps introduce a certain delay to the execution of the action and thus contribute to the latency which represents the time difference between the first moment the action can be initiated according to the specification of the query in the body of the reactive rule and the moment the action is actually initiated by the actuator.

To reduce the impact of latency and network delays, it seems furthermore desirable that the time of external actions is determined within the actuator and not by the system. Note that this adheres to the approach to prefer application over system time for the time of events. Employing application time for events means that the time of events is determined at the sensors and not by the system. Likewise, the initiation of the action is carried out by the actuator and thus the time for the initiation on the action should be determined by the actuator as well.

INDIRECT FEEDBACK Physical actions are requested within the EPS and executed by external actuators. As a consequence, the system has inherently no general knowledge on the progress of requested actions. It has no information when the action is initiated and when it is completed. Therefore, it depends on the feedback that is provided by the corresponding actuators to determine the current status of running actions. However, due to the heterogeneity of actuators in large infrastructures, the quality of feedback may vary substantially. For instance, not every actuator can provide the feedback that is desired to determine when and if an action was actually successful, some even cannot provide any feedback at all.

As a consequence, it is often mandatory to rely on indirect feedback from related sensors that allows inferences on the execution status of actions. For instance, when no direct feedback on the successful activation of a ventilator is provided by the actuator, one can use the information on the current airflow measured by an adjacent anemometer to derive this information indirectly. If even no indirect feedback is available, domain knowledge can be used to specify, for instance, that the activation of the ventilator is successful 20 seconds after the request was emitted by the EPS. Note that although feasible, the last alternative only rarely results in a valuable representation of the action.

In summary, it is desirable to obtain the feedback on the execution of actions directly from the actuator, but due to technical limitations

the feedback may need to be inferred from other sources including very generic information provided by the EPS itself.

5.4.2 *Representation of Actions*

Actions are formalized in a way similar to the representation of events and stateful objects. Each action has a type and carries a payload of data, namely its parameters. Actions are represented by means of Xcerpt construct terms that conforms to the Dura specific restrictions.

PAYLOAD The payload of an action contains its concrete parameters that are determined by the rule that triggers its execution. Moreover, each action instance is associated with a key uniquely identifying each action instance and with provenance information that is determined by the rule that triggers the action. The two values are represented in the payload by means of the atomic attributes `id{`*`identifier`*`}` and `ref{`*`identifier`*`}`, respectively, and are implicitly added to the schema of each action.[4]

Both attributes are determined by the EPS and do not need to be specified by the programmer.

TIMES ASSOCIATED WITH ACTIONS Each action is related to three times, namely, the initiation time, the time it is successful and the time it fails. Naturally, actions either succeed or fail and thus it is reasonable that only one of the latter two time values is actually determined at runtime.

As it has been discussed above, the times associated with actions should be determined by the actuators. However, on the other hand, actuators may unfortunately not be capable of reporting the desired feedback so that indirect feedback needs to be used to derive the status and hence the values of the corresponding times. Accordingly, the means for the specification of the status and times of actions need to be generic enough to cover the complete spectrum of required specifications.

To this end, the initiation, success and failure of actions is specified in Dura by means of event queries. As all observable information that is available to the EPS is, by design, represented by means of events, event queries are a suitable and expressive way for the specification of those properties. The corresponding queries are associated with the schema of the action in a distinct `initiates on`, `succeeds on`, and `fails on` part.

4 Note however, that the times of actions are, in contrast to the times of events, not explicitly represented in their payloads.

```
«action schema»
initiates on «event query»
succeeds on «event query»
fails on «event query»
```

Either of the three parts is optional and for convenience, the initiation of an action defaults to the time the action is requested in case the `initiates on` part is omitted. In contrast, actions without a `succeeds on` or `fails on` part never succeed and never fail, respectively. Thus, referencing times for the success and failure of actions that have not been specified in their schema by means of event queries causes an error in the syntactic analysis at compile time.

Be aware that an action initiates, succeeds, and fails whenever the respective event query matches. Therefore, the programmer is responsible for the specification of appropriate queries, ie, queries for the success and failure that are mutually exclusive and match only once. And extension of Dura that avoids these issues is discussed in Chapter 13, though.

REFERENCING THE TIMES OF ACTIONS Similar to the key and the time of events which can be concisely referenced in the `where` part of event queries by means of the functions `id`, `begin`, and `end`, the time of actions can be concisely referenced by means of the null-ary functions `req` and `init` in the schema of actions. Thereby, `init` can only be specified in a `succeeds on` or `fails on` part.

As the schema of action specifies the properties of a single action, the functions unambiguously reference the respective time of the action and are thus specified without further parameters. However, be aware that the same functions are reused for composite actions where they expect an action identifier as parameter.

EXAMPLE In the following Listing 5.37, the schema of the activate lights action is elaborated. This action turns on the ceiling lighting within a certain area.

Listing 5.37: Specifying Success of a Physical Action

```
activate-em-light{ light{long} }
succeeds on and{
    event e: action-confirmation{ instance{id()} }
  } where { end(e) <= init() + 10sec }
fails on not{
    event e: action-confirmation{ instance{id()} }
  } where { end(e) <= init() + 10sec }
```

However, the corresponding actuator is only capable of acknowledging the reception of the action request but not the initiation and the success of the action. As the action is not very likely to fail and because of the lack of other means that are capable to indirectly confirm

the success of the action, the acknowledgment is indeed interpreted as the success of the action. Note that the specification for the initiation of the action is omitted from the schema. As a consequence the initiation of the action defaults to the time the action is request by the system.

EVENTS ENTAILED BY ACTIONS The execution of actions entails, similar to the modification of stateful objects, events that can be contained in the queries of rules to react to their initiation, success, or failure.

The name and the payload of the events is derived from the schema of the corresponding actions. For each action there are four different events, namely, `«name»$requested`, `«name»$initiated`, `«name»$succeeded`, and `«name»$failed`. Thereby the end of the reception time of the events refers to the corresponding time of the action. For instance, the end of the reception time of the `«name»$initiated` event refers to the initiation time of the respective action instance.

The described events of an action share a common schema that contains the composite attribute `payload` consisting of the attributes of the action, including the implicit attributes `id` referring to a key that uniquely identifies the action instance, and the implicit attribute `ref` containing the provenance information on the action that is required for internal purposes. Accordingly, the concrete parameters of each action instance can be obtained from each of these events.

Listing 5.38: Schema of `activate-em-light$initiated` Events

```
activate-em-light$initiated{
  id{identifier},
  reception-time{ begin{timestamp}, end{timestamp} },
  payload{ id{identifier}, ref{identifier}, light{long} }
}
```

Listing 5.38 describes the schema of events representing the initiation of the `activate-em-light` action from Listing 5.37. Mind the different between the two `id` attributes. The `id` contained in the `payload` attribute corresponds to the key of the action whereas the `id` directly contained in the event type corresponds to the key of the event reporting about the action instance.

REPRESENTATION OF DOMAIN KNOWLEDGE In addition to the specification of the initiation and success of actions, the schema of actions can also specify domain knowledge, eg, on the duration of actions, that is known to apply for all instances of the corresponding action.

Listing 5.39: Specifying Domain Knowledge

```
activate-em-light{ light{long} }
  where{ init() + 2sec <= succ(), init() + 2sec <= fail() }
```

The domain knowledge of actions is specified by means of a conjunction of inequations in a `where` part associated to their schema.[5] For instance, the domain knowledge specified in Listing 5.39 denotes that the execution of `activate-em-light` takes at least two seconds.

5.4.3 *Action Invocation*

The syntax for the initiation of actions resembles the syntax for queries of events and stateful objects. The keyword *`action`* is succeeded by an alphanumeric identifier and a construct term determining the name and the parameters of the action.

Listing 5.40: An Atomic Action Invocation

```
action a: adapt-ventilation{ area{var A} }
```

Instead of extracting values from the pattern, as in case of event queries, the values bound to the variables are injected to the corresponding actions as parameters. Accordingly, the specification of the action resembles a construct term and therefore generic expressions can be specified. However, the applied variables and identifiers need to be positively bound with respect to the query in the body of the corresponding reactive rule.

For the sake of simplicity, the action type and its identifier will be used interchangeably. For instance, the identifier `a` will be used as an abbreviation for the action `adapt-ventilation`.

5.4.4 *Action Composition*

The composition of actions is crucial to obtain useful reactions that cannot be realized by means of the execution of single atomic actions. In fact, one of the main advantages of using a central EPS is the integration of different subsystems that are controlled by independent sub-systems. In this way, the provided information and capabilities can be combined to realize more powerful and appropriate reactions that are based on the execution of compositions of actions.

Composition of complex actions is expressed in a manner similar to the composition of event queries. Several actions are grouped to-

5 Be aware that the keyword `where` is used in the schema as well as for the composition of actions with different semantics, namely, to specify domain knowledge of and temporal dependencies between actions, respectively.

gether by means of the `compound` operator. At runtime, the given actions are then executed concurrently. Note that the compound operator does not specify any temporal dependencies between actions nor does it determine when the composite action is deemed successful or failed as these are different aspects that should be kept separate in order to obtain a clear separation of concerns.

EXAMPLE The action in Listing 5.41 represents a composition of the actions `open-fire-damper` and `activate-ventilators`. When it is executed, both sub-actions are executed concurrently and the value bound to the variable `A` is injected as parameter to of both actions.

Listing 5.41: A Composite Action

```
compound{
  action a: open-fire-dampers{ area{var A} },
  action b: activate-ventilators{ area{var A} }
}
```

SEPARATION OF DIMENSIONS Due to the clear separation of orthogonal dimensions, `compound` is the only available operator that is required for the composition of actions. It just specifies the actions that are intended to be executed concurrently and provides concrete values for their parameters by means of variables, but does not introduce further, eg, temporal dependencies between actions.

Other approaches often introduce composition operators that combine several independent dimensions of actions into a single monolithic operator. For instance, a sequence operator actually combines specification of the composition and temporal dependencies of an action in a single operator. Although monolithic operators seem convenient at first sight, previous research has shown that languages build on monolithic operators loose expressiveness compared to languages with a clear separation of dimensions [BE08b]. Even though this observation has been made for event query languages, it still holds for actions as it is further discussed in the next section.

Note that, in contrast to event queries, even conjunctions and disjunctions of actions actually represent a combination of several dimensions. As, for instance, a conjunction of actions combines the specification of a composition with the specification of temporal dependencies and the execution result of the composite action.

5.4.5 *Temporal Dependencies*

The composition of physical actions, in contrast to internal actions, often requires a certain timing between sub-actions as the effect of initiating actions too early or too late may result in unintended be-

havior that does not achieve the intended goal. Therefore, complex actions require means to specify temporal dependencies between different actions.

Temporal dependencies between actions are specified in the `where` part of complex actions. They refer to the actions from the separate `compound` part by means of the action identifiers and specify a certain timing of actions that needs to be satisfied by the EPS at runtime. Temporal dependencies are specified by means of conjunctions of inequations, similar to the specification of temporal conditions between events, that determine lower bounds for the initiation of actions. To this end, action identifiers are used in combination with `init`, `succ`, and `fail` to distinguish between the different time-points that are associated with the actions. Thereby, referencing a time that is not specified in the schema of the corresponding action by means of an event query causes an error in the syntactic analysis.

For other applications it seems appropriate to furthermore support disjunctions of temporal dependencies that enable non-deterministic actions. However, for EM applications the determinism of actions is a crucial requirement and as a consequence, disjunctive dependencies are not further considered here although they can be integrated in our approach (see Section 13.3).

EXAMPLE The complex action in Listing 5.42 uses temporal dependencies to specify that the ventilators should only be activated after the fire dampers have been successfully opened. Note that this behavior corresponds to a simple sequence of actions that can also be expressed by many other approaches with a notion of composite actions.

Listing 5.42: A Sequence of Actions in Dura

```
compound{
  action a: open-fire-dampers{ area{var A} },
  action b: activate-ventilators{ area{var A} }
} where { succ(a) <= init(b) }
```

Although the clear separation of dimensions seems cumbersome and longish, it enables the elaboration of more sophisticated dependencies that cannot be expressed with monolithic sequence operators. Consider, for instance, the composite action in Listing 5.43. The complex action from above is complemented by a third sub-action that warns persons close to the outlet of the ventilation system of the imminent emission of smoke. To be effective, a time delay of 20 seconds between the issue of the warning and the actual emission of smoke is added to the temporal dependencies of the action.

The same action cannot be expressed by formalisms relying solely on monolithic sequence operators as they mix the invocation of ac-

Listing 5.43: An Action with Sophisticated Dependencies

```
compound{
  action a: open-fire-dampers{ area{var A} },
  action b: activate-ventilators{ area{var A} },
  action c: warn-of-smoke-emission{ area{var A} }
} where { succ(a) <= init(b),
          init(c) + 20sec <= init(b) }
```

tions and the specification of temporal dependencies. As a consequence, those formalisms can specify that `a` and `c` are executed concurrently 20 seconds before the initiation of `c` but they cannot enforce independent dependencies between `a` and `b` and between `c` and `b`, as they are for instance specified in Listing 5.43.

SATISFYING TEMPORAL DEPENDENCIES At runtime complex actions are actually executed by the EPS according to their temporal dependencies. To this end, the system defers actions whose initiation is explicitly specified in the right hand side of temporal dependencies until the lower bound specified by the left hand side is exceeded. In this way, the system can guarantee that the temporal dependencies are actually satisfied at runtime independent from any, a priori unknown, runtime effects.

The execution of complex actions can be understood as some kind of feedback loop. The EPS requests the execution of actions which eventually results in the observation of their success or failure. This, in turn, determines the lower bounds for the initiation of further actions which will be eventually exceeded and thus these actions will be requested for execution and so forth.

```
where { succ(a) <= init(b) }
```

To satisfy the preceding temporal dependency, which is taken from Listing 5.42, the system simply defers the initiation of `b` until the success of `a` has been observed. When several dependencies for the initiation of an action are specified, the system simply defers its execution until all of them are satisfied. By contrast, the inequation `succ(a) <= succ(b)` does not constrain the initiation of actions and can hence only be observed at runtime. Accordingly, this type of dependencies covers another aspect of actions as they rather specify the success of the action which is determined *after* the action is initiated. Therefore, the given constraints cannot be specified in the `where` part of actions.

In summary, temporal dependencies specify lower bounds for the initiation of actions that need to be exceeded before the action is requested by the system. Although the available constraints seem rather limited, we will determine that due to requirements on complex actions and the inherent properties of physical actions other inequations cannot be satisfied in general.

RELATIONS TO SCHEDULING AND PLANNING In general, implicit assumptions are avoided in our approach as they influence the semantics of actions in a way that can easily be overlooked by programmers. Accordingly, the system may not implicitly assume dependencies that are not explicitly specified by the user. This contrasts strongly with approaches concerned with the planning and scheduling of the execution of actions [TVP03; VF99] which try to determine strategies of executing actions by adapting the execution strategy in accordance with the given temporal dependencies and are thus not further considered in the scope of this thesis.

For instance, some of these approaches derive `succ(a) <= init(b)` from the in Dura invalid dependencies `succ(a) <= succ(b)`. When the derived dependencies are interpreted as the temporal dependencies

```
where { succ(a) <= init(b) }
```

it is indeed guarantee that `succ(a) <= succ(b)` is always satisfied at runtime. However, the derived temporal dependencies introduce an implicit relation between the success of `a` and the initiation of `b` which, if missed by the programmer, can cause unintended effects and is therefore undesirable for EM purposes.

Note that, the analysis of complex actions in Dura can actually verify that `succ(a) <= succ(b)` is always satisfies if the action is executed according to the dependencies `succ(a) <= init(b)`. However, to obtain a clear separation of these different aspects of actions, the two types of formulas are clearly separated and specified in two different parts which contain either temporal dependencies that specify how to execute actions or temporal assertions that follow from the given temporal dependencies.

5.4.6 *Execution Status*

Complex actions usually try to achieve a higher level goal that cannot be realized by individual actions. Naturally, the success of the action depends on the achievement of this goal which often cannot be inferred from the raw success of the sub-actions. As a consequence, Dura employs versatile event queries to specify the success of actions beyond the success of their sub-actions.

Like the specification of the execution status of physical actions, the success and failure of complex actions is specified in dedicated `succeeds on` and `fails on` parts by means of event queries. Thereby, the `where` part of the event query specifying the success of the action may refer to the time-points of sub-actions by means of the action identifiers specified in the composite action.

EXAMPLE The action from Listing 5.42 is intended to extract smoke from a certain area. Accordingly the action is deemed successful if the

smoke concentration drops below 10% in the respective area within two minutes from the beginning of its execution and fails otherwise.

Listing 5.44: Specifying the Success of Complex Actions

```
compound{
  action a: open-fire-dampers{ area{var A} },
  action b: activate-ventilators{ area{var A} }
} where { succ(a) <= init(b) }
  succeeds on and{
      event e: smoke{ area{var A}, amount{var C} }
    } where { C < 0.1, init(a) < end(e), end(e) < init(a)+2min }
```

Note that the `where` part of the complex action and the `where` part of the query in the `succeeds on` part are both referring to the action `open-fire-damper` by means of the identifier `a`.

INITIATION TIME As composite actions are executed by the EPS their initiation time defaults to the moment they are requested by the system. However, this behavior can be adapted to the needs of the programmer by explicitly specifying the desired time-point in an `initiates on` part associated with the composite action.

5.4.7 *Temporal Assertions*

Temporal assertions are formulas that specify properties of actions that should hold at runtime but do not directly specify an execution order between actions and are statically verified at compile time. They are intended as a means for programmers to statically verify whether the specified assertions which they expect to be satisfied at runtime are actually satisfied at runtime. Moreover, temporal assertions contribute to a clear separation of dimensions, as they separate temporal dependencies that influence the execution order of actions and are specified in the `where` part of actions from inequations that are merely the consequence of the given execution order.

Temporal assertions are specified in the `hence` part of complex actions. They are denoted by means of inequations that resemble the specification of temporal dependencies but are not limited to specify merely lower bounds for the initiation of actions. Assertions that do not follow from the temporal dependencies of an action and thus do not necessarily hold at runtime are statically detected during compile time and result in compilation errors.

EXAMPLE The assertions in Listing 5.45 specifies that the sequential execution of the `open-fire-dampers` and `activate-ventilators` actions takes at least 20 seconds if the execution of both actions is successful and at least one minute if `a` is successful and `b` fails.

Recall that assertions do not influence the execution of actions and are just verified at compile time. Accordingly, it is admissible that both formulas specify an upper bound for the initialization of `a`.

Listing 5.45: A Complex Action with Assertions

```
compound{
  action a: open-fire-dampers{ area{var A} },
  action b: activate-ventilators{ area{var A} }
} where { succ(a) <= init(b) }
  hence { succ(b)-init(a) > 20sec, fail(b)-init(a) > 1min }
```

Note that these assertions do not immediately follow from the temporal conditions in the `where` part. To actually verify this particular assertion, reliable information on the duration of both actions needs to be available, eg, in the schema of the corresponding actions.

5.4.8 *Semantic Analysis for Actions*

The clear separation of concerns and a good coverage of the four orthogonal dimensions of actions is clearly desirable for complex actions, in particular for complex actions in EM. However, due to the separation of dimensions and the specification of temporal dependencies and assertions by means of generic formulas, the programmer is able of specifying actions that are syntactically correct but either cannot be executed due to inconsistencies in the temporal dependencies or contain temporal assertions that simply do not follow from the specified temporal dependencies and thus are not guaranteed to hold at runtime.

As a consequence, a static analysis capable of rejecting incorrect and faulty actions is highly desirable to prevent errors that would arise during the execution of actions at runtime. Therefore the semantic analysis of complex actions should cover at least the three following aspects.

VIABILITY Atomic temporal dependencies need to be viable in the sense that they can be actually satisfied by the system at runtime. As the referred time-points of actions can only be affected indirectly, arbitrary temporal dependencies between actions may not necessarily be satisfiable at runtime.

For instance, due to inherent runtime effects, it is impossible for the system to guarantee that two actions are actually initiated at the same time, because the distribution of the action request to the actuator is subject to latency. Note that this still holds if the initiation refers to the time the request is emitted by the system as the derivation of the request is also subject to latency.

FAIRNESS Using action identifiers and generic relations to specify temporal dependencies between actions seems desirable. In fact, it is even mandatory to facilitate the integration of multiple independent dimensions without losing expressiveness.

However, the flexibility of temporal dependencies may result in inconsistent specifications, for instance in cyclic dependencies between actions, that prevent sub-actions from being executed. Accordingly, the semantic analysis must verify at compile time whether the temporal dependencies allow that all sub-actions can actually be executed at runtime and that there is no "orphan code" that contains actions which will never be executed.

COMPLIANCE Specifying temporal assertions is only meaningful if it is verified during compile time that the corresponding assertions of an action will actually be satisfied at runtime, that is, that the complex action complies with its assertions.

However, the verification often requires domain knowledge that might not be available for all kinds of physical actions. Accordingly, it is mandatory that the semantic analysis scales with the available knowledge about actions. The analysis must be able to verify basic properties without further action specific knowledge but at the same time needs to be able to incorporate domain knowledge, e.g., specified in the schema of actions, if it is available.

5.4.9 *Complex Action Rules*

For large programs it is desirable to minimize redundancy among rules to obtain reusable and clear code that can be easily maintained and adapted. In case of events, this requirement is realized by deductive rules that derive higher level events which can be queried by multiple rules. Likewise, composite actions can be made reusable by means of so-called complex action rules.

Complex action rules assign names to (anonymous) complex actions in a way that resembles procedures that assign names to certain fragments of code. They are denoted

```
FOR «name»{ «parameters» } DO «action» END
```

Thereby, the parameters are specified by terms that contain variables which are to be injected when the corresponding action is initiated.

EXAMPLE A complex action rule is used in Listing 5.46 to assign the name `adapt-ventilation` to the anonymous complex action that has been elaborated in previous examples. Accordingly, the complex action in the head of the rule can be executed by referring to its name `adapt-ventilation` rather than by repeating the entire code wherever the respective behavior is required.

Listing 5.46: A Complex Action Rule

```
FOR
  adapt-ventilation{ area{var A} }
DO
  compound{
    action a: open-fire-dampers{ area{var A} },
    action b: activate-ventilators{ area{var A} }
  } where { succ(a) <= init(b) }
    succeeds on and{
        event e: smoke{ area{var A}, amount{var C} }
      } where { C < 0.1, init(a) < end(e), end(e) < init(a)+2min }
END
```

Note that variables that occur in the body of the complex action rule, that is, the parameters of the action, are bound when the complex action is invoked. Therefore, these variables can be used as parameters for the sub-actions specified in the head of the complex action rule and in the event queries determining its success and failure of the complex action.

ACTION IDENTIFIER FOR COMPLEX ACTIONS For the specification of complex action rules it is sometimes desirable to reference the initiation time or the key of the complex action, eg, to specify temporal constraints for its sub-actions or in the queries determining its success and failure, respectively.

To this end, an action identifier can be specified in the body of a complex action rule to refer to the initiation of the respective complex action. The identifier can subsequently be used in the supplement of a complex action in combination with `init`, but not with `succ` or `fail`, in the `where` part of a event query determining the success or failure of the complex action. Moreover, the identifier can also be specified in the `group by` part of an event query.

Listing 5.47: A Complex Action Rule with Unique Success

```
FOR
  action v: adapt-ventilation{ area{var A} }
DO
  compound{
    action a: open-fire-dampers{ area{var A} },
    action b: activate-ventilators{ area{var A} }
  } where { succ(a) <= init(b) }
    succeeds on and{
        event e: smoke{ area{var A}, amount{var C} }
      } where { C < 0.1, init(v) < end(e), end(e) < init(v)+2min }
        group by { v }
END
```

The identifier `v` is used in Listing 5.47 in the `where` and `group by` part of the event query to refer to the initiation of the complex action `adapt-ventilation`. By means of the grouping it is guaranteed that the action is only successful once, whereas the action in Listing 5.46 is as often successful as the smoke concentration drops below 10% within one minute from the initiation of the `open-fire-dampers` actions, which seems rather undesirable as it may happen multiple times. Note, however, that in this particular case, the same behavior can be obtained by omitting *`action`* `v` from the rule body and by substituting `a` for `v` everywhere else.

IMPLICIT PROVENANCE INFORMATION Each sub-action is linked to the instance of the complex action that caused its initiation. More precisely, the key identifying the complex action instance is reference by the `ref` attribute in the payload of each initiated sub-action.

Listing 5.48: Specifying Success Dependent on Sub-actions

```
FOR
  action v: adapt-ventilation{ area{var A} }
DO
  compound{
    action a: open-fire-dampers{ area{var A} },
    action b: activate-ventilators{ area{var A} }
  } where { succ(a) <= init(b) }
    succeeds on and{
      event e: activate-ventilators$succeeded{ payload{ ref{id(v)} } }
    } where { end(e) <= init(v) + 1min }
END
```

For instance, when initiated by the complex action in Listing 5.48, the `ref` attribute of the `open-fire-dampers` and `activate-ventilators` actions refers to the `id` of the corresponding `adapt-ventilation` instance that caused their initiation. This property is utilized in the `succeeds on` part of the `adapt-ventilation` action to specify its success relative to the success of the `activat-ventilators` actions that has been initiated by the respective complex action. To this end, the expression `id(v)` is specified in the event query in the `succeeds on` part. Note that without this restriction, the query would match arbitrary `activate-ventilator` instances, in particular those that are not related to the complex actions and were by chance executed concurrently.

In this particular case, the same effect can be achieved by substituting `id{id(b)}` for `ref{id(v)}`, though.

5.4.10 *Conditional Actions*

Temporal dependencies of complex actions specify the execution order of actions relative to their initiation, success, and failure. However,

in some situations it may be more suitable to specify the execution of actions in relation to the current state of the infrastructure, static data, or the occurrence of certain events instead of referring to the initiation and success of preceding actions.

Conditional actions realize this requirement by providing a deep integration between actions and queries for events and stateful objects. They are denoted

```
IF «query» THEN «action»
```

The `IF` part specifies a condition that constraints the execution of the actions in the `THEN` part: Whenever the composite action containing the conditional action is initiated and the query in the `IF` part matches, the action in the `THEN` part are executed. Thereby, `«query»` corresponds to a conventional event query or to a query for stateful objects.

Be aware that the query in the `IF` part may indeed match multiple times and that hence the corresponding actions are initiated multiples times as well. This can be avoided by constraining the query appropriately, eg, by constraining the time in which the condition is evaluated or by applying grouping. To this end, the event query can refer to sub-actions, in particular to their times, of the composite action the query is contained in.

By means of conditional actions generic conditions for the execution of actions, which are exceeding the capabilities of merely temporal dependencies, are specified by event queries. Similar to reactive rules, conditional actions aim at a tight integration between the pure declarative world of events and stateful objects and the imperative world of actions by combining both notions in the specification of conditional actions. Naturally, conditional actions can be specified in composite actions. Moreover, they are a means to obtain a hierarchical composition of actions as it is often required for the specification of composite processes.

Note that the functionality of conditional actions can as well be realized by several reactive rules. However, using one single complex actions to specify composite processes is clearly preferable over a set of (more or less) independent rules that may be scattered over a large program, as it is for instance proposed in [KEP00; Bry+06].

EXAMPLE To obtain permission for the execution of an action that requires the confirmation of a human operator prior to its execution, a `request-operator-confirmation` action is executed which eventually displays a corresponding message in the graphical user interface used by operators in the control room. When the operator accepts the proposed action and confirms its execution, a `request-confirmed` event is sent from the user interface back to the EPS and the proposed action is subsequently initiated.

This behavior is implemented by means of the composite action in Listing 5.49. To this end, a conditional action is used to defer

Listing 5.49: A Simplified Conditional Action

```
compound{
  action a: request-operator-confirmation{ ... }

  IF event e: request-confirmed{ req-id{id(a)} }
       where { end(e)-init(a) <= 20sec }
  THEN action b: ...
}
```

the execution of the proposed action `b` until its execution is actually confirmed by the operator by means of a corresponding request confirmed event that needs to be issued within 20 seconds after the initiation of the request action.

Note how the event query in the `IF` part of the conditional action refers to the `request-operator-confirmation` action of the complex action. In the `where` part of the event query the expression `init(a)` is used to constrain the time window in which confirmations are accepted. Moreover, the expression `id(a)` is used in the query term of the event query to match the key of the action instance with the key that is referred by the confirmation event issued by the operator.

TEMPORAL DEPENDENCIES OF SUB-ACTIONS The incorporation of conditional actions into composite actions introduces a nesting structure into actions. Naturally, the execution of sub-actions that are nested inside a complex action depends on the execution of actions directly contained in the complex action. Eg, in Listing 5.49 the execution of `b` indirectly depends on the action `a`, as the occurrence of the confirmation message is triggered by the execution of `a`.

As a consequence, the temporal dependencies need to reflect the causality between actions contained in the complex action and nested sub-action. Therefore, the times of sub-actions may not be used to specify lower bounds on the initiation of actions contained in the complex action. However, the times of actions contained in a complex action may be used to specify lower bounds on the initiation of sub-actions. Accordingly, with respect to Listing 5.49 the temporal dependencies `init(a) + 30sec <= init(b)` are valid, as `a` determines a lower bound on the nested action `b` whereas `succ(b)` must not be used in a lower bound for, eg, the initiation of `a`.

STATUS QUERIES OF COMPRISING ACTIONS Actions specified in the `THEN` part of a conditional action are only executed when the query in the `IF` part matches. Accordingly, actions in the `THEN` part may not be executed at all and therefore the respective action identifiers can only be used in temporal conditions and event queries that are nested in the `THEN` part. They cannot be used in event queries that determine the status of any comprising action.

For instance, in Listing 5.49 the action identifier `a` can be used in event queries to specify the success or failure of the outermost complex action, whereas the action identifier `b` cannot be used for this purpose.

DEPENDENCIES BEYOND THE EXECUTION STATUS Many physical actions have several meaningful interpretations for their success or failure which depend on the context the action is used in and their concrete purpose within the context.

One example is, for instance, the activation of ventilators in an area. From a technical point of view, the activation is successful when the ventilator sends a positive confirmation of its activation. However, from a more abstract point of view, the ventilator is activated successful when it actually generates the desired airflow. Both interpretations are meaningful for certain purposes but they are not compatible with each other. However, the success of the action can only be specified by either of the two alternatives and thus the programmer needs to pick one of them.

Conditional actions are particularly useful in this kind of situation to deal with the various notions of success required for different situations. They can specify dependencies beyond the success and failure of actions in a generic manner that is specific for the context they are used in and are not limited to rely on the notion of success that has been chosen for the used actions.

5.5 RELATIONS TO XCHANGE^{EQ}

Dura is based on the foundation of the high-level EQL XChange^{EQ}. As a natural consequence, the design and basic ideas of Dura have been substantially influenced by previous work on XChange^{EQ}, such as, [BEo7; BEo8b; Ecko8]. In particular, the rule based declarative language paradigm, the dimensions of complex events, and the clear separation of concerns that is considered for the design of Dura is inspired by XChange^{EQ}. The correspondence of the two languages is also reflected in the syntax of Dura that has been adopted from XChange^{EQ}.

But although XChange^{EQ} and Dura seem remarkably similar on the first sight, substantial changes and extensions have been incorporated into Dura which affect the fundamental basics as well as many details of the language.

REPRESENTATION OF EVENTS XChange^{EQ} integrates the pattern based query approach from Xcerpt to query and construct events and is thus capable to handle event messages providing a payload of XML data. By contrast, Dura applies a simplified version of the pattern based query approach that is (almost) backward compatible to the

one used for Xcerpt and XChange^{EQ} but not capable to query and construct generic XML messages.

In addition Dura represents all properties of events, in particular their reception time and key, directly in their payload whereas XChange^{EQ} considers these properties as meta-data. However, integrating this kind of data into the payload of event is highly desirable as it becomes available to external entities receiving derived events from the EPS and facilitates direct access and versatile application in rules, including the specification of user-defined times for derived events.

TIMES OF EVENTS In XChange^{EQ}, events are by default associated with a single time with fixed semantics that cannot be adapted by programmers.

By contrast, events in Dura can be associated with an arbitrary number of independent and user defined times and time lines, respectively. Accordingly, programmers gain the flexibility to introduce the most appropriate time line or time lines for different kinds of events. Moreover, the reception time of events, which is for convenience implicitly determined with a natural default time interval, can be adapted to the requirements of programmers.

STRUCTURE OF RULES To obtain a meaningful semantics and to facilitate an efficient evaluation XChange^{EQ} rules need to be stratifiable [Ecko8] and furthermore rules need to be free of cycles and recursion. Yet, Eckert [Ecko8] proposes to apply well-founded semantics for XChange^{EQ} to circumvent this restriction.

In Dura rules need to be stratifiable, although well-founded semantics are an option for future research as well. But in contrast to XChange^{EQ}, Dura facilitates recursive rules, if certain conditions on the time of derived events are met.

ELIMINATION OF RELATIVE TIMER EVENTS In XChange^{EQ}, time windows for event accumulation are inevitably specified by means of events, often by means of so-called relative timer events which determine a time window relative to the time of an event. This entails some drawbacks with respect to the expressiveness and ease of use of negations and aggregations in XChange^{EQ}. Using events to specify time windows is in many cases rather cumbersome. In addition the time window for event accumulation is restricted to a closed convex time interval.

By contrast, Dura uses temporal dependencies to specify time windows for negation and grouping instead and thus does not require a notion of relative timer events. In this way, the same mechanism is used to specify both, temporal conditions between events and time intervals for negation and grouping. In addition, time intervals are

no longer restricted to be closed or convex, as the formulas to specify temporal conditions include negations and disjunctions and time points obtained from different events can be used to specify time windows for negations and groupings.

DIMENSIONS OF COMPLEX EVENTS The intrinsic language model of XChange^{EQ} identifies four dimensions of complex events, namely, data extraction, event composition, temporal and other dependencies, event accumulation, and elaborates language operators that are merely devoted to a single dimension in order to obtain a clear separation of concerns.

The general idea of a clear separation of concerns and different dimensions of complex events is acquired from XChange^{EQ} but the applied dimensions are further refined and separated in Dura. In particular event accumulation is further refined into grouping and aggregation and separated from the head of declarative rules to obtain more expressive sub-queries, including queries corresponding to the `HAVING` clause of SQL.

NEGATION AND AGGREGATION Negation and aggregation is in XChange^{EQ} restricted to atomic event queries. Accordingly, negation and aggregation over composite queries cannot be directly specified but must be expressed by means of auxiliary rules instead, which might not always be possible and seems rather cumbersome.

By contrast, Dura facilitates negation and aggregation of composite sub-queries and hence supports multiple cascaded groupings and composite queries with nested negations.

ACCESS TO NON-VOLATILE DATA The applied language model of XChange^{EQ} is purely event based and does not integrate queries for persistent data. Yet, at least queries for static relations seem to be easy to integrated into XChange^{EQ}.

By contrast, Dura directly integrates access to static and dynamic data by means of (static and non-static) stateful objects and queries thereof. In this way, Dura facilitates queries for static database tables and queries for persistent stateful data that can be modified by means of appropriate actions.

STATE BASED PROCESSING Event queries in Dura integrate queries for stateful objects and therefore the derivation of events and the initiation of actions may be contingent upon the current values of stateful objects. Thus, Dura facilitates state based processing which is not directly supported in XChange^{EQ}.

COMPLEX ACTIONS Only Dura provides a notion of complex actions. However, XChange^{EQ} facilitates the execution of remote proce-

dure calls in a very flexible way by means of reactive rules which is not directly supported by Dura but can be easily borrowed from XChange^{EQ} and integrated into Dura, if desired.

6 EMERGENCY MANAGEMENT USE CASE

To further illustrate the language Dura and to strengthen and deepen the understanding of its basic principles, we elaborate a coherent and consecutive example rule set that is related to an Emergency Management (EM) scenario. To this end, we combine aspects of the EMILI use cases into a single scenario that is concerned with the outbreak of a fire in a metro station.

The rules which are elaborated in the following are based on the textual descriptions and flowcharts of the EM scenarios contained in [SBR11a; SBR11b; Vra+10]. In particular the layout and the available equipment of the considered metro station is adopted from these descriptions. Moreover, the EM principles outlined in Section 2.1 are incorporated as a basic principle for the elaborated methods and actions.

6.1 PRELIMINARIES

Before we elaborate the rules that address a fire scenario in a metro station, we briefly introduce the characteristic of the considered station and its representation in Dura.

6.1.1 *Station Layout and Characteristics*

The station is a mid-sized station connecting the metro lines Line A and Line B. The station consists of a mezzanine with small shops and maintenance rooms for technical staff and two central platforms serving as stops for the metro lines Line A and Line B, respectively. The platforms and the mezzanine are each connected by several elevators, escalators, and stairs. Moreover, each platform is connected by single-track railway tunnels to other stations, which are not further represented in the model considered for the use case. The geometric layout of the station is depicted in Figure 6.1.

The different regions of the station are separated into areas and subareas of different size and shape, according to their relevance for EM purposes: The entire station, the tunnels, the mezzanine level and each platform correspond to an area. Moreover, the mezzanine level and the platforms are subdivided into several subareas as it is depicted by means of boxes in Figure 6.2 for the platform of Line A.

The station contains various EM related technical devices including smoke and temperature sensors scattered throughout the entire station, anemometers in the tunnels, portable fire extinguishers on the

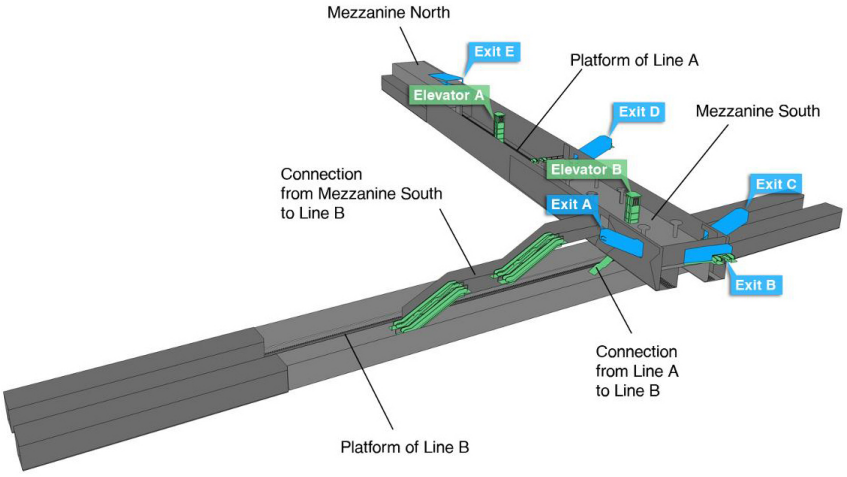


Figure 6.1: Geometric Overview of the Station (from [SBR11a])

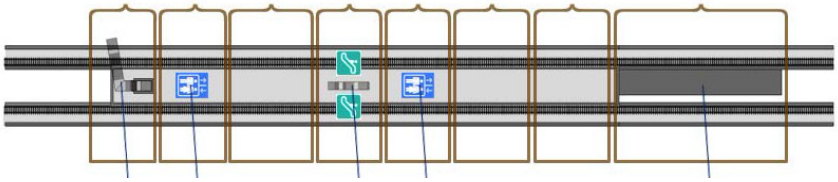


Figure 6.2: Areas of the Platform of Line A (from [SBR11a])

platforms and mezzanine level, several escalators and elevators on each platform, lighting system controlling emergency lights leading the way out of the station, surveillance cameras on the platforms and entrance area, exhaust dampers, air conditioning systems for fresh air supply in the station, and smoke extraction systems with several fans in the tunnels.

Some of those devices, for instance, temperature sensors, regularly communicate status information to the control center whereas others only generate events on particular occasions, eg, fire extinguishers only generate an event when the extinguisher is removed from its retainer. In addition, the actuators are capable to carry out commands that are issued in the control center, eg, elevators can be directed to a certain level and emergency lights can be turned on and off.

6.1.2 *Representation in Dura*

To facilitate the elaboration of rules, the metro station and its equipment are represented in Dura by means of events, stateful objects, and actions. To this end, the information provided by sensors and actuators is directly mapped to events in Dura. Likewise, the commands that can be executed by actuators are represented in Dura by means of external actions. Moreover, topology information and further characteristics of the station are modeled by means of stateful objects.

The representation of the metro station by means of events, stateful objects, and actions is in the majority of cases straight forward. We will thus only address the representation of the most relevant ones in the following.

STATION TOPOLOGY Each area is identified by means of a unique key: the entire station is associated with the key `1`; the mezzanine and the two platforms are associated with the key `10`, `11`, and `12`, respectively; and the subareas of platform `11`, which are depicted in Figure 6.2, are associated from left to right with the keys `110` to `117`.

Listing 6.1: Representation of Topology Information

```
static
station-area{ station{long}, area{long} }

static
area-subarea{ area{long}, subarea{long} }

static
related-areas{ area1{long}, area2{long} }
```

Topology information about the station and its areas is most notably represented by means of the static stateful objects from Listing 6.1. The first two stateful objects `station-area` and `area-subarea` identify the areas of a station and the subareas of an area, respectively. The latter stateful object `related-areas` corresponds to a symmetric relation interrelating all subareas of an area and all areas of a station. For instance, `10`, `11`, and `12` are related areas as well as the areas `110` to `117`.

OPERATION MODE The operation mode is a central notion of EM that summarizes the conditions of the station and its areas. As in the examples from the previous chapter, the operation mode of each area, including the area representing the station and all subareas, is represented by the stateful object `operation-mode` specified in Listing 6.2.

Listing 6.2: Operation Mode Schema

```
operation-mode{ area{long}, mode{int} }
on conflict select { max mode, min id }
```

Thereby, the conflict resolution of the stateful object ensures that in case of concurrent updates to the operation mode of an area, the largest operation mode prevails.

APPLICATION OF CONSTANT DEFINITIONS Dura does not provide a notion of enumerations and therefore constants are used in the following to achieve a similar effect.

For instance, the attribute `mode` of the stateful object `operation-mode` is represented by means of integer values. To facilitate a representation of operation modes that is convenient for humans, the following constant definitions are provided whose values imply a natural order of operation modes, eg, `OPM_NORMAL` < `OPM_EXCEPTIONAL`.

Listing 6.3: Constant Definitions Representing Operation Modes

```
CONST OPM_NORMAL = 0
CONST OPM_EXCEPTIONAL = 1
CONST OPM_EMERGENCY = 2
CONST OPM_EMERGENCY_MAJOR = 3
```

The same scheme is retained in many similar situations in the following. However, most often the actual values assigned to the constants are not relevant for the semantics and thus the definitions are often omitted.

SIMULATION EVENTS The result of external simulators is represented in Dura by means of events. Thereby, the corresponding events are associated with an additional time, the so-called, semantic time. The semantic time of a simulation event indicates when the event is deemed to occur according to the simulation whereas its reception time indicates when the event has been received by the Event Processing System (EPS).

Listing 6.4: Smoke Simulation Schema

```
smoke-simulation{
  sensor{long}, subarea{long}, conc{double},
  semantic-time{ begin{timestamp}, end{timestamp} }
} where { reception-time.end + 3min <= semantic-time.end }
```

Note the additional temporal constraint for `smoke-simulation` events in Listing 6.4. It formalizes the domain knowledge that the smoke simulation looks at least three minutes ahead.

6.2 SITUATION ASSESSMENT

Situation assessment is a crucial part for EM. It includes the identification of emerging incidents, the verification of alarm messages and the elimination of false alarms, the estimation of different characteristics of (emerging) incidents, and the classification of incidents in different categories according to their type and severity.

Situation assessment forms the basis for operators to choose appropriate strategies to counteract upon incidents and furthermore facilitates the automatic execution of immediate reactions. Thereby, situation assessment is a reoccurring task that needs to be continuously repeated to recognize the development of an incident, eg, the rise of its severity, and to include further and more precise information that becomes available in the course of the incident.

6.2.1 *Enrichment of Basic Events*

To begin with, basic events that are received by the EPS are enriched with information relevant for the elaboration of subsequent rules and their representation is homogenized to facilitate more generic rules.

The representation of temperature events from sensors of different manufacturers is homogenized by means of the deductive rule in Listing 6.5, which derives `temp` events from `temp-value` events and from `sensor-msg` events that contain temperature information. In addition, the location of the sensor emitting the temperature value, that is, the subarea the sensor is directly contained in, is determined by means of a query to the static stateful object `sensor`.

Listing 6.5: Schema Mediation of Temperature Readings

```
DETECT
  temp{ sensor{var I}, value{var Tcel}, subarea{var S} }
ON
  and{
    or{
      event e: temp-value{
        val{var Tfah}, sid{var I}
      } let { var Tcel = (Tfah - 32)/1.8 }

      event e: sensor-msg{
        type{TYPE_TEMP_SENSOR}, value{var Tkel}, sensor-id{var I}
      } let { var Tcel = Tkel - 273.15 }
    }

    state s: sensor{ sensor{var I}, subarea{var S} }
  }
END
```

Furthermore, temperature sensors must be identified that suddenly stop to emit events and are thus considered broken. Temperature sensors are regularly emitting temperature readings in intervals between ten and 30 seconds. This domain knowledge is exploited by the rule in Listing 6.6 to derive `temp-sensor-failure` events by inferring the failure of a temperature sensor from missing temperature values. Note that this is not always sound as the respective sensor may be func-

tional whereas the communication infrastructure is malfunctioning but this aspect is not further distinguished here.

Listing 6.6: Sensor Failure Detection

```
DETECT
  temp-sensor-failure{ sensor{var I}, subarea{var S} }
ON
  and{
    event e: temp{ sensor{var I}, subarea{var S} }
    not event f: temp{ sensor{var I} }
  } where { f during from-end(e, 1min) }
END
```

6.2.2 *Incident Categorization*

To enable the reactions that are suited for and tailored to a certain incident, it is mandatory to determine the type of an incident, such as, overcrowding or fire, and to asses whether it is a proper alarm that needs immediate reaction or just a potential incident that may evolve into an alarm but requires further manual clarification.

In addition to `temp` and `smoke` events, temperature and smoke sensors are emitting `pre-alarm` and `full-alarm` events if certain thresholds for the temperature and smoke concentration are exceeded. However, the occurrence of a `pre-alarm` does not necessarily indicate a relevant incident. For instance, `temp-pre-alarms` may be cause by a developing fire, but they can likewise be caused by the air conditioning system of a train exhausting hot air into the direction of a temperature sensor. However, the threshold for full alarms is selected in a way that considerably reduces the risk of false alarms and thus full alarms represent serious incidents that need to be reacted upon.

Based on these and further events, including `temp-sensor-failure` events derived by the rule from above, the rules in Listing 6.7 derive `certain-fire-alarm` and `uncertain-fire-alarm` events. Both rules focus on fire incidents, yet similar rules for other kinds of incidents can be easily elaborated.

6.2.3 *Alarm Verification*

Uncertain alarms need to be further investigated to determine whether they represent false alarms, caused by malfunctioning equipment or inaccurate and indirect measurements, or if they are caused by incidents that will eventually evolve into proper alarms and thus require appropriate reactions. Therefore, uncertain alarms are manually verified either by local staff that is sent to the affected area or by operators in the control center by means of images from surveillance cameras.

Listing 6.7: Incident Categorization

```
DETECT
  uncertain-fire-alarm{ subarea{var S} }
ON
  or{
    event e: fire-extinguisher-removal{ subarea{var S} }
    event e: smoke-pre-alarm{ subarea{var S} }
    event e: temp-pre-alarm{ subarea{var S} }
    event e: temp-sensor-failure{ subarea{var S} }
  }
END

DETECT
  certain-fire-alarm{ subarea{var S} }
ON
  or{
    event e: smoke-full-alarm{ subarea{var S} }
    event e: temp-full-alarm{ subarea{var S} }
  }
END
```

However, false alarms occur on a regular basis and thus the manual verification cannot be issued for every uncertain alarm. In particular in critical situations, the manual verification can only be requested if there are other uncertain alarms detected in the same area which increases the likelihood for the presence of an actual alarm.

The basic idea for the verification of uncertain alarms is as follows. Unless the area affected by the alarm is not already in emergency mode, a warden responsible for the respective area is requested to investigate the situation. However, as guards may be unavailable for various reasons, they need to confirm the verification request by sending a `request-confirmation` event back to the EPS within 20 seconds. If the confirmation holds off, a verification request is sent to the operators in the control center. In any case, the result of the manual verification is communicated back to the EPS by means of an `alarm-verification` event.

The desired behavior is realized by means of the rules in Listing 6.8 to Listing 6.10.

The rule in Listing 6.8 evaluates whether an uncertain alarm occurring in a subarea that is not already in emergency mode needs to be verified and, if so, requests its manual verification by means of the complex action `fire-alarm-verification`. To this end, the static stateful object `related-areas` is queried in addition to `uncertain-fire-alarm` events to determine the number of further uncertain alarms that occurred on the same platform within the last minute. Note that due to the condition `f` *during* `from-end-backward(e, 1min)` an uncertain alarm matching the sub-query `e` cannot simultaneously match the sub-query

Listing 6.8: Detect Required Alarm Verification

```
ON
  and{
    event e: uncertain-fire-alarm{ subarea{var S} }
    state s: operation-mode{ area{var S}, mode{var M} }
      where{ var M < OPM_EMERGENCY }

    event f: uncertain-fire-alarm{ subarea{var S'} }
    state t: related-areas{ area1{var S'}, area2{var S} }
  } where { s valid-at end(e), f during from-end-backward(e, 1min) }
    group by { e } aggregate { var NumAlarms = count(f) }
    where { var NumAlarms >= 2 }
DO
  action a: fire-alarm-verification{ subarea{var S} }
END
```

Listing 6.9: Alarm Verification Action

```
FOR
  action v: fire-alarm-verification{ subarea{var S} }
DO
  compound{
    IF
      state s: responsibility{ subarea{var S}, warden{var W} }
        where { s valid-at init(v) }
    THEN
      compound{
        action a: request-warden-verification{
          warden{var W}, subarea{var S}
        }

        IF
          not event e: request-confirmation{ request-id{id(v)} }
            where { end(e) - init(a) <= 20sec }
        THEN
          action b: request-cc-verification{ subarea{var S} }
      }
  } succeeds on and{
      event f: verification-result{ request-id{id(v)} }
    } where { end(f) - init(v) <= 3min }
END
```

Listing 6.10: Verification Result Processing

```
DETECT
  certain-fire-alarm{ subarea{var S} }
ON
  event e: alarm-verification{ subarea{var S}, confirmation{true} }
END
```

`f` and thus that action is initiated if there are at least three, not two, uncertain alarms in the same area.

The request for alarm verification in Listing 6.9 is based on the external `request-warden-verification` and `request-cc-verification` actions which request the investigation of the respective area from a warden and the operator in the control center, respectively. Initially, a warden responsible for the respective subarea is determined by querying the stateful object `responsibility` and subsequently the warden is requested to verify the alarm in the respective area. Unless the request is confirmed by the warden within 20 seconds, the operator in the control center is requested to verify the alarm. To clearly distinguish several concurrently issued confirmation requests `request-confirmation` and `alarm-verification` events, need to reference the unique key of the `fire-alarm-verification` action, which is contained in the payload of `request-warden-confirmation` and `request-cc-confirmation` actions.

Finally, the rule in Listing 6.10 derives `certain-fire-alarm` events on the reception of `alarm-verification` events that confirm the presence of a certain alarm.

6.2.4 *Fire Size Estimation*

Once a certain fire alarm is detected, the next step is to determine some characteristic values of the fire from the received events which allow the estimation of the fire size and the determination of the initial conditions for fast computable simulations

To this end, mean values of the temperature values and smoke concentration within the last two minutes are computed which allow inferences on the current size of the fire. Naturally, this provides only a rough and momentary estimation that gradually becomes more precise when more information becomes available in the course of the fire. However, to obtain reliable characteristics, larger smoke and temperature values need to be given stronger weights in the mean and therefore the power mean [Bul03] determined by

$$M_p(a_1, \ldots, a_n) = \left(\frac{1}{n} \sum_{k=1}^{n} a_k^p \right)^{\frac{1}{p}}$$

with a sufficiently large p is applied instead of the arithmetic mean.[1]

Based on the fire characteristics, the current size of the fire is estimated by means of two thresholds for the mean temperature and smoke concentration, respectively: If both of the mean values are below their lower threshold the fire size is rated as small fire; if one of the mean values exceeds its lower threshold but both values fall below their upper threshold the fire is rated as significant fire; and in the remaining case the fire is rated as large fire.

1 Note that for $p = 1$ the power mean actually coincides with the arithmetic mean.

Listing 6.11: Determining Fire Characteristics

```
DETECT
  fire-characteristics{
    subarea{var S}, temp{var TempMean}, smoke{var SmokeMean}
  }
ON
  and{
    event e: certain-fire-alarm{ subarea{var S} }
      let { var Window = from-end-backward(e, 1min) }

    event f: temp{ subarea{var S}, value{var Temp} }
      let { var TempP = Temp ** P }

    event g: smoke{ subarea{var S}, conc{var Smoke} }
      let { var SmokeP = Smoke ** P }
  } where { f during Window, g during Window }
    group by { e }
    aggregate {
      var TempSum = sum(TempP), var TempSize = count(f),
      var SmokeSum = sum(SmokeP), var SmokeSize = count(g)
    } let {
      var TempMean = ((1.0/TempSize)*TempSum) ** (1.0/P),
      var SmokeMean = ((1.0/SmokeSize)*SmokeSum) ** (1.0/P)
    }
END
```

Listing 6.12: Fire Size Estimation

```
DETECT
  fire-size{ subarea{var S}, size{FIRE_SIZE_LARGE} }
ON
  event e: fire-characteristics{
    subarea{var S}, temp{var Temp}, smoke{var Smoke}
  } where and{ TEMP_LIMIT_2 < Temp, SMOKE_LIMIT_2 < Smoke }
END

DETECT
  fire-size{ subarea{var S}, size{FIRE_SIZE_SIGNIFICANT} }
ON
  event e: fire-characteristics{
    subarea{var S}, temp{var Temp}, smoke{var Smoke}
  } where and{
      or{ TEMP_LIMIT_1 >= Temp, SMOKE_LIMIT_1 >= Smoke },
      Temp < TEMP_LIMIT_2, Smoke < SMOKE_LIMIT_2
    }
END

DETECT
  fire-size{ subarea{var S}, size{FIRE_SIZE_SMALL} }
ON
  event e: fire-characteristics{
    subarea{var S}, temp{var Temp}, smoke{var Smoke}
  } where or{ Temp < TEMP_LIMIT_1, Smoke < SMOKE_LIMIT_1 }
END
```

The corresponding behavior is implemented by the rules in Listing 6.11 and Listing 6.12, respectively. For these rules the values of the constants are determined as follows: `P = 25.0`, `TEMP_LIMIT_1 = 100`, `TEMP_LIMIT_2 = 400`, `SMOKE_LIMIT_1 = 0.1`, and `SMOKE_LIMIT_2 = 0.2`.

6.3 OPERATION MODE GOVERNANCE

The operation mode of an area is an important property that significantly affects EM related procedures and decisions. Accordingly, it is mandatory to correctly determine the operation mode for each area and to continuously adapt it to the evolving situation. Moreover, the operation mode must not only be correctly identified, but also needs to be appropriately propagated between areas, eg, from a subarea to its comprising area.

6.3.1 *Updating the Operation Mode*

Emergency managers are very conservative with respect to the evacuation of a station. When in doubt, they prefer to be on the save side and evacuate a station once too often rather than to risk the lives of passengers and personnel by mistakenly delaying the evacuation of a station.

As the operation mode substantially influences various EM related decisions, emergency managers are rather conservative in the determination of the operation mode as well, in particular if it is automatically realized by means of rules. Therefore, the operation mode of an area may be automatically increased but it must only be manually decreased. To this end, an auxiliary action is specified in Listing 6.13 that facilitates the adaptation of the operation mode of an area and implicitly discards undesirable automatic decreases.

Listing 6.13: Auxiliary Update Action

```
FOR
  action i: increase-opm{ area{var A}, mode{var Mnew} }
DO
  compound{
    IF
      state s: operation-mode{ area{var A}, mode{var Mold} }
        where { s valid-at init(i), var Mold < var Mnew }
    THEN
      action a: operation-mode$update{
        query{id(s)}, set{ area{var A}, mode{var Mnew} }
      }
  } succeeds on and{
      state t: operation-mode{ area{var A}, mode{var Mactual} }
    } where { t valid-at init(i), var Mactual >= var Mnew }
END
```

In the following, the operation mode is only adapted by means of this auxiliary action. In this way it is ensured that the operation mode is increased but never decreased.

6.3.2 *Detecting Operation Mode Crossovers*

In addition to its general relevance for EM, modeling the operation mode by means of a stateful object facilitates the prevention of alarm avalanches of redundant alarms. For instance, redundant `fire-alarm` events may be caused based on different observations of the same incident by different sensors or from different justifications leading to the derivation of multiple similar events. But although there may be different justifications and observations resulting in redundant `fire-alarm` events, the operation mode of the corresponding area is only raised once.

Listing 6.14: Operation Mode Escalation

```
DETECT
  operation-mode-escalation{ area{var A}, mode{var Mnew} }
ON
  event e: operation-mode$updated{
    old-payload{ area{var A}, mode{var Mold} },
    new-payload{ area{var A}, mode{var Mnew} }
  } where { var Mold < var Mnew }
END
```

This is exploited by the rule in Listing 6.14. Based on detected operation mode updates `operation-mode-escalation` events are derived that are significantly less frequent than `fire-alarm` events. In this way, the rule effectively realizes a filter for alarm events, as operators can focus on the derived escalation events which are easier to recognize and assess than the larger amount of redundant alarm events.

6.3.3 *Identifying and Adapting the Operation Mode*

The operation mode needs to be continuously adapted to the current conditions in the metro station which is identified by means of complex events.

When a certain fire alarm is detected, the operation mode of the corresponding area needs to be immediately adapted to *emergency*. Moreover, if there have already been issues up to five minutes prior to the detection of the certain alarm, that is, the operation mode has already been above *critical* and dropped back to or below *critical*, the operation mode needs to be updated to *emergency major* as this indicates severe problems that require special attention.

In addition, the fire size also affects the operation mode of an area: When the fire size increases to *significant* or *large* the operation mode of the respective area is adapted to *emergency major*.

Listing 6.15: Adapting Operation Modes

```
ON
  event e: certain-fire-alarm{ subarea{var S} }
DO
  compound{
    action a: increase-opm{ area{var S}, mode{OPM_EMERGENCY} }

    IF
      and{
        state s: operation-mode{ area{var S}, mode{var M} }
        state s': operation-mode{ area{var S}, mode{var M'} }
      } where {
        var M > OPM_EXCEPTIONAL, var M' <= OPM_EXCEPTIONAL,
        s valid-during from-end-backward(e, 2min),
        s' valid-at end(e)
      }
    THEN
      action b: increase-opm{ area{var S}, mode{OPM_EMERGENCY_MAJOR} }
  }
END

ON
  event e: fire-size{ subarea{var S}, size{var F} }
    where { var F >= FIRE_SIZE_SIGNIFICANT }
DO
  action a: increase-opm{ area{var S}, mode{OPM_EMERGENCY_MAJOR} }
END
```

This behavior is realized by means of the rules in Listing 6.15. Note that in the first rule, the operation mode is initially updated to *emergency* by means of the action `a`, even if the condition of the conditional action is satisfied and the operation is thus updated to *emergency major* as well. However, both updates are issued simultaneously and therefore the conflict resolution specified for the stateful object operation mode in Listing 6.2 only accepts *emergency major* as the definite operation mode of the respective area.

6.3.4 *Propagating Operation Modes*

After determining the operation mode for subareas, the identified mode needs to be propagated to the operation mode of the station and of the platform or mezzanine containing the affected subarea. More precisely, the operation mode is directly propagated from a subarea to its comprising area. Moreover, the operation mode of an area propagates with reduced severity, that is, with *exceptional* instead of

emergency and with *emergency* instead of *emergency major*, to its related areas and to the station itself.

Listing 6.16: Propagation of Operation Modes

```
ON
  and{
    event e: operation-mode-escalation{ area{var S}, mode{var M} }
    state s: area-subarea{ area{var A}, subarea{var S} }
  }
DO
  action a: increase-opm{ area{var A}, mode{var M} }
END

ON
  and{
    event e: operation-mode-escalation{ area{var A}, mode{var M} }
    state s: station-area{ station{var T}, area{var A} }
  } where { var M >= OPM_EMERGENCY }
    let { var Mprop = var M - 1 }
DO
  compound{
    action a: increase-opm{ area{var T}, mode{var Mprop} }

    IF
      state u: related-areas{ area1{var A}, area2{var A'} }
    THEN
      action b: increase-opm{ area{var A'}, mode{var Mprop} }
  }
END
```

This behavior is realized by the rules in Listing 6.16 by means of the static stateful objects `station-area` and `area-subarea`. As the name suggests, the stateful object `station-area` associates each area with its comprising station and the stateful object `area-subarea` associates each subareas with its comprising area.

6.4 IMMEDIATE REACTIONS

Immediate reactions can be classified into unspecific and specific immediate reactions. Unspecific reactions, such as the release movement of elevators, are executed in a very early phase of the emergency and thus do not require specific details on the fire size and future smoke propagation determined by simulations. In contrast, specific reactions, such as, the determination of the evacuation routes, require some or even detailed knowledge of the emergency specifics. As the characteristics may change in the course of the emergency it may be furthermore required to execute the specific reactions multiple times to adapt their effect to the new situation.

6.4.1 *Elevator Deactivation*

When a fire is detected, elevators in the station are directed to perform a so-called release movement, which moves each elevator car without further stops to a safe floor, open the car's door, and then deactivates the elevator.

Elevators are represented in Dura by means of the static stateful object `elevator` which identifies the floor, more precisely, the subarea, each elevator opens into and a priority that orders the floors of each elevator according to their suitability as a destination for the release movement. Accordingly, to pick a suited destination, each elevator needs to be directed to the floor with the highest priority that is currently not directly affected by the fire and thus currently not in emergency mode.[2]

Listing 6.17: Elevator Release Movement Action

```
FOR
  action r: elevator-release-movement{ elevator{var E} }
DO
  compound{
    IF
      and{
        state s: elevator{ elev{var E}, floor{var F}, prio{var P} }
        not state t: operation-mode{ area{var F}, mode{var M} }
          where { var M > OPM_EXCEPTIONAL, t valid-at init(r) }

        not and{
          state s': elevator{ elev{var E}, floor{var F'}, prio{var P'} }
          not state t': operation-mode{ area{var F'}, mode{var M'} }
            where { var M' > OPM_EXCEPTIONAL, t' valid-at init(r') }

          event r': elevator-release-movement$initiated{ id{id(r)} }
        }
      } where { var P' > var P }
    THEN
      compound{
        action a: move-car-to{ elevator{var E}, dest{var F} }
        action b: deactivate-elevator{ elevator{var E} }
      } where { succ(a) <= init(b) }
  } succeeds on and{
      event e: move-car-to$success{ payload{ ref{id(r)} } }
    } where { e during [init(r), init(r) + 1min] }
END
```

The corresponding behavior is realized by means of the complex action rule in Listing 6.17. To obtain the floor with the highest priority, we use a conditional action to query the floor `F` with priority `P` so that no other floor `F'` has a priority higher than `P`. In addition,

2 Note that this issue is much more relevant for airports, as in case of metro stations the best suited floor is almost always the one located on the surface.

both floors `F` and `F'` are constrained not to be in emergency mode by means of queries to the stateful object `operation-mode`. The respective elevator is then instructed to move its car to the floor `F` and to open its doors by means of the action `move-car-to`. Subsequently, the elevator is temporarily deactivated to prevent its further usage by passengers.

Note that the sub-query for `elevator-release-movement$initiated` is actually required to obtain a correct query. The identifier `r` has positive polarity only in the outermost conjunctive query. As a consequence, omitting the event query and substituting `r` for `r'` in the temporal condition `t' valid-at init(r')` results in a query that is not weakly range restricted. This workaround is somewhat inconvenient but necessary. It may be avoided by a more liberal from of range restriction in future extensions of the language definition, though.

6.4.2 *Announcing Safe Evacuation Routes*

When an emergency situation is detected the station needs to be evacuated as quickly as possible. To this end, passengers need to be guided through the station on safe evacuation routes, that is, routes that do not cross subareas known to be directly affected by the emergency and are thus in emergency mode or subareas known to contain a significant amount of smoke within the next several minutes.

The emergency lights considered in the following are rather simple and just point in a certain direction. Therefore, emergency lights need to be activated so that one light points to the next light, effectively describing a route that leads out of the station. Thus, each emergency light can be thought of as the starting point of one or several (partly overlapping) evacuation routes. Accordingly, evacuation lights are represented in Dura by the static stateful object `em-light` which identifies location of each light and all evacuation routes that start at the respective light when it is turned on. Evacuation routes are in turn represented by the static stateful object `evacuation-route` which identifies all subareas that are part of or crossed by the particular route.

Both static stateful objects are used by the complex action rule in Listing 6.18 to realize the adaptation of the emergency lighting in subareas. The query in the `IF` part matches all lights `L` that are the starting point of at least one evacuation route `R` which satisfies that none of the subareas it crosses is in *emergency* mode or expected to contain more than ten percent smoke within the next three minutes. As an emergency light may be the starting point of several routes, a grouping is applied to execute the action in the `THEN` part only once per light. The complex actions fails unless there is at least one emergency light in the area that is activated and thus points in the direction of a safe evacuation route.

Listing 6.18: Adapt Emergency Lighting

```
FOR
  action l: adapt-em-lighting{ subarea{var A} }
DO
  compound{
    IF
      and{
        state s: em-light{
          subarea{var A}, light{var L}, route{var R}
        }

        not{
          state u: evacuation-route{ route{var R}, segment{var S} }
          state v: operation-mode{ area{var S}, mode{var M} }
        } where { v valid-at init(l), var M > OPM_EXCEPTIONAL }

        not{
          state w: evacuation-route{ route{var R}, segment{var S'} }
          event e: smoke-simulation{ subarea{var S'}, conc{var C} }
        } where { e.semantic-time.end <= init(l)+3min, var C > 0.1 }
      } group by { l, var L }
    THEN
      action a: activate-em-light{ light{var L} }
  } fails on not{
      event e: activate-em-light$succeeded{ payload{ ref{id(l)} } }
    } where { end(f) = init(l) }
END
```

Note how the second negated sub-query queries the result of a simulation that is carried out based on the determined characteristics of the detected fire. Because of the constraints on the semantic time, specified in Listing 6.4, it is actually sufficient for the negation to bound above the `semantic-time` instead of the `reception-time`.

Be aware that if this action is executed multiple times, eg, in case of changing conditions, people may get lost as the emergency light they have been directed to turns out and they hence cannot determine where to go next. To account for this issue, another action is required that deactivates lights in a manner that prevents passengers currently using a certain route from getting lost.

Part III

FORMAL SEMANTICS AND SEMANTIC ANALYSIS

7 SEMANTICS OF COMPLEX ACTIONS

The ambition of this and the following chapters is the elaboration of a static semantic analysis for complex actions that is capable of verifying desirable properties of complex action and that is if furthermore provably sound and complete. But without a formally precise semantics of complex actions the evidence of the soundness and completeness can neither be provided in a formally precise and credible way. Therefore, this chapter is devoted to the elaboration of a formal semantics for complex actions that is sufficiently precise to prove the desired properties of the static analysis for complex actions and is at the same time appropriate for the particularities of external actions, in particular by their contingent duration and effect.

The semantics of complex action and the static analysis presented in this and the following chapters are an extension and revision of work previously presented in [HB13].

7.1 INFORMAL INTRODUCTION

To begin with, the particular properties of complex actions that are substantially affecting their formal semantics are recapitulated. Moreover, basic idea for the formalization of the execution of actions eventually resulting in a formal semantics of complex actions is informally motivated.

7.1.1 *Properties Specific to the Execution of Actions*

Compared to imperative procedures, physical actions have some particular properties that substantially impact the way in which complex actions are executed. Most notably, physical actions are subject to latency, have an uncertain duration and result, and are merely requested by rather than directly executed by the Event Processing System (EPS).

Naturally, these properties also need to be considered for the static analysis of complex actions which is intended to verify properties that are related to the execution of complex action at runtime and hence must be suited to deal with them appropriately.

SUBJECT TO LATENCY Physical actions are merely requested by the EPS and actually executed by external actuators. As a consequence, the distribution of requests generated by the EPS to the corresponding actuator may require a substantial amount of time. Thus, the initia-

tion of actions is subject to latency which can hardly be estimated in advance.

UNCERTAIN DURATION AND RESULT The duration and result of physical actions can hardly be predetermined as physical actions alter and affect physical processes that can hardly be formalized in a sufficiently precise and reliable manner. Consequently, their duration and result, ie, whether they succeed or fail, is contingent until the action is actually executed. Yet, sometimes the duration of actions can be at least bounded below by means of available domain knowledge.

AFFECTED AND OBSERVED TIME-POINTS From a very abstract point of view, executing a complex action resembles the determination of time-points for the corresponding sub-actions in accordance with their temporal dependencies. However, due to the aforementioned properties of physical actions, their time-points cannot be determined in an arbitrary fashion as physical actions are merely requested by the EPS and the execution is subject to runtime effects.

For instance, the initiation time of actions cannot be precisely determined due to the system inherent latency. However, at least a lower bound for the initiation of actions can be affected by the EPS because actions are only initiated *after* they have been requested and thus lower bounds on the request of actions entail lower bounds for their initialization. In contrast, the duration and result of actions cannot be affected at all by the EPS because it cannot determine, eg, whether and when an action is successful as this is subject to the outcome of the corresponding physical processes.

Accordingly, the initiation of actions may be *affected* in the sense that a lower bound for the initiation of actions may be specified whereas their success and failure are merely *observed*.

7.1.2 *Satisfying Temporal Dependencies*

During runtime, actions need to be executed according to their temporal dependencies. To this end, lower bounds for the initiation need to be established in a way such that the resulting values for the initiation, success, and failure meet the temporal dependencies.

However, due to the incomplete knowledge on actions and the resulting uncertainty for the exact values of their initiation, success, and failure the required bounds remain undetermined until runtime where they become gradually available when the respective actions are executed.

EXAMPLE The temporal dependencies `succ(a) + 5 <= init(b)` specify the lower bound `succ(a) + 5` for the initiation of `b`. However, the

exact value of `succ(a)` remains unknown until the action `a` has actually succeeded.

In order to satisfy the temporal dependencies in presence of latency and unknown durations of actions the following strategy is applied. The action `a` is requested, as there are no dependencies that constrain its initiation. If the action is successful, the value of `succ(a)` is eventually determined and thus the request of the action `b` can be deferred until the time `succ(a) + 5` has lapsed. Note that at this time the action `b` is only requested and because of latency the time `init(b)` of its actual initiation is even later. In this way, the temporal dependencies `succ(a) + 5 <= init(b)` are guaranteed to hold.

Naturally, this strategy can be applied to all formulas that refer to the initiation of actions on the right hand side of an inequation expressing a temporal dependency. Formulas of this kind bear a special meaning and therefore they are also called *lower bound formulas*.

RESTRICTION TO LOWER BOUND FORMULAS By design, complex actions in Dura only support temporal dependencies that are composed from lower bound formulas. Although it seems desirable to consider more generic formulas for the specification of the execution of actions, this restriction is not specific to Dura but applies to all systems that establish a clear separation between the specification of the execution and the specification of properties that should be satisfied when the action is executed.

The time of the success and failure of actions cannot be determined by the EPS and thus temporal dependencies that refer to those times on the right hand side of an inequation can only be observed at runtime. Therefore, these kind of dependencies do not necessarily hold at runtime, unless the EPS applies a dynamic execution strategy that effectively enforces temporal dependencies that are not explicitly specified by the programmer. However, to obtain a clear separation of dimensions, temporal dependencies are intended to explicitly specify the execution strategy of actions. They should not entail additional implicit dependencies which can easily be overlooked by programmers and are merely established by the scheduling algorithm to satisfies the explicitly specified temporal dependencies.

However, if desired, generic formulas can be specified as assertions in the `hence` part of complex actions whereby it is verified whether they are actually satisfied at runtime when the complex action is executed according to its temporal dependencies. In this way, the full flexibility of generic formulas is preserved and at the same time programmers are required to explicitly and unambiguously specify all temporal dependencies that affect the execution of the action.

EVENT DRIVEN FEEDBACK LOOP In general, the execution of complex actions resembles an event driven feedback loop which roughly

relates to the Observe-Orient-Decide-Act (OODA) model [Gra05] that describes different processes of an agent competitively interacting with other agents in the environment.

At some point the execution of a complex action is triggered by an event. This causes the execution of certain sub-actions which can be sensed by means of events and eventually leads to the detection of their success or failure. These events either satisfy the preconditions of conditional actions or determine the values of lower bounds for the initiation of further actions. Hence, further actions are eventually requested when their corresponding bounds are exceeded. This, in turn, leads to the detection of further events which determines values of further lower bounds and thus a new iteration of the feedback loop begins.

7.1.3 *Basic Ideas and Approach*

To obtain a semantics of actions that is suitable for a static analysis the execution of complex actions needs to be represented in a formal and precise manner. But although the execution strategy that is applied by the EPS described above can be precisely formalized, the uncertain runtime effects of physical actions substantially complicate a precise formalization of their behavior. If the runtime effects are known, eg, if they have been measured during runtime, the execution of the action can be precisely reproduced. However, due to the physical nature of actions this information often cannot be determined at compile time and thus is not available for the semantic analysis.

As a consequence, the semantics of complex actions needs to be formalized by abstracting away the unknown runtime effects. To this end, the event driven feedback loop described above is formalized by means of a fixpoint computation that iteratively determines the times of the executed actions as a function of so-called scenarios which describes the occurring runtime effects. Subsequently, the desired properties are verified for all possible scenarios. If the verification fails for a single scenario it fails for the entire complex action. This guarantees that all runtime effects that may actually occur at run time are covered at compile time.

Moreover, if specific domain knowledge is available, this information can be incorporated into the analysis in order to exclude certain scenarios that do not correspond to the delays as they are known to appear at runtime.

7.2 FORMALIZATION OF COMPLEX ACTIONS

For the sake of simplicity, complex actions are formalized in a more concise manner that omits the syntactic constructs of Dura and contains only information that is crucial for the static analysis of com-

plex actions. As names and parameters are not relevant for the intended analysis, they are omitted from the formal representation and only temporal formulas that correspond to the temporal dependencies, domain knowledge, and temporal assertions of complex actions are represented in the formalization.

7.2.1 *Formal Representation of Complex Actions*

For convenience, the sub-actions of a complex action are referred to by means of the corresponding action identifier which is more concise than their actual name. The time-points of sub-actions, which are particularly important for the static analysis, are represented by means of variables associated with the corresponding action identifier.

For instance, the time-points referring to the initiation, success, and failure of the following action

```
action a: open-fire-dampers{ area{var A} }
```

are represented by the variables a_{init}, a_{succ}, and a_{fail}, respectively.

DEFINITION 7.1. The set of *identifiers* and *variables* is denoted $\mathcal{I}$ and $\mathcal{V}$, respectively. Variables are furthermore classified into *affected* variables $\mathcal{V}_a$ and *observed* variables $\mathcal{V}_o$ with

$$\mathcal{V}_a = \bigcup_{f \in \mathcal{I}} \{f_{init}\} \quad \text{and} \quad \mathcal{V}_o = \bigcup_{f \in \mathcal{I}} \{f_{succ}, f_{fail}\}.$$

The separation of affected and observed variables reflects the different extent in which the EPS can influence the values that are determined for those time-points at runtime. The initiation of actions can be deferred by the system and thus the system can determine lower bounds for the value of affected variables whereas the values of observed variables can merely be observed by the EPS.

Note that the given notions resemble activated and received time-points from [VF99].

DEFINITION 7.2. *Temporal formulas* are inductively defined by

1. $\top$ and $\bot$ are temporal formulas
2. each *atomic formula* $u + d \leqslant v$ with $u, v \in \mathcal{V}$ and $d \in \mathbb{Q}$ is a temporal formula
3. if F and G are temporal formulas then $F \wedge G$ is a temporal formula

For convenience, temporal formulas of the form $u + -d \leqslant u'$ and $v + 0 \leqslant v'$ are represented more concisely by $u - d \leqslant u'$ and $v \leqslant v'$, respectively.

DEFINITION 7.3. A *complex action* is a triple $C = (W, D, H)$ of temporal dependencies W, domain knowledge D, and temporal assertions H represented by temporal formulas.

A complex action in Dura is formally represented by a complex action $C = (W, D, H)$ by mapping its temporal dependencies to the temporal formula W, its domain knowledge to the temporal formula D, and its temporal assertions to the temporal formula H.

DEFINITION 7.4. The *variables* occurring in a temporal formula F are denoted $\mathrm{var}(F)$.

Variables of a temporal formula commonly correspond to the initiation, success, and failure times of the corresponding complex action and its sub-actions.

EXAMPLE 7.1. The formal representation of complex actions omits all information that is not crucial for the static analysis from the specification of the action in Dura.

Listing 7.1: Specification of a Complex Action

```
compound{
  action a: open-fire-dampers{ area{var A} },
  action b: activate-ventilators{ area{var A} }
} where { succ(a) + 5sec <= init(b) }
  hence { init(a) + 20sec <= succ(b), init(a) + 1min <= fail(b) }
```

Accordingly, the complex action from Listing 7.1 is formally represented by the triple $C = (W, D, H)$ with

$$\begin{aligned} W &= a_{\text{succ}} + 5 \leqslant b_{\text{init}} \\ D &= \top \\ H &= a_{\text{init}} + 20 \leqslant b_{\text{succ}} \wedge a_{\text{init}} + 60 \leqslant b_{\text{fail}} \end{aligned}$$

whereby a_{init} and b_{init} are affected variables and a_{succ}, b_{succ}, and b_{fail} are observed variables.

7.2.2 *Formalization of Domain Knowledge*

The execution of complex actions follows the algorithm that is provided by the EPS and obeys some very natural and basic properties. Sub-actions are not initiated before the initiation of the complex action. Moreover, actions succeed or fail only after they have been initiated. These implicit assumptions are formalized by the following definitions.

DEFINITION 7.5. For a temporal formula W its *lower bound formulas* are denoted

$$W_{\mathcal{L}} = \bigwedge_{\substack{v \in \mathcal{V}_a \\ u + d \leqslant v \text{ is sub-formula of } W}} u + d \leqslant v$$

As discussed in Section 7.1.2, lower bound formulas are closely related to the execution of actions by the EPS. Ie, for a complex action $C = (W, D, H)$ only the lower bound formula $W_{\mathcal{L}}$ is actually considered by the EPS for the execution of C.

DEFINITION 7.6. The *axiomatic closure* of the temporal dependencies W is denoted

$$W_{\mathcal{A}} = W \wedge \bigwedge_{f_{init} \in var(W)} \left(\perp_{init} \leqslant f_{init} \wedge f_{init} \leqslant f_{succ} \wedge f_{init} \leqslant f_{fail} \right)$$

whereby the special variable $\perp_{init} \in \mathcal{V}_o$ refers to the initiation of the complex action.

The axiomatic closure simply formalizes that sub-actions are only initiated after the corresponding complex action has been initiated and that actions are only successful or fail after they have been initiated.

In addition to the very generic domain knowledge denoted by the axiomatic closure there may be more specific information available that applies only for certain actions. This kind of information includes, eg, information on the system latency and bounds on the duration of certain actions. Domain knowledge is specified in a way that resembles the specification of temporal dependencies and assertions by means of inequalities in the schema of actions.

Accordingly, analogous to the formalization of temporal dependencies and assertions, domain knowledge is formally represented by means of temporal formulas.

EXAMPLE 7.2. If it is known that the opening of the fire dampers always takes longer than 30 seconds, this information is incorporated into the schema of the `open-fire-dampers` action and represented by means of the following formula

$$D \;=\; a_{init} + 30 \leqslant a_{succ} \wedge a_{init} + 30 \leqslant a_{fail}.$$

Moreover, the subsequent formula specifies that the latency of the system exceeds 100 milliseconds

$$D' \;=\; \perp_{init} + 0.1 \leqslant a_{init} \wedge \perp_{init} + 0.1 \leqslant b_{init} \wedge a_{succ} + 0.1 \leqslant b_{init}$$

The available domain knowledge is eventually considered by the semantic analysis introduced in Chapter 8 to verify the validity of assertions at runtime

7.2.3 *Formalization of Conditional Actions*

Complex actions containing conditional actions are also represented by means of a triple $C = (W, D, H)$. Thereby, temporal conditions are

simply represented as discussed above. Additional information inferable from the event query in the `IF` part is currently not considered by the analysis algorithm, although it can be incorporated into the domain knowledge of complex actions.

For a discussion on the implications of the imprecise formalization of complex actions refer to Section 8.3.3.

Listing 7.2: A Conditional Action

```
compound{
  action a: ...

  IF state s: is-empty{ area{var A} }
      where{ s valid-at succ(a) }
  THEN action b: ...

} where { succ(a) + 10sec <= init(b) }
```

EXAMPLE 7.3. Accordingly, the complex action from Listing 7.2 is formalized by $C = (W, \top, \top)$ with

$$W \;=\; a_{\text{succ}} + 10 \leqslant b_{\text{init}}$$

In this particular example, C precisely formalizes the complex action because the query in the `IF` part does not affect the execution of the action. However, if the temporal conditions of the event query are adapted to, eg, `s valid-at succ(a) + 23sec`, the action `b` will not be executed until 23 seconds after the success of `a` which corresponds to the additional domain knowledge

$$D \;=\; a_{\text{succ}} + 23 \leqslant b_{\text{init}}$$

which is not present in the primal formalization of C.

7.3 FIXPOINT THEORY

In order to statically verify properties of complex actions, their behavior during runtime needs to be characterized in a formally precise manner appropriate for formal reasoning.

To this end, we develop a notion of runtime traces that formalizes the execution of complex actions with respect to their temporal dependencies and the execution strategy that is applied by the EPS.

7.3.1 *Preliminaries*

DEFINITION 7.7. $\mathbb{P} = \mathbb{Q}^+ \cup \{\infty\}$ and $\mathbb{D} = \mathbb{Q}^+ \cup \{\infty\}$ denote the set of *time-points* and the set of *durations*, respectively. Time-points and durations can be added up in a canonical manner

$$+ : \mathbb{P} \times \mathbb{D} \to \mathbb{P}, \quad (p, d) \mapsto p + d.$$

Thereby, ∞ denotes an arbitrarily large time-point or duration with

$$\forall q \in \mathbb{Q} : \; q < \infty \;\text{ and }\; \infty + q = q + \infty = \infty$$

Time-points and durations are distinguished to emphasize their different semantics. Durations will be used to describe possible delays during runtime whereas time-points will refer to the absolute times, for instance, at which time an action succeeded.

DEFINITION 7.8. A *variable assignment* maps variables to values in $\mathbb{D}$ or $\mathbb{P}$. In the following, variable assignments are denoted by sets of variable value pairs as it is known from substitutions used in model theory [ABW88].

Thereby, $\mathrm{im}(\sigma)$ denotes the *image* of a variable assignment σ, that is, $\mathrm{im}(\sigma)$ corresponds to the set of all values that are actually substituted for variables by σ.

Variable assignments are intended to describe the execution of actions during runtime. To this end, they map variables corresponding to the initiation, success and failure of sub-actions to actual time-points. Thereby the value ∞ indicates that an action has not been initiated, did not succeed, or did not fail.

Variable assignments can be naturally applied to formulas and composite expressions in a canonical manner. For $v_i \in \mathcal{V}$ and $d_i \in \mathbb{Q}$, temporal formulas F and G, and a variable assignment σ follows

$$\begin{aligned}
&\sigma(F \wedge G) = \sigma(F) \wedge \sigma(G) \\
&\sigma(v_1 + d_1) = \sigma(v_1) + d_1 \\
&\sigma\big(\{v_1 + d_1, \ldots, v_k + d_k\}\big) = \big\{\sigma(v_1 + d_1), \ldots, \sigma(v_k + d_k)\big\}
\end{aligned}$$

EXAMPLE 7.4. The complex action $C = (a_{\mathrm{succ}} + 5 \leqslant b_{\mathrm{init}}, \top, \top)$ derived from Listing 7.1 may be, for instance, initiated at time 2 followed by the initialization of a at time 5 and its success four seconds later. Eventually b may be initiated at time 16 and fails at time 67.

This specific execution of the action is formalized by means of the following variable assignment

$$\varsigma = \big\{2/\bot_{\mathrm{init}}, 5/a_{\mathrm{init}}, 9/a_{\mathrm{succ}}, \infty/a_{\mathrm{fail}}, 16/b_{\mathrm{init}}, \infty/b_{\mathrm{succ}}, 67/b_{\mathrm{fail}}\big\}$$

which maps variables of C to time-points whereby $\varsigma(a_{\mathrm{fail}}) = \infty$ and $\varsigma(b_{\mathrm{succ}}) = \infty$ specify that a never fails and b never succeeds. Moreover, ς satisfies the temporal conditions W and the assertions H, that is, the variable free formulas

$$\begin{aligned}
\varsigma(W) \;&=\; 9 + 5 \leqslant 16 \\
\varsigma(H) \;&=\; 5 + 20 \leqslant \infty \;\wedge\; 5 + 60 \leqslant 67
\end{aligned}$$

hold under the axioms of $\mathbb{P}$ from Definition 7.7. In the following, this circumstance is also denoted by $\varsigma \models W$ and $\varsigma \models H$.

Obviously, not every arbitrary variable assignment describes how an action is actually executed during runtime. An arbitrary variable assignment may, for instance, indicate that a sub-action succeeds before it started or may not satisfy the temporal dependencies of actions. Thus, to obtain a valid representation for the execution of actions, those invalid variable assignments need to be excluded by incorporating the temporal dependencies and the delays that occur at runtime.

7.3.2 *Fixpoint Iteration*

The time-points of actions are not determined arbitrarily but are the result of the execution strategy that is applied by the EPS and runtime effects that determine, eg, how long it takes to execute a physical action and whether it is successful or not. However, in general, these properties are unknown prior to the actual execution of the action. It cannot be known in advance how long it will take to execute an external action and whether the execution will be successful or not.

To obtain a generic and valuable semantics of actions, these properties need to be considered in a suited way. To this end, the runtime effects are abstracted away and the elaborated semantics of complex actions is a function of so called scenarios.

DEFINITION 7.9. A variable assignment Δ is called *scenario* for a formula W if it maps all variables of W to corresponding delays rather than time-points, that is, if Δ has the signature $\mathrm{var}(W) \to \mathbb{D}$.

For the sake of readability, $\Delta(v)$ is also denoted Δ_v.

Scenarios are meant to abstract away runtime effects that can hardly be specified formally or that are a priori unknown. Each scenario describes one particular series of developments for the different outcomes of sub-actions and time delays that can potentially occur during runtime.

EXAMPLE 7.5. The scenario

$$\delta = \{2/\bot_{\mathrm{init}}, 3/a_{\mathrm{init}}, 4/a_{\mathrm{succ}}, \infty/a_{\mathrm{fail}}, 2/b_{\mathrm{init}}, \infty/b_{\mathrm{succ}}, 51/b_{\mathrm{succ}}\}$$

describes, eg, that the action a is initiated 3 second after the complex action was initiated and succeeds 4 seconds after it has been initiated but never fails.

Note how the delays described by this scenario correspond to the delays that occur during the execution described in Example 7.4 by means of ς. However, be aware that delays for the initiation of actions are specified relative to the earliest possible time an action could have been initiated. Eg, because of the dependency $a_{\mathrm{succ}} + 5 \leqslant b_{\mathrm{init}}$ the earliest time for the initiation of b is 14, however b is actually initiated at time 16, thus the delay $\delta(b_{\mathrm{init}})$ for the initiation of b corresponds to 2 and *not* to 7.

A scenario characterizes all runtime specific properties that influence the execution of physical actions. Therefore, if a particular scenario is known, all relevant time-points related to the execution of a complex action can be determined in a deterministic manner by imitating the execution strategy that is applied by the EPS.

To this end, the stepwise determination of time-points as it is informally described in Section 7.1.2 is formalized by a stepwise computation of variable assignments. Thereby, the runtime specific properties that affect the time-points are obtained from a predetermined scenario.

DEFINITION 7.10. The *preconditions* for the determination of a variable $v \in \mathrm{var}(W)$ of the temporal dependencies W are denoted

$$\mathrm{pre}_W(v) = \{u + d \mid u + d \leqslant v \text{ is sub-formula of } W\}$$

Informally, the preconditions of a variable f_{init} describe the earliest possible time for the request of the action f with respect to the times the executions of f depends on.

EXAMPLE 7.6. Given the axiomatic closure

$$\begin{aligned} W_{\mathcal{A}} \;=\; W \wedge \big(\perp_{\mathrm{init}} \leqslant a_{\mathrm{init}} \wedge a_{\mathrm{init}} \leqslant a_{\mathrm{succ}} \wedge a_{\mathrm{init}} \leqslant a_{\mathrm{fail}}\big) \;\wedge \\ \big(\perp_{\mathrm{init}} \leqslant b_{\mathrm{init}} \wedge b_{\mathrm{init}} \leqslant b_{\mathrm{succ}} \wedge b_{\mathrm{init}} \leqslant b_{\mathrm{fail}}\big) \end{aligned}$$

of the temporal dependencies $W = a_{\mathrm{succ}} + 5 \leqslant b_{\mathrm{init}}$ from Line 7.1, the preconditions of b_{init} correspond to

$$\mathrm{pre}_{W_{\mathcal{A}}}(b_{\mathrm{init}}) = \{\perp_{\mathrm{init}} + 0,\ a_{\mathrm{succ}} + 5\}$$

Hence, in this particular case, $\mathrm{pre}_{W_{\mathcal{A}}}(b_{\mathrm{init}})$ indicates that b is requested only if the complex action has already been initiated and if furthermore a has been successful at least 5 seconds ago.

The actual determination of values of further variables is accomplished by means of the operator T. The operator formalizes one step of the informally introduced feedback loop.

DEFINITION 7.11. For a temporal formula W and a runtime scenario Δ the operator $\mathsf{T}_{\Delta,W}$ maps variable assignments to variable assignments with

$$\begin{aligned} \mathsf{T}_{\Delta,W}(\sigma) = \Big\{ \Big(\max\{\sigma(\mathrm{pre}_W(v))\} + \Delta_v \Big) \Big/ v \;\Big|\; \\ \mathrm{var}\big(\mathrm{pre}_W(v)\big) \subseteq \mathrm{dom}(\sigma) \Big\} \end{aligned}$$

whereby $\max \emptyset = 0 \in \mathbb{P}$.

Note how the operator T makes a transition from merely syntactic formulas on the right to actual time-points that are associated with a variable ν on the left.

EXAMPLE 7.7. Recall the complex action from Line 7.1 with the temporal dependencies $W = a_{succ} + 5 \leqslant b_{init}$, which specify that b is initiated at least 5 seconds after the success of a.

Moreover, consider the variable assignment

$$\sigma = \{2/\bot_{init}, 5/a_{init}, 9/a_{succ}\}$$

which describes an ongoing execution of the complex action and thus contains a set of already observed time points. According to σ, the action a has been successful at time 9 whereas b has not been initiated yet. Therefore, when the action is executed by the EPS the next step would be to request b so that b is initiated at time 14 or later.

Note how the following application of $\mathsf{T}_{\delta,\mathcal{W}_A}$ to σ corresponds to the behavior of the EPS.

$$\begin{aligned}\mathsf{T}_{\delta,\mathcal{W}_A}(\sigma) &= \sigma \cup \{(\max\{\sigma(\bot_{init}+0,\ a_{succ}+5)\} + \delta_{b_{init}})/b_{init}\} \\ &= \sigma \cup \{(\max\{2,14\}+2)/b_{init}\} \\ &= \sigma \cup \{16/b_{init}\}\end{aligned}$$

In this case $\max\{\sigma(\bot_{init}+0,\ a_{succ}+5)\} = 14$ determines the earliest possible time for the initialization of b by means of the already observed time-points obtained from σ. Moreover, the latency $\delta_{b_{init}}$ is incorporated to determine when b is actually initiated according to the scenario δ from Example 7.5. Eventually, the computation yields a new variable assignment which incorporates the determined value 16 for the initiation of b in addition to all previously determined values.

Accordingly, the operator T adds values for affected variables of actions to the variable assignment σ whose initiation would have been requested be the EPS in the situation described by σ. To obtain a complete variable assignment that contains values for all variables of $\mathcal{W}_A$, the operator is applied multiple times to incrementally determine all missing values.

Note that by determining $\max \emptyset = 0$ the request time of complex actions is artificially set to 0. In this way, the operator T describes the execution of sub-action relative to the request of the complex action.

DEFINITION 7.12. The *powers* of T are inductively defined by

$$\begin{aligned}\mathsf{T} \uparrow 0 &= \emptyset \\ \mathsf{T} \uparrow n+1 &= \mathsf{T}(\mathsf{T} \uparrow n)\end{aligned}$$

and the *least fixpoint* of $\mathsf{T} \uparrow$ is denoted $\mathbf{T} = \mathrm{lfp}(\mathsf{T} \uparrow)$.

Remark. For the sake of readability the indices of the operator T and of related notions may be omitted. For instance, T, $\mathbf{T}$ and $\mathsf{T}\uparrow$ will be often used as abbreviations for $\mathsf{T}_{\Delta,\mathcal{W}_\mathcal{A}}$, $\mathbf{T}_{\Delta,\mathcal{W}_\mathcal{A}}$ and $\mathsf{T}_{\Delta,\mathcal{W}_\mathcal{A}}\uparrow$ in the following.

The fixpoint $\mathbf{T}$ basically describes, based on one particular scenario, how the action will be executed by the system if the given delays are actually observed at runtime. Note that the operator T is monotonic [ABW88] and that the fixpoint $\mathbf{T}$ actually exists for any complex action C. Moreover, the fixpoint is unique and is reached after a finite number of iterations.[1]

EXAMPLE 7.8. A complete iteration of $\mathsf{T}_{\Delta,\mathcal{W}_\mathcal{A}}$ for $W = a_{succ} + 5 \leqslant b_{init}$ is given in Figure 7.1.

$$
\begin{aligned}
&\mathsf{T}\uparrow 0 = \emptyset\\
&\mathsf{T}\uparrow 1 = \{\Delta_{\perp_{init}}/\perp_{init}\}\\
&\mathsf{T}\uparrow 2 = \mathsf{T}\uparrow 1 \cup \{\Delta_{\perp_{init}} + \Delta_{a_{init}}/a_{init}\}\\
&\mathsf{T}\uparrow 3 = \mathsf{T}\uparrow 2 \cup \{\Delta_{\perp_{init}} + \Delta_{a_{init}} + \Delta_{a_{succ}}/a_{succ}\}\\
&\qquad\qquad \cup \{\Delta_{\perp_{init}} + \Delta_{a_{init}} + \Delta_{a_{fail}}/a_{fail}\}\\
&\mathsf{T}\uparrow 4 = \mathsf{T}\uparrow 3 \cup \{\max\{\Delta_{\perp_{init}},\ \Delta_{\perp_{init}} + \Delta_{a_{init}} + \Delta_{a_{succ}} + 5\} + \Delta_{b_{init}}/b_{init}\}\\
&\qquad = \mathsf{T}\uparrow 3 \cup \{\Delta_{\perp_{init}} + \Delta_{a_{init}} + \Delta_{a_{succ}} + 5 + \Delta_{b_{init}}/b_{init}\}\\
&\mathsf{T}\uparrow 5 = \mathsf{T}\uparrow 4 \cup \{\Delta_{\perp_{init}} + \Delta_{a_{init}} + \Delta_{a_{succ}} + 5 + \Delta_{b_{init}} + \Delta_{b_{succ}}/b_{succ}\}\\
&\qquad\qquad \cup \{\Delta_{\perp_{init}} + \Delta_{a_{init}} + \Delta_{a_{succ}} + 5 + \Delta_{b_{init}} + \Delta_{b_{fail}}/b_{fail}\}\\
&\mathsf{T}\uparrow 6 = \mathsf{T}\uparrow 5 = \mathbf{T}
\end{aligned}
$$

Figure 7.1: A Complete Fixpoint Iteration

Note that not only the initiation of actions, but also the time-point of their success and failure are determined by T according to the scenario Δ. With the concrete scenario δ from Example 7.5 and with $\mathbf{T} = \mathrm{lfp}(\mathsf{T}_{\delta,\mathcal{W}_\mathcal{A}}\uparrow)$ follows

$$\mathbf{T}(\perp_{init}) = 2, \quad \mathbf{T}(a_{init}) = 5, \quad \mathbf{T}(a_{succ}) = 9$$

which means that the complex action is initiated at time 2 and the sub-action a is initiated at time 5 and succeeds at time 9.

However, under certain conditions the fixpoint $\mathbf{T}$ may not contain values for all variables of the complex action as the next example demonstrates.

EXAMPLE 7.9. For $W = a_{succ} + 5 \leqslant b_{init} \wedge b_{succ} + 3 \leqslant a_{init}$ the EPS will neither initiate a nor b at runtime as the initiation of a (indirectly) depends on the initiation of b and vice versa. Therefore it is desirable

1 For a formal proof of these and further properties of T refer to Section 9.2.

that the fixpoint $\mathbf{T}$ assigns ∞ to the variables a_{init} and b_{init}. However, the fixpoint $\mathbf{T}$ contains neither of the two variables instead.

$$T \uparrow 0 = \emptyset$$
$$T \uparrow 1 = \left\{\Delta_{\perp_{\text{init}}} / \perp_{\text{init}}\right\}$$
$$T \uparrow 2 = T \uparrow 1 = \mathbf{T}$$

This observation finally leads to the desired notion of runtime traces which describe the execution of actions according to their temporal dependencies and assign values in $\mathbb{P}$ to all variables of an action.

7.3.3 *Runtime Traces*

Runtime traces formalize the execution of complex actions. On an abstract level, runtime traces correspond to particular solutions of temporal formulas. More precisely, for a complex action $C = (W, D, H)$, runtime traces are those solutions of W that correspond to the possible instances of the complex action at runtime.

Basically, runtime traces are an extensions of variable assignments obtained from a fixpoint iteration. Thereby, variables of the complex action that are not contained in the fixpoint are mapped to infinity so as to to obtain a complete representation of the execution of the action.

DEFINITION 7.13. For temporal dependencies W the variable assignment τ is called *runtime trace* of W iff there is a scenario Δ such that with $\mathbf{T} = \text{lfp}(T_{\Delta, W_A} \uparrow)$ holds

$$\tau|_{\text{dom}(\mathbf{T})} = \mathbf{T}$$
$$\forall v \notin \text{dom}(\mathbf{T}) : \ \tau(v) = \infty$$

Mind the difference between scenarios and runtime traces. A scenario describes the time delays that occur at runtime in a real world system and specify which actions are successful and which fail. In contrast, a runtime trace describes, based on one particular runtime scenario, how a composite action with certain execution constraints is actually executed at runtime by the EPS. It specifies the exact time points for the initiation, success and failure of all of its sub-action relative to the request of the complex action.

DEFINITION 7.14. For temporal dependencies W and a scenario Δ the corresponding runtime trace is denoted τ_{Δ, W_A}.

Note that a runtime trace is indeed completely determined by the temporal dependencies W and a scenario Δ. Accordingly, the given notation is a convenient abbreviation for the characterization of runtime traces from Definition 7.13.

In the following, the convention is adopted that variable assignments are denoted σ whereas runtime traces are denoted τ to emphasize the difference between variable assignments and runtime traces on a syntactical level.

DEFINITION 7.15. A runtime trace τ is a *definite runtime trace* iff

$$\forall f_{init} \in \mathrm{dom}(\tau) : \ \tau(f_{succ}) = \infty \vee \tau(f_{fail}) = \infty$$

The notion of definite runtime traces is particularly relevant for Emergency Management (EM) as definite runtime traces satisfy the natural assumption that no action both succeeds after finite time and fails after finite time. However, for other application domains the restriction to definite runtime traces might be dropped which entails a slightly different and less complicated semantic analysis.

DEFINITION 7.16. A variable assignment τ is called *runtime trace of* $C = (W, D, H)$ iff τ is a runtime trace of $W_{\mathcal{L}}$ and $\tau \models D$.

Runtime traces of C are very specific solutions of the formula corresponding to temporal conditions of C. Each runtime trace corresponds to one possible way in which the action can be actually executed at runtime and conversely for each way the action is executed there is a corresponding runtime trace.

Note that, by design, the EPS only determines values for affected variables and hence ignores all atomic formulas with observable variables on the right hand side of the inequation. Therefore, runtime traces of $C = (W, D, H)$ are defined as being runtime traces of $W_{\mathcal{L}}$ in order to correspond to the way actions are actually executed at runtime.

7.3.4 *Recapitulation of Notions*

The execution of complex actions depends on its temporal dependencies and on contingent runtime effects which are formalized by means of so-called scenarios. Each scenario gives rise to a particular instance of the complex actions which differs from other instances in the exact time points for the initiation, success, and failure of its sub-actions.

As it cannot be predetermined which scenario will actually take place at runtime, the semantics of a complex action is formalized as a function of scenarios. As a consequence, there is no single semantics of a complex action but instead a family of different possible semantics, each of which is caused by a different scenario and determined by means of a fixpoint iteration which formalizes the execution algorithm applied by the EPS. Thereby, each possible semantics is called runtime trace and naturally satisfies the temporal dependencies of the corresponding actions. Accordingly, runtime traces correspond to specific solutions of the temporal dependencies of a complex actions.

However, mind the difference between runtime traces and solutions to the temporal dependencies of complex actions. Only runtime traces describe the execution of complex actions in an appropriate manner. And although runtime traces necessarily satisfy the temporal dependencies of the corresponding, not every solution to the temporal dependencies corresponds to a runtime trace. In some rather extreme cases, there even exists infinitely many solutions to the temporal dependencies of a complex action, but none of them corresponds to a (definite) runtime trace (for details refer to Section 8.1.1).

Finally, definite runtime traces of a complex action are runtime traces that additionally satisfy that each sub-action either succeeds or fails, but not both. As this is a natural and desirable assumption for complex actions, the definite runtime traces of a complex action represent the desired formal semantics that is suitable to prove the soundness and completeness of the static analysis for complex actions.

8

STATIC ANALYSIS OF COMPLEX ACTIONS

Based on the semantics of complex actions desirable and crucial properties of complex actions are identified that need to be satisfied at runtime when the actions are actually executed.

However, merely characterizing desirable properties does not contribute to the quality of complex actions. In addition, it needs to be statically verifiable whether the properties are satisfied by a given action. To this end, a static analysis algorithm is elaborated and its correctness and completeness is formally verified based on the semantics of complex actions introduced in the previous chapter.

8.1 REQUIREMENTS AND DESIRABLE PROPERTIES

The clear separation and a good coverage of the four orthogonal dimensions is clearly desirable for complex actions, in particular for complex actions applied in Emergency Management (EM). However, the resulting expressiveness comes at a price. Due to the separation of orthogonal dimensions, programmers have more liberties in the specification of actions. Thus flaws can be introduced more easily into the specification of actions than with more constrained languages. Moreover, assertions specified by programmers are not necessarily satisfied at runtime.

Therefore, complex actions call for a strong semantic analysis capable of identifying actions with undesired behavior and actions not complying with their specification in order to prevent issues at runtime.

8.1.1 *Undesired Behavior of Complex Actions*

The following examples demonstrate some of the undesired behavior of actions that may emerge at runtime and should therefore be identified by the static analysis which is elaborated in the following.

Recall that a complex action is formally represented by a triple $C = (W, D, H)$ of temporal formulas corresponding to its temporal dependencies, domain knowledge, and temporal assertions, respectively. Moreover, each runtime trace of C describes one possible instance of the execution of the complex action.

EXAMPLE 8.1. Consider the complex action $C = (W, \top, \top)$ with

$$W = a_{succ} \leqslant b_{succ}$$

As a_{succ} and b_{succ} are observed variables their values cannot be affected by the Event Processing System (EPS). Therefore, when the complex action C is initiated, the sub-actions a and b are simply concurrently requested. However, at runtime a may fail whereas b is successful and the system has no means to prevent this behavior. Accordingly, the temporal dependencies W that have been specified by the user are not necessarily satisfied at runtime.

It is highly desirable that programmers can rely on the fact that the specified temporal dependencies of the complex action will be actually satisfied at runtime. Therefore, the static analysis needs to be capable to reject complex actions as the one from Example 8.1 that do not necessarily satisfy their temporal dependencies.

EXAMPLE 8.2. Consider the complex action $C = (W, \top, \top)$ with

$$W \;=\; a_{succ} + 5 \leqslant b_{init} \;\wedge\; b_{succ} - 3 \leqslant a_{init}$$

At runtime a will only be initiated after b has been successful and b will only be initiated after a has been successful. Thus, because of this cyclic dependency between a_{init} and b_{init} neither a nor b will be initiated at runtime.

EXAMPLE 8.3. Consider the complex action $C = (W, \top, \top)$ with

$$W \;=\; a_{succ} + 5 \leqslant b_{init} \;\wedge\; b_{succ} - 7 \leqslant a_{init}$$

In analogy to Example 8.2, neither a nor b will be initiated by the EPS. However, in contrast to the previous example there are indeed variable assignments σ that satisfy W and for which furthermore holds $\sigma(a_{init}) < \infty$ and $\sigma(b_{init}) < \infty$ like, for instance,

$$\sigma = \{0/\bot_{init}, 2/a_{init}, 3/a_{succ}, \infty/a_{fail}, 8/b_{init}, 9/b_{succ}, \infty/b_{fail}, \}$$

However, none of these variable assignments corresponds to the way actions are executed by the EPS, that is, none of them corresponds to a runtime trace.

EXAMPLE 8.4. Consider the complex action $C = (W, D, \top)$ with

$$\begin{aligned} W \;&=\; a_{succ} + 5 \leqslant b_{init} \\ D \;&=\; a_{fail} + 3 \leqslant b_{init} \end{aligned}$$

For this complex action there are actually instances that initiate the actions a and b. Eg, the runtime trace

$$\tau = \{1/\bot_{init}, 3/a_{init}, 5/a_{succ}, 7/a_{fail}, 12/b_{init}, \infty/b_{succ}, 13/b_{fail}\}$$

corresponds to such an instance of C. Note, however, that for this τ holds $\tau(a_{succ}) < \infty$ and $\tau(a_{fail}) < \infty$ which is rather unintuitive as it

implies that a is both successful and fails. Therefore τ is not definite as definite traces cannot have this unintuitive property.

However, the domain knowledge specifies that a always fails but the initiation of b depends on the success of a. Therefore for all definite runtime traces τ of C, which necessarily satisfy $W_{\mathcal{L}} \wedge D$ and $\tau(a_{succ}) = \infty$ or $\tau(a_{fail}) = \infty$, follows $\tau(b_{init}) = \infty$. Therefore the action b is never initiated by any instance of C that satisfies the intuitive property that actions are either successful or fail.

For each of the three complex actions from Example 8.2 to 8.4 there is a sub-action that is never initiated by any instance of the complex action at runtime. At least if only instances are considered with the natural assumption that sub-actions are either successful or fail. However, if sub-actions are never initiated they may aswell be removed from the complex action and therefore it is likely that the temporal dependencies do not entail the behavior that is intended by the programmer.

Therefore, the analysis needs to be able to reject complex actions, like the ones from Example 8.2 to 8.4, that contain sub-actions which are never initiated by any instance of the action at runtime.

EXAMPLE 8.5. Consider the complex action $C = (W, \top, H)$ with

$$\begin{aligned} W &= a_{succ} + 5 \leqslant b_{init} \\ H &= b_{succ} \leqslant a_{init} + 17 \end{aligned}$$

Although there are definite runtime traces that satisfy H, eg,

$$\tau = \{1/\bot_{init}, 3/a_{init}, 5/a_{succ}, \infty/a_{fail}, 12/b_{init}, 13/b_{succ}, \infty/b_{fail}\}$$

there are as well definite runtime traces that do not, eg,

$$\tau = \{2/\bot_{init}, 3/a_{init}, 11/a_{succ}, \infty/a_{fail}, 12/b_{init}, 23/b_{succ}, \infty/b_{fail}\}$$

Similar to the temporal dependencies of complex actions, their temporal assertions are only meaningful if they are actually satisfied at runtime. Temporal assertions are intended as a mean for programmers to specify the behavior they expect from the action when it is actually executed at runtime.

Accordingly, the analysis needs to reject complex actions that do not necessarily comply with their temporal assertions, such as the complex action from Example 8.5.

8.1.2 *Desirable Properties of Complex Actions*

From the examples of the previous section we derive three properties of complex actions, namely, viability, fairness, and compliance. Those three properties of complex actions are eventually verified by the static analysis that is elaborated in the following.

DEFINITION 8.1 (VIABILITY). A complex action $C = (W, D, H)$ is *viable* iff $W_{\mathcal{L}} \models W$.

By design, the EPS only determines values for affected variables and hence ignores all atomic formulas with observable variables on the right hand side of the inequation, that is, for a complex action $C = (W, D, H)$ the EPS only considers $W_{\mathcal{L}}$ for the execution of actions. Therefore, the temporal dependencies of C should correspond to the lower bound formulas $W_{\mathcal{L}}$.

Note that with the given definition of viability, it follows for all viable complex actions C that for each runtime trace τ of C the variable free formula $\tau(W)$ is satisfied. Moreover, the definition of viability effectively realizes a clear separation between temporal dependencies and temporal assertions. Eg, with $F = a_{succ} \leqslant b_{succ}$ the complex action $C = (F, F, \top)$ is *not* viable although every runtime trace of C satisfies F. However, F is clearly an assertion that cannot be actively enforced by the EPS but can only be observed at runtime. Consequently, F should not be combined with temporal dependency but should be specified as an assertion of C instead.

DEFINITION 8.2 (FAIRNESS). A complex action $C = (W, D, H)$ is *fair* iff for each $f_{init} \in var(W)$ there is a definite runtime trace τ of C with $\tau(f_{init}) < \infty$.

Example 8.2 to 8.4 from the previous section specify complex actions that contain sub-actions which are never initiated by any instance of the given complex actions at runtime.

It seem to be too strong a restriction to demand that all sub-actions are initiated by all instances of the complex action, as it would effectively prevent the specification of alternatives, such as, execute b if a is successful and c otherwise. However, it seems desirable to require that each sub-action is initiated by at least one instance of the complex action, that is, for all actions f there is a runtime trace τ with $\tau(f_{init}) < \infty$.

DEFINITION 8.3 (COMPLIANCE). A complex action $C = (W, D, H)$ is *compliant* iff for each definite runtime trace τ of C the variable free formula $\tau(H)$ holds.

As it has been discussed in the previous section the temporal assertions of complex actions are only meaningful if they are satisfied at runtime by each instance of the complex action, that is, if for each runtime trace τ the variable free formula $\tau(H)$ is satisfied.

8.1.3 *Requirements for the Semantic Analysis*

To be valuable in practice, an analysis algorithm for formal verification of complex actions and their properties should consider the following aspects.

SUITABILITY FOR PHYSICAL ACTIONS The amount of knowledge available on physical actions is usually very heterogeneous and may range from no a priori information on the duration and the result, eg, when complicated physical effects are involved, to comprehensive knowledge, eg, in case of actions that cannot fail.

Therefore, the analysis must be capable to scale with the amount of knowledge that is available. That is, very basic properties of actions should be verified without any a priori knowledge. Moreover, specific knowledge, if available, should be integrated into the analysis so as to verify more sophisticated properties of actions.

VERIFICATION AT COMPILE TIME Naturally, the analysis of complex actions must be static, that is, the analysis is carried out at compile time and identifies actions with undesired behavior that would emerge at runtime.

A static analysis helps programmers to identify undesired behavior of actions and to correct it before the actions are actually carried out and their effects actually manifest. If the undesired behavior is only detected at runtime, then it is often too late to correct the actions, in particular in EM.

CORRECTNESS AND COMPLETENESS The analysis needs to be correct in the sense that if it testifies the viability, fairness, and compliance of a complex action, then the complex action actually has these properties. If the analysis is not correct, it does not provide reliable information on the properties of complex actions which substantially reduces its usefulness.

Moreover, it is highly desirable that the analysis is complete in the sense that for each complex action the analysis algorithm terminates.

8.2 STATIC TEMPORAL ANALYSIS

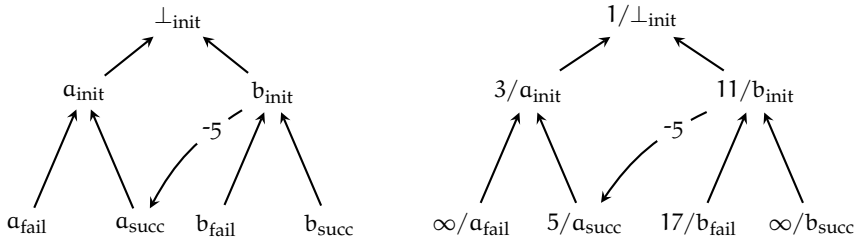
Although the formalization of properties of complex actions from Section 8.1.2 is precise in a mathematical sense it is not suitable to verify the properties of complex actions in an algorithmic manner. For each sufficiently interesting complex action which contains at least one sub-action that is actually initiated at runtime there are infinitely many runtime traces. However, as the properties universally quantify over runtime traces a direct algorithmic verification of them seems impossible.

Therefore, the following section elaborates an alternative but equivalent characterizations of the properties of complex actions that can indeed be algorithmically tested in finite time.

DEFINITION 8.4. A *dependency graph* G_F of a temporal formula F is a directed weighted graph $G = (V, E, w)$ with $V = \mathrm{var}(F)$, $E \subseteq V \times V$, and $w : E \to \mathbb{Q}$ such that for each sub-formula $u + d \leqslant v$ of F there is an edge $e = (v, u) \in E$ with $w(e) = -d$.

Note that dependency graphs correspond to distance graphs described by [DMP91] to verify the consistency of Simple Temporal Problem (STP) and to temporal distance graphs of [BE08a; Eck08] where they are used to derive information related to the garbage collection of events.

EXAMPLE 8.6. The axiomatic closure of the temporal dependencies $W = a_{succ} + 5 \leqslant b_{init}$ is represented by the graph G_{W_A} in Figure 8.1a. For the sake of readability, the labels of edges with weight zero are omitted from the graphical representation.



(a) G_{W_A} (b) G_{W_A} Augmented with τ

Figure 8.1: Two Dependency Graphs of W_A

In the following it is furthermore convenient to concisely represent runtime traces within dependency graphs. An example of such an extended dependency graph is contained in Figure 8.1b for the runtime trace

$$\tau = \{1/\bot_{init}, 3/a_{init}, \infty/a_{fail}, 5/a_{succ}, 11/b_{init}, \infty/b_{succ}, 17/b_{fail}\}$$

The following definitions provides some convenient notions that are related to paths in (dependency) graphs and which are commonly used in the literature, eg, in [Cor+01].

DEFINITION 8.5. For a graph $G = (V, E, w)$ the notation $v_0 \overset{p}{\rightsquigarrow} v_n$ indicates that $p = \langle v_0, \ldots, v_n \rangle$ is a *path* in G with $(v_i, v_{i+1}) \in E$ for $0 \leqslant i < n$. To emphasize that a path is a proper edge in G, ie, $n = 1$, the notation $v_0 \to v_n$ is used.

The number of edges of p is denoted $\mathrm{length}(p) = n$ and the *weight* or *distance* of p is denoted $w(p) = \sum_{0 \leqslant i < n} w(v_i, v_{i+1})$. Moreover, for a node $s \in V$ the notion $s \in p$ indicates that s lies on the path p, that is, there is a $0 \leqslant k \leqslant n$ such that $s = v_k$.

DEFINITION 8.6. For a complex action $C = (W, D, H)$ the set of all nodes on any path from v to $\bot_{\text{init}}$ in $G_{W_A \wedge D}$ is denoted

$$\rho(v) = \left\{ s \in p \;\middle|\; p \text{ with } v \overset{p}{\rightsquigarrow} \bot_{\text{init}} \text{ is a path in } G_{W_A \wedge D} \right\}.$$

Note that for all variables $v \in \text{var}(W_A)$ there is a path $v \rightsquigarrow \bot_{\text{init}}$ in G_{W_A}. And although the definition of ρ seems to have appeared from nowhere it will turn out in the following that there is a strong connection between ρ and the fairness of actions.

8.2.2 *Basic Ideas and Informal Introduction*

This section motivates and sketches the major ideas that are used throughout the entire formal part concerned with the static analysis in Chapter 9. Most notably, it provides the intuition for the relationship between definite runtime traces τ of a complex action $C = (W, D, H)$ with $\infty \in \text{im}(\tau)$ and solutions σ of the temporal formula $W_A \wedge D$ with $\infty \notin \text{im}(\sigma)$.

ABSENCE OF DEFINITE RUNTIME TRACES Runtime traces are designed to map variables on a cycle in G_{W_A} to infinity. Thus, if G_{W_A} of a viable complex action $C = (W, D, H)$ contains a cycle it follows that there is at least one f_{init} with $\tau(f_{\text{init}}) = \infty$ for all runtime traces τ of C.

Moreover, paths in $G_{W_A \wedge D}$ denote (indirect) dependencies between variables. More precisely, if there is a path $u \overset{p}{\rightsquigarrow} v$ in $G_{W_A \wedge D}$ then all runtime traces of C satisfy $u - w(p) \leqslant v$. Consider, for instance, the action $C = (W, D, \top)$ with

$$\begin{aligned} W &= a_{\text{succ}} + 5 \leqslant b_{\text{init}} \\ D &= a_{\text{fail}} + 7 \leqslant b_{\text{init}} \end{aligned}$$

and the corresponding dependency graph in Figure 8.2. In this graph there is a path $b_{\text{init}} \rightsquigarrow a_{\text{fail}}$ and $b_{\text{init}} \rightsquigarrow a_{\text{succ}}$. However, for definite runtime traces τ holds that $\tau(a_{\text{fail}}) = \infty$ or $\tau(a_{\text{succ}}) = \infty$ and thus, because of the indicated dependencies, it follows for all definite runtime traces τ of C that $\tau(b_{\text{init}}) = \infty$. Hence, there is no instance of C that actually initiates b.

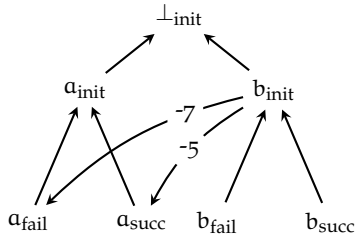


Figure 8.2: The Dependency Graph $G_{W_A \wedge D}$

These observations generalize to the following statement. If there is a f_{init} such that f_{init} is contained in a cycle of G_{W_A} or if there is a g_{init} with $g_{succ} \in \rho(f_{init})$ and $g_{fail} \in \rho(f_{init})$ then for all runtime traces τ holds $\tau(f_{init}) = \infty$.

CONSTRUCTION OF DEFINITE RUNTIME TRACES Consider, for instance, the action $C = (W, D, H)$ with

$$\begin{aligned} W &= a_{succ} + 5 \leqslant b_{init} \\ D &= a_{succ} + 13 \leqslant b_{succ} \wedge b_{fail} - 9 \leqslant a_{succ} \\ H &= b_{fail} \leqslant a_{succ} \end{aligned}$$

and the variable assignment

$$\sigma = \{2/\bot_{init}, 5/a_{init}, 7/a_{fail}, 23/a_{succ}, 29/b_{init}, 37/b_{succ}, 30/b_{fail}\}$$

which is included in the dependency graph of G_{W_A} in Figure 8.3. Note that $\sigma \models W_A \wedge D$ and $\sigma \not\models H$.

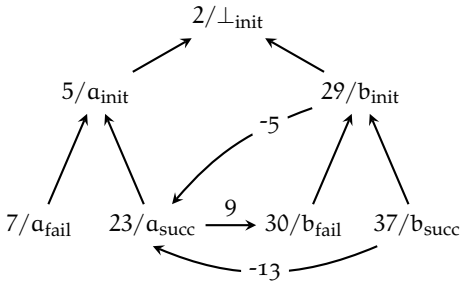


Figure 8.3: $G_{W_A \wedge D}$ and σ with $\sigma \models W_A \wedge D$ and $\sigma \not\models H$

From the given variable assignment σ we can construct the following scenario Δ such that for the runtime trace τ_{Δ, W_A} holds $\tau_{\Delta, W_A} = \sigma$:

$$\Delta = \{2/\bot_{init}, 3/a_{init}, 2/a_{fail}, 18/a_{succ}, 1/b_{init}, 8/b_{succ}, 1/b_{fail}\}$$

Note that the runtime trace τ is obviously not definite. However, it can be made definite by constructing a new scenario Δ' from Δ by adapting the values of $\Delta'_{a_{fail}}$ and $\Delta'_{b_{succ}}$ to ∞. Then with $\tau' = \tau_{\Delta', W_A}$ it follows

$$\tau' = \{2/\bot_{init}, 5/a_{init}, \infty/a_{fail}, 23/a_{succ}, 29/b_{init}, 37/b_{succ}, \infty/b_{fail}\}$$

Note that furthermore $\tau'(a_{succ}) = \tau(a_{succ})$ and $\tau'(v) \geqslant \tau(v)$ for all variables $v \in var(W_A)$. Therefore, as $\tau \not\models H$ holds it follows that

$$\tau'(a_{succ}) = \tau(a_{succ}) < \tau(b_{fail}) - 9 \leqslant \tau'(b_{fail}) - 9$$

and thus τ' is a definite runtime trace of C with $\tau' \not\models H$.

Be aware that this example was carefully chosen to avoid some rather technical details and to provide an easy to understand explanation. However, in general for a complex action $C = (W, D, H)$ with

$W_{\mathcal{A}}$ is acyclic and for all $f_{init}, g_{init} \in var(W_{\mathcal{A}})$ follows $g_{succ} \notin \rho(f_{init})$ or $g_{fail} \notin \rho(f_{init})$ the following proposition holds. For each variable assignment σ with $\sigma \models W_{\mathcal{A}} \wedge D$ and each variable $\nu \in var(W_{\mathcal{A}})$ there is a definite runtime trace τ of C with $\tau(\nu) = \sigma(\nu)$ and $\tau(u) \geqslant \sigma(u)$ for all $u \in var(W_{\mathcal{A}})$.

Accordingly, under the given premises, a solution of $W_{\mathcal{A}} \wedge D$ can be transformed into a definite runtime trace of C.

REDUCTION TO VARIABLE ASSIGNMENTS IN $\mathbb{Q}$ To construct a variable assignment with values in $\mathbb{Q}$ from a definite runtime trace τ with $\infty \in im(\tau)$ we can take advantage of the following observation. The value ∞ denotes an arbitrarily large time-point which is larger than any finite time-point. However, in practice, for each specific runtime trace the arbitrarily large time-point ∞ can be identified by a sufficiently large time-point in $\mathbb{Q}$. And therefore each runtime trace τ can be translated into an equivalent variable assignments τ^{fin} with $im(\tau^{fin}) \subseteq \mathbb{Q}$.

Consider, for instance, the action $C = (W, D, H)$ with

$$\begin{aligned} W &= a_{succ} + 5 \leqslant b_{init} \\ D &= a_{fail} + 23 \leqslant b_{fail} \wedge b_{fail} - 9 \leqslant a_{succ} \\ H &= b_{fail} \leqslant a_{succ} + 30 \end{aligned}$$

and the definite runtime trace $\tau_{\Delta, W_{\mathcal{A}}}$ of C with

$$\Delta = \{0/\bot_{init}, 2/a_{init}, 7/a_{fail}, \infty/a_{succ}, \infty/b_{init}, \infty/b_{succ}, \infty/b_{fail}\}$$

which is also included in the dependency graph of $G_{W_{\mathcal{A}}}$ in Figure 8.4. Note that all finite values are smaller than 8 and thus the value ∞ can be identified with values larger or equal to 8.

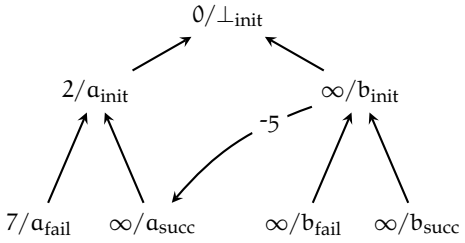


Figure 8.4: $G_{W_{\mathcal{A}}}$ and τ with $\tau \models W_{\mathcal{A}} \wedge D$ but $\infty \in im(\tau)$

However, to satisfy $W_{\mathcal{A}}$ the value ∞ cannot be simply identified by a single value, but instead it needs to be identified by several values larger than 7 that satisfy the constraints of $W_{\mathcal{A}}$. This is realized by extending $W_{\mathcal{A}}$ to $W_{\mathcal{A}}^{fin}$ with formulas that ensure that all variables ν

with $\tau(v) = \infty$ become larger than 7 and by adapting the scenario Δ to Δ^{fin} whereby $\Delta^{\text{fin}}(v) = 0$ for all v with $\tau(v) = \infty$.

$$W_{\mathcal{A}}^{\text{fin}} = W_{\mathcal{A}} \wedge \bigwedge_{\tau(v)=\infty} \bot_0 + 8 \leqslant v$$

$$\Delta^{\text{fin}} = \{0/\bot_{\text{init}}, 2/a_{\text{init}}, 7/a_{\text{fail}}, 0/a_{\text{succ}}, 0/b_{\text{init}}, 0/b_{\text{succ}}, 0/b_{\text{fail}}\}$$

In this way, the fixpoint iteration that determines the runtime trace $\tau' = \tau_{\Delta^{\text{fin}}, W_{\mathcal{A}}^{\text{fin}}}$ automatically provides suited values for all variables v with $\tau(v) = \infty$ so that $\tau'(v) \geqslant 8$ and $\tau' \models W_{\mathcal{A}}$. The runtime trace and the corresponding graph $G_{W_{\mathcal{A}}^{\text{fin}}}$ is depicted in Figure 8.5. Moreover, note that only variables that are mapped to ∞ by τ are affected by the described adaptations whereas variables that are mapped to finite values remain the same.

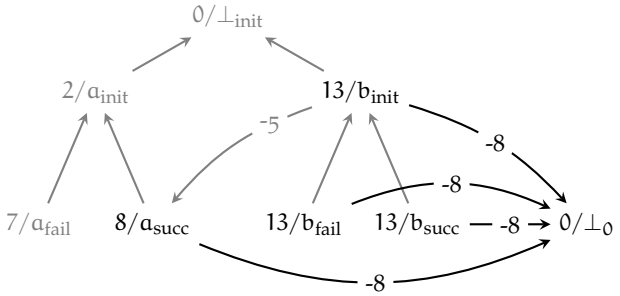


Figure 8.5: $G_{W_{\mathcal{A}}^{\text{fin}}}$ and τ' with $\infty \notin \text{im}(\tau')$ but $\tau' \not\models D$

Although $\infty \notin \text{im}(\tau')$, it does not follow that $\tau' \models D$ because variables that have been mapped to ∞ before and thus trivially satisfied all atoms of D can be inappropriately chosen by τ'.

However, the promising idea to apply the transformation to the temporal dependencies $W_{\mathcal{A}} \wedge D$ instead of $W_{\mathcal{A}}$ introduces further issues here. $W_{\mathcal{A}}^{\text{fin}} \wedge D$ contains cycles and runtime traces have been designed to map variables that are contained in a cycle to correspond to the behavior of the EPS. Therefore, for the runtime trace $\tau'' = \tau_{\Delta^{\text{fin}}, W_{\mathcal{A}}^{\text{fin}} \wedge D}$ there are variables with $\tau''(v) = \infty$ as Figure 8.6 illustrates.

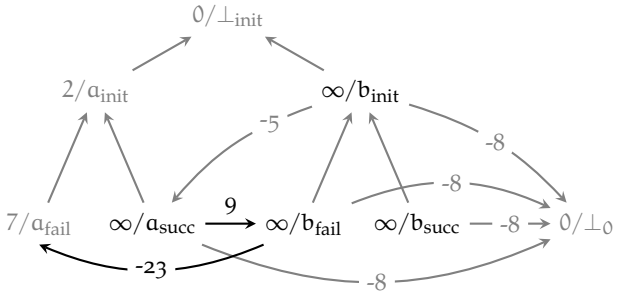


Figure 8.6: $G_{W_{\mathcal{A}}^{\text{fin}} \wedge D}$ and τ'' with $\tau'' \models W_{\mathcal{A}} \wedge D$ but $\infty \in \text{im}(\tau'')$

The final solution that obtains a variable assignment τ^{fin} that satisfies $W_{\mathcal{A}} \wedge D$ and $\infty \notin \text{im}(\tau^{\text{fin}})$ is as surprising as beautiful. Although the operator T has been designed to map variables on a cycle to ∞, this property only holds if $\mathsf{T} \uparrow 0 = \emptyset$. If, however, $\mathsf{T} \uparrow 0$ is instead determined in a suited manner, eg, by $\mathsf{T} \uparrow 0 = \tau'$, then for $\tau^{\text{fin}} = \text{lfp}(\mathsf{T}_{\Delta^{\text{fin}}, W_{\mathcal{A}}^{\text{fin}} \wedge D} \uparrow)$ holds $\tau^{\text{fin}} \models W_{\mathcal{A}} \wedge D$ and $\text{im}(\tau^{\text{fin}}) \subseteq \mathbb{Q}$. Note that furthermore the primal runtime trace τ can be reconstructed from τ^{fin} by substituting all values larger than 7 with ∞.

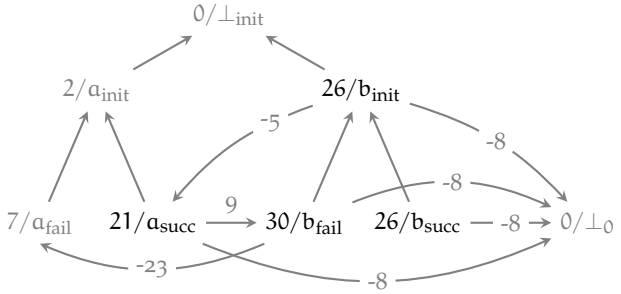


Figure 8.7: $G_{W_{\mathcal{A}}^{\text{fin}} \wedge D}$ and τ^{fin} with $\tau^{\text{fin}} \models W_{\mathcal{A}} \wedge D$ and $\infty \notin \text{im}(\tau^{\text{fin}})$

Again, this example was carefully chosen. In general it can be shown that if C is fair and $W_{\mathcal{A}} \wedge D$ is consistent, then for every runtime trace τ the corresponding fixpoint τ^{fin} exists and is determined within a finite amount of iteration steps. Moreover, it can be shown that $\tau^{\text{fin}} \models_{\mathbb{Q}} W_{\mathcal{A}} \wedge D$ and furthermore $\tau^{\text{fin}} \models_{\mathbb{Q}} H \implies \tau \models_{\mathbb{P}} H$.

RELATIONS TO FAIRNESS The observations made in the first two examples can be generalized to the following. If there is a cycle in $G_{W_{\mathcal{A}}}$ or if there are variables f_{init} and g_{init} such that $g_{\text{succ}} \in \rho(f_{\text{init}})$ and $g_{\text{fail}} \in \rho(f_{\text{init}})$ then there is a sub-action a such that $\tau(a_{\text{init}}) = \infty$ for all runtime traces τ of C.

Conversely, suppose $G_{W_{\mathcal{A}}}$ is acyclic and for all $f_{\text{init}}, g_{\text{init}} \in \text{var}(W_{\mathcal{A}})$ holds $g_{\text{succ}} \notin \rho(f_{\text{init}})$ or $g_{\text{fail}} \notin \rho(f_{\text{init}})$. If $W_{\mathcal{A}} \wedge D$ is consistent, then there is a σ with $\sigma \models_{\mathbb{Q}} W_{\mathcal{A}} \wedge D$ and according to the observation from the previous examples for each sub-action a a definite runtime trace τ can be constructed from σ with $\tau(a_{\text{init}}) = \sigma(a_{\text{init}}) < \infty$. Therefore, as a_{init} has been arbitrarily chosen, it follows that C is fair.

Accordingly, to verify whether C is fair it suffices to verify that $G_{W_{\mathcal{A}}}$ is acyclic and that for all $f_{\text{init}}, g_{\text{init}} \in \text{var}(W_{\mathcal{A}})$ holds $g_{\text{succ}} \notin \rho(f_{\text{init}})$ or $g_{\text{fail}} \notin \rho(f_{\text{init}})$.

RELATIONS TO COMPLIANCE By definition, for each definite runtime trace τ of a complex action C holds $\tau \models_{\mathbb{P}} W_{\mathcal{A}} \wedge D$. Thus the question whether a complex action is compliant resembles the question whether $W_{\mathcal{A}} \wedge D \models_{\mathbb{Q}} H$ which can be verified by established algorithms [SK00; SV98; BB12b].

However, runtime traces map variables to values in $\mathbb{P}$ and thus there is no direct relation between runtime traces and solutions of $W_A \wedge D$ in $\mathbb{Q}$. Example 8.2, for instance, specifies a complex action C for which holds that there are runtime traces τ of C but none of those runtime traces is a solution of $W_A \wedge D$ in $\mathbb{Q}$. Moreover, Example 8.3 specifies a complex action C for which holds that none of the solutions in $\mathbb{Q}$ of $W_A \wedge D$ are runtime traces.

Nevertheless there are two important aspects that can be observed from the previous examples. If $C = (W, D, H)$ is a fair complex action with $W_A \wedge D$ consistent in $\mathbb{Q}$ then for each definite runtime trace τ of C there is a τ^{fin} with $\tau^{fin} \models W_A \wedge D$ and $\tau^{fin} \models_{\mathbb{Q}} H \implies \tau \models_{\mathbb{P}} H$. Thus if $W_A \wedge D \models_{\mathbb{Q}} H$ it follows for each runtime trace τ of C that $\tau \models_{\mathbb{P}} H$. Conversely, if $W_A \wedge D \not\models_{\mathbb{Q}} H$ then there is a σ with $\sigma \models_{\mathbb{Q}} W_A \wedge D$ and $\sigma \not\models_{\mathbb{Q}} H$ and therefore there is a definite runtime trace τ of C with $\tau \not\models_{\mathbb{P}} H$.

Accordingly, to verify whether each runtime trace τ of a fair action C satisfies H, that is, whether C is compliant, it actually suffices to verify that $W_A \wedge D \models_{\mathbb{Q}} H$.

8.2.3 *Analogies to Skolemization*

Skolemization [Bry+07] is a technique used in mathematical logic to construct a universal formula from a non-universal formula. Interestingly, Skolemization and the construction of τ^{fin} from τ share some remarkable commonalities that may help readers familiar with the matter to gain a better understanding of τ^{fin} and its relationships to the compliance of complex actions.

The Skolemization of a non-universal formula φ yields a universal formula φ^{sko}. However, the Skolemization introduces new function symbols to the signature of the underlying formal language and thus φ and φ^{sko} are not equivalent in the sense that they share the same models. Nevertheless, each model $M^{sko} \models \varphi^{sko}$ can be converted into a model of φ by simply omitting the interpretation of redundant function symbols that are only present in φ^{sko} and not in φ. Conversely, from each model $M \models \varphi$ an interpretation M^{sko} can be constructed that interprets the additional function symbols of φ^{sko} in the right manner so that $M^{sko} \models \varphi^{sko}$. As a consequence, under certain conditions, it suffices to consider the universal formula φ^{sko} to draw conclusions on the satisfiability of φ.

Similar to the relationship between M and M^{sko}, definite runtime traces τ and solutions of $W_A \wedge D$ in $\mathbb{Q}$ cannot be directly related. Definite runtime traces necessarily map some variables to infinity and thus $\infty \in \mathrm{im}(\tau)$ but $\infty \notin \mathbb{Q}$ and thus definite runtime traces are no solutions of $W_A \wedge D$ in $\mathbb{Q}$ and vice versa. However, if the complex action $C = (W, D, H)$ obeys certain conditions, then for each definite runtime trace τ of C a corresponding τ^{fin} exists with $\infty \notin \mathrm{im}(\tau^{fin})$.

And conversely, from every solution of $W_{\mathcal{A}} \wedge D$ in $\mathbb{Q}$ a definite runtime trace τ can be constructed. As a consequence, very similar to the relationship between φ and φ^{sko}, under certain conditions it suffices to verify properties of the solution to $W_{\mathcal{A}} \wedge D$ in $\mathbb{Q}$ to draw conclusions on properties of definite runtime traces τ.

8.2.4 *Desirable Properties Reconsidered*

This section provides the formal foundations for the informal examples of the previous section. It establishes formally provable relationships between the desirable properties of complex actions described in Section 8.1.2 and properties of graphs and Disjunctive Temporal Problems (DTPs) that can be algorithmically tested. However, the rather technical proofs are omitted from this section. For the formal proofs refer to Chapter 9.

THEOREM 8.1. *A complex action* $C = (W, D, H)$ *is not fair only if the graph* $G_{W_{\mathcal{A}}}$ *is cyclic or there are variables* f_{init} *and* g_{init} *with* $g_{succ} \in \rho(f_{init})$ *and* $g_{fail} \in \rho(f_{init})$.

THEOREM 8.2. *A complex action* $C = (W, D, H)$ *with* $W_{\mathcal{A}} \wedge D$ *consistent is not fair if the graph* $G_{W_{\mathcal{A}}}$ *is cyclic or there are variables* f_{init} *and* g_{init} *with* $g_{succ} \in \rho(f_{init})$ *and* $g_{fail} \in \rho(f_{init})$.

The preceding theorems provide the formal foundations for the relationships between the fairness of actions and properties of the corresponding dependency graph. As a consequence, it is not mandatory to analyze all infinitely many runtime traces to verify the fairness of an action but instead it suffices to verify the properties of a finite graph.

THEOREM 8.3. *For a fair complex action* $C = (W, D, H)$ *with* $W_{\mathcal{A}} \wedge D$ *consistent holds* $W_{\mathcal{A}} \wedge D \models_{\mathbb{Q}} H$ *only if for all definite runtime traces* τ *of* C *holds* $\tau \models_{\mathbb{P}} H$.

THEOREM 8.4. *For a fair complex action* $C = (W, D, H)$ *with* $W_{\mathcal{A}} \wedge D$ *consistent holds* $W_{\mathcal{A}} \wedge D \models_{\mathbb{Q}} H$ *if for all definite runtime traces* τ *of* C *holds* $\tau \models_{\mathbb{P}} H$.

The prior two theorems provide the formal foundations for the relationships between the compliance of actions and the entailment $W_{\mathcal{A}} \wedge D \models_{\mathbb{Q}} H$. Accordingly, to verify the compliance of complex actions it is sufficient to verify the consistency of the DTP $W_{\mathcal{A}} \wedge D \wedge \neg H$ which is known to be decidable [TVP03; BB12a].

8.3 ANALYSIS ALGORITHM

The results presented in Section 8.2.4 form the basis of the analysis algorithm as they provide relationships between the viability, fairness

and compliance of complex actions and graph properties and the entailment of temporal formulas which can be decided in finite time.

8.3.1 *Pseudocode*

The static analysis algorithm Algorithm 1 takes as input a complex action $C = (W, D, H)$ with $W_{\mathcal{A}} \wedge D$ consistent in $\mathbb{Q}$ and determines whether C is viable, fair, and compliant.

Note that the consistency of the STP $W_{\mathcal{A}} \wedge D$ can be verified in a preprocessing step in polynomial time [DMP91]. Moreover, the restriction to complex actions C with $W_{\mathcal{A}} \wedge D$ consistent seems reasonable as otherwise there are variables that are mapped to infinity by any runtime trace of C.

Algorithm 1: Viability, Fairness, and Compliance Test

input : an action $C = (W, D, H)$ with $W_{\mathcal{A}} \wedge D$ consistent

if $W_{\mathcal{L}} \not\models_{\mathbb{Q}} W$ **then** `/* ensure viability */`
  fail C is not viable;

if $G_{W_{\mathcal{A}}}$ *is cyclic* **then** `/* ensure fairness */`
  fail C is not fair;

for $f_{\text{init}} \in \text{var}(W_{\mathcal{A}})$ **do**
  if $f_{\text{init}} \rightsquigarrow g_{\text{succ}}$ *and* $f_{\text{init}} \rightsquigarrow g_{\text{fail}}$ *are path in* $G_{W_{\mathcal{A}} \wedge D}$ **then**
    fail C is not fair;

if $W_{\mathcal{A}} \wedge D \not\models_{\mathbb{Q}} H$ **then** `/* ensure compliance */`
  fail C does not comply with H;

success C is viable, fair, and compliant;

The analysis algorithm is based on the following auxiliary functions which are applied to verify the properties described in the previous section to draw conclusions on the properties of the complex action.

CONSTRUCTING $G_{W_{\mathcal{A}}}$ AND $G_{W_{\mathcal{A}} \wedge D}$ Both formulas W and D are finite and it furthermore follows from Definition 8.4 that for the dependency graphs $G_{W_{\mathcal{A}}}$ and $G_{W_{\mathcal{A}} \wedge D}$ the number of vertices V has the order of $\mathcal{O}(|\text{var}(W)|)$ and the number of edges E has the order of $\mathcal{O}(w + d)$ whereby w and d denote the number of atomic formulas in W and D, respectively.

Accordingly both dependency graphs are finite and can be constructed in finite time.

CYCLE DETECTION IN G_{W_A} (LINE 3) Cycles contained in a directed weighted graph G can be detected by means of depth-first search [Cor+01] over G with a running time of $\mathcal{O}(V + E)$.

TRAVERSING PATHS IN $G_{W_A \wedge D}$ (LINE 6) In an (acyclic) graph all paths leading from f_{init} to $\bot_{init}$ can be determined with a running time of $\mathcal{O}(E)$. Accordingly, the running time of line 5-7 is in the order of $\mathcal{O}(V \cdot E)$.

ENTAILMENT OF TEMPORAL FORMULAS (LINE 1 AND 8) The entailment of temporal formulas $F \models_Q G$ is equivalent to the inconsistency of the DTP $F \wedge \neg G$ in Q. Moreover, checking the consistency of DTPs is known to be NP-hard [TVP03] and can be decided with exponential running time [ACG00; BD13].

Note that the given assessments of the running time is not very tight. However, as the analysis algorithm is executed at compile time, we would like to emphasis that the algorithm always terminates. The efficiency of the algorithm is a matter of secondary importance here.

8.3.2 *Correctness and Completeness*

Correctness and completeness are two substantial properties of algorithms meaning that the answer an algorithm gives is indeed correct and that the algorithm will eventually give an answer for any input, respectively.

CORRECTNESS The correctness of Algorithm 1 with respect to fairness follows from Theorem 8.1 and 8.2. Moreover, from Theorem 8.3 and 8.4 follows the correctness with respect to compliance and with $C = (W_\mathcal{L}, \top, W)$ if further follows the correctness with respect to viability.

Note that the algorithm is only correct in the sense that it terminates whenever it determines that one of the three properties is not satisfied. That is, if an action is not viable and not compliant, Algorithm 1 only outputs not viable. However, as all three properties are considered to be equally important for EM purposes and thus complex actions should always have all three properties this circumstance does not seem to be a major issue.

COMPLETENESS As a complex action $C = (W, D, H)$ is determined by a triple of finite formulas it follows from the discussion in Section 8.3.1 that Algorithm 1 terminates for each input. Hence, Algorithm 1 is complete with respect to viability, fairness, and compliance, respectively.

8.3.3 *Compliance Under Incomplete Knowledge*

The domain knowledge of a complex action C may be specified incompletely, as further and more detailed knowledge is wittingly or unwittingly omitted from the specification of the complex action C. Be aware that in particular in case of physical actions the domain knowledge may never be completely available or that temporal formulas may not be suited to formalize it. Eg, Section 7.2.3 indicates that the current formalization of conditional action omits domain knowledge that can be extracted from temporal conditions of its event queries.

As a consequence, the static analysis verifies the compliance of the action $C = (W, D, H)$ as it is specified by the programmer, but in reality the complex action always satisfies the additional domain knowledge D', that is, actually the compliance of $C' = (W, D \wedge D', H)$ should have been verified instead. So how does the compliance of C which is checked by the static analysis relate to the compliance of the more accurately formalized action C'?

If the complex action C is compliant then $W_A \wedge D \models_Q H$ and thus $W_A \wedge D \wedge D' \models_Q H$. Hence, all definite runtime traces τ of C' satisfy H and therefore from C compliant follows C' compliant. However, for certain actions may follow that $W_A \wedge D \not\models_Q H$ but $W_A \wedge D \wedge D' \models_Q H$ and thus C' may be compliant whereas C is not.

Accordingly, the compliance of C is a conservative approximation of the compliance of C'.[1] That is, if C' is not compliant the analysis always rejects the compliance of C but there are cases in which the compliance of C' is wrongly rejected as C is not compliant but C' is. In particular for EM the last observation is important as the domain knowledge of many physical actions intrinsically cannot be completely specified in a suitable manner and thus remains incomplete. However, an analysis that is too conservative is still acceptable whereas an analysis that it too optimistic is not, at least for EM.

8.3.4 *Variation for Non-definite Runtime Traces*

Although the restriction to consider only definite runtime traces of complex actions seems desirable, not only for EM purposes, this restriction can be easily omitted from the analysis algorithm.

PROPOSITION 8.5. *An action* $C = (W, D, H)$ *with* $W_A \wedge D$ *consistent is fair (considering also non-definite runtime traces) iff* G_{W_A} *is acyclic.*

PROPOSITION 8.6. *For an action* $C = (W, D, H)$ *that is fair (considering also non-definite runtime traces) and with* $W_A \wedge D$ *consistent holds* $W_A \wedge D \models_Q H$ *iff for all traces* τ *of* C *holds* $\tau \models_{\mathbb{P}} H$.

1 Note that the viability and fairness of complex actions is not related to their domain knowledge and thus the viability and fairness of C and C' coincide.

Both propositions follow from the proofs of Theorem 8.1 to 8.4 by omitting all references to definite runtime traces. Accordingly, if desired, Algorithm 1 can be adapted for non-definite runtime traces simply by omitting lines line 5 to 7.

8.3.5 *Revision of Prior Work*

In prior work presented in [HB13], the algorithm for the verification of the compliance of actions has been based on work proposed in [BB12b]. Thereby we made the incorrect assumption that [BB12b] describes a generalization of DTPs for the domain $\mathbb{P}$ that can verify whether $F \models_{\mathbb{P}} G$ holds by checking the inconsistency of the formula $F \wedge \neg G$ in $\mathbb{P}$.

However, the previous sections suggest, under the additional premise that $W_A \wedge D$ is consistent in $\mathbb{Q}$, that it actually suffices to verify whether $F \models_{\mathbb{Q}} G$ holds to draw conclusions on the compliance of actions. Accordingly, approaches verifying the consistency of DTPs in $\mathbb{Q}$, such as [SK00; SV98; BB12b], are actually sufficient to verify the compliance of actions.

8.4 TEMPORAL CONSTRAINT SATISFACTION PROBLEMS

Temporal constraint satisfaction is related to the proposed static analysis of complex actions in two aspects. First of all, the static analysis is based on algorithms that verify the consistency of STPs and DTPs. Moreover, there are extensions that introduce uncertainty into the determination of time-points by discriminating between free and contingent constraints, a notion that is related to affected and observed time-points. These extensions are then used to verify whether there exists solutions of constraints that can be found in an incremental fashion which is relates to the feasibility of the stepwise execution of actions at runtime.

However, temporal constraint satisfaction approaches are often related to planning and scheduling of composite action, an issue that is, by design, not considered for complex actions as it contradicts a clear separation of dimensions of actions. They furthermore do not consider the possibility of actions to fail and thus are restricted to domains not including ∞. Nevertheless, static plans for the execution of action that are obtained from the following approaches can indeed be specified as temporal dependencies of complex actions.

8.4.1 *Simple and Disjunctive Temporal Problems*

STPs and DTPs have been well studied in the literature [DMP91; SK00; SV98] as they are suited to express and to reason about temporal constraints which is closely related to scheduling, planning, and tem-

poral reasoning in general. STPs and DTPs are specific Constraint Satisfaction Problems (CSPs) which involve a set of variables over a continuous domain, usually $\mathbb{Q}$ or $\mathbb{R}$, and a set of (disjunctive) constraints that correspond to atomic temporal formulas from Definition 7.2.

Two relevant issues, in particular for DTPs, are deciding consistency and answering queries that are related to the solutions of the constraints [SV98]. Thereby, checking the consistency of STPs has a polynomial time complexity [DMP91] whereas checking the consistency of DTPs is known to be NP-hard [TVP03]. Nevertheless, developing efficient algorithms and polynomial-time approximation algorithms for the verification of the consistency of DTPs has been deeply investigated by the community [SV98; TP03; OC00]. Moreover, extensions of DTPs have been proposed [BB12b; TVP03] that are also capable of dealing with atomic constraints of the form $u + d < v$. Accordingly, for two temporal formulas F and G the consistency of $F \wedge \neg G$ can be verified by these approaches.

Beyond their relevance for scheduling and planning, DTPs and DTPs are also suited to automatically determine temporal relevance conditions that enable garbage collection of events that contribute to complex event queries [BB12b; BE08a; Eck08].

8.4.2 *Temporal Problems with Uncertainty*

Vidal and Fragier [VF99] propose Simple Temporal Problem under Uncertainty (STPU) that incorporate a notion of uncertainty into STPs. STPUs discriminate between free constraints, which correspond to the constraints of STPs, and contingent constraints whose duration cannot be decided by the system but is provided by the external world. In [VF99] varying notions of controllability are elaborated which are intended to replace the notion of consistency for STPUs and reflect, eg, whether a dynamic strategy for assigning values to free constraints exists that is guaranteed to satisfy all constraints in any situation.

In [VY05] STPUs are generalized to allow for disjunctions of free and contingent constraints. And similar to DTPs, a major issue has since been to elaborate efficient algorithms [MM05; PVY07; Ven+10] that verify the various notions of controllability proposed in [VF99].

Moreover, dynamic execution algorithm for temporal constraints with both preferences and uncertainty have been elaborated by Rossi, Venable, and Yorke-Smith [RVY04; RVY06].

8.4.3 *Temporal Problems with Predicates*

Another formalism dealing with uncertainty are Conditional Temporal Problems (CTPs) as they are proposed by Tsamardinos, Pollack, and Ramakrishnan [TPR03]. However, in contrast to STPUs the uncer-

tainty arises from the outcome of observation not from the uncertainty of the duration of contingent constraints.

CTPs are represented by distance graphs of STPs that are furthermore augmented by observation nodes which correspond to decision points and determine which of several alternative execution paths is to be taken. A notion of controllability that is similar to the corresponding notions of STPUs is elaborated.

Recently, the work on CTP has been further extended to incorporate contingent durations of actions [FRV10; Lan+13].

9 FORMAL PROOFS

This chapter provides the formal proofs of the theorems Theorem 8.1 to Theorem 8.4 which are only informally motivated in the previous chapter.

But before we actually prove these theorems, we will restate the most relevant definitions from the previous chapters and introduce some convenient notations. Moreover, we will elaborate particular properties of runtime traces that facilitate the following proofs.

9.1 PRELIMINARIES

DEFINITION 7.10. The *preconditions* for the determination of a variable $v \in \mathrm{var}(W)$ of the temporal dependencies W are denoted

$$\mathrm{pre}_W(v) = \{u + d \mid u + d \leqslant v \text{ is sub-formula of } W\}$$

DEFINITION 7.11. For a temporal formula W and a runtime scenario Δ the operator $T_{\Delta,W}$ maps variable assignments to variable assignments with

$$T_{\Delta,W}(\sigma) = \left\{ \left(\max\{\sigma(\mathrm{pre}_W(v))\} + \Delta_v \right) \Big/ v \;\middle|\; \mathrm{var}(\mathrm{pre}_W(v)) \subseteq \mathrm{dom}(\sigma) \right\}$$

whereby $\max \emptyset = 0 \in \mathbb{P}$.

DEFINITION 7.13. For temporal dependencies W the variable assignment τ is called *runtime trace* of W iff there is a scenario Δ such that with $\mathbf{T} = \mathrm{lfp}(T_{\Delta,W_\mathcal{A}} \uparrow)$ holds

$$\tau|_{\mathrm{dom}(\mathbf{T})} = \mathbf{T}$$
$$\forall v \notin \mathrm{dom}(\mathbf{T}) : \ \tau(v) = \infty$$

DEFINITION 7.15. A runtime trace τ is a *definite runtime trace* iff

$$\forall f_{\mathrm{init}} \in \mathrm{dom}(\tau) : \ \tau(f_{\mathrm{succ}}) = \infty \vee \tau(f_{\mathrm{fail}}) = \infty$$

DEFINITION 7.16. A variable assignment τ is called *runtime trace of* $C = (W, D, H)$ iff τ is a runtime trace of $W_\mathcal{L}$ and $\tau \models D$.

DEFINITION 8.6. For a complex action $C = (W, D, H)$ the set of all nodes on any path from v to $\bot_{\mathrm{init}}$ in $G_{W_\mathcal{A} \wedge D}$ is denoted

$$\rho(v) = \{s \in p \mid p \text{ with } v \overset{p}{\rightsquigarrow} \bot_{\mathrm{init}} \text{ is a path in } G_{W_\mathcal{A} \wedge D}\}.$$

DEFINITION 9.1. The following definition generalizes $\mathsf{T}\uparrow$ from Definition 7.12 so that the value of $\mathsf{T}\uparrow 0$ can be determined by a variable assignment σ.

$$\begin{aligned}\mathsf{T}^{\sigma}\uparrow 0 &= \sigma\\ \mathsf{T}^{\sigma}\uparrow n+1 &= \mathsf{T}\big(\mathsf{T}^{\sigma}\uparrow n\big)\end{aligned}$$

In the primal definition of $\mathsf{T}\uparrow$ the value of $\mathsf{T}\uparrow 0$ has always been committed to $\emptyset$.

Remark. For the sake of readability the indices of the operator T may be omitted. Eg, T, $\mathbf{T}$ and $\mathsf{T}\uparrow$ will be often used as abbreviations for $\mathsf{T}_{\Delta,W_{\mathcal{A}}}$, $\mathbf{T}^{\emptyset}_{\Delta,W_{\mathcal{A}}}$ and $\mathsf{T}^{\emptyset}_{\Delta,W_{\mathcal{A}}}\uparrow$ in the following.

DEFINITION 9.2. The domain of the fixpoint $\mathbf{T}^{\emptyset}_{\Delta,W_{\mathcal{A}}}$ merely depends on the temporal formula $W_{\mathcal{A}}$ and is independent of the scenario Δ. Therefore we adopt the notion

$$\operatorname{dom}(W) = \operatorname{dom}\big(\mathbf{T}^{\emptyset}_{\Delta,W_{\mathcal{A}}}\big)$$

DEFINITION 9.3. For a complex action $C = (W, D, H)$ the length of the *longest path* from ν to $\bot_{\text{init}}$ in $G_{W_{\mathcal{A}}}$ is denoted

$$\lambda(\nu) = \max\big\{\text{length}(p) \;\big|\; p \text{ with } \nu \overset{p}{\rightsquigarrow} \bot_{\text{init}} \text{ is a path in } G_{W_{\mathcal{A}}}\big\}$$

Moreover, if there is a path $\nu \overset{p}{\rightsquigarrow} \bot_{\text{init}}$ that contains a cycle then $\lambda(\nu) = \infty$ as the length of the longest path is unbounded.

DEFINITION 9.4. For a variable assignment σ with $\operatorname{im}(\sigma) \subseteq \mathbb{Q}$ and a temporal formula F the notation $\sigma \models_{\mathbb{Q}} F$ means that the variable free formula $\sigma(F)$ holds under the axioms of $\mathbb{Q}$.

DEFINITION 9.5. For temporal formulas F and G the notation $F \models_{\mathbb{Q}} G$ denotes that for all variable assignments σ with $\operatorname{im}(\sigma) \subseteq \mathbb{Q}$ and $\sigma \models_{\mathbb{Q}} F$ follows $\sigma \models_{\mathbb{Q}} G$.

DEFINITION 9.6. The notations $\sigma \models_{\mathbb{P}} F$ and $F \models_{\mathbb{P}} G$ are defined likewise with $\mathbb{P}$ substituted for $\mathbb{Q}$.

Remark. For the sake of readability $\mathbb{Q}$ and $\mathbb{P}$ may be omitted from $\models_{\mathbb{Q}}$ and $\models_{\mathbb{P}}$ if ambiguities can be ruled out.

THEOREM 9.1 (DECHTER, MEIRI, AND PEARL [DMP91]). *A Simple Temporal Problem (STP) F is consistent in $\mathbb{Q}$ iff there is no negative cycle in G_F. Moreover, if $\nu \overset{p}{\rightsquigarrow} u$ is a path in G_F and $\sigma \models_{\mathbb{Q}} F$ then $\sigma \models_{\mathbb{Q}} u \leqslant \nu + w(p)$.*

The following theorem is an adaptation of the widely recognized relationship of semantic properties and entailment of formulas in mathematical logic [Bry+07] adapted to temporal formulas.

THEOREM 9.2. *For temporal formulas G and H holds $G \models_{\mathbb{Q}} H$ iff the Disjunctive Temporal Problem (DTP) $G \wedge \neg H$ is inconsistent in $\mathbb{Q}$.*

Remark. For the sake of simplicity, we assume without loss of generality that for all temporal formulas F that are considered in the following neither $\bot$ nor $\top$ is sub-formula of F.

LEMMA 9.3. *For temporal formula W and a variable $v \in \mathrm{var}(W)$ follows $v \in \mathrm{dom}(T \uparrow n)$ if and only if $\lambda(v) < n$.*

Proof. $\Longrightarrow$: If $\lambda(v) \geqslant n$ then there is a path p in G_{W_A} such that $p = \langle v_n, \ldots, v_0 \rangle$ and $v_n = v$ and $v_0 = \bot_{\text{init}}$. Accordingly, W contains the sub-formulas

$$(v_0 + d_0 \leqslant v_1),\ (v_1 + d_1 \leqslant v_2),\ \ldots,\ (v_{n-1} + d_{n-1} \leqslant v_n)$$

As $v_i \in \mathrm{pre}(v_{i+1})$ for $0 \leqslant i < n$, it follows that $v_{i+1} \in \mathrm{dom}(T \uparrow i+1)$ only if $v_i \in \mathrm{dom}(T \uparrow i)$. Thus,

$$v_n \in \mathrm{dom}(T \uparrow n) \implies \ldots \implies v_0 \in \mathrm{dom}(T \uparrow 0)$$

But as $v_0 \notin \emptyset = \mathrm{dom}(T \uparrow 0)$ it follows $v_n \notin \mathrm{dom}(T \uparrow n)$.

$\Longleftarrow$: By induction. *Base Case* ($n = 1$): From $\lambda(v) < 1$ follows $v = \bot_{\text{init}}$ and as $\mathrm{pre}(\bot_{\text{init}}) = \emptyset$ it further follows that $v \in \mathrm{dom}(T \uparrow 1)$.

Inductive Case ($n \to n+1$): If $\lambda(v) < n+1$, then for $u \in \mathrm{var}\big(\mathrm{pre}(v)\big)$ holds

$$\begin{aligned}
\lambda(u) &= \max\big\{\mathrm{length}(p) \;\big|\; u \overset{p}{\rightsquigarrow} \bot_{\text{init}}\big\} \\
&= \max\big\{\mathrm{length}(p) \;\big|\; v \to u \overset{p}{\rightsquigarrow} \bot_{\text{init}}\big\} \\
&< \max\big\{\mathrm{length}(p) \;\big|\; v \overset{p}{\rightsquigarrow} \bot_{\text{init}}\big\} \\
&= \lambda(v) < n+1
\end{aligned}$$

Hence, for all $u \in \mathrm{var}\big(\mathrm{pre}(v)\big)$ follows $\lambda(u) < n$ and furthermore, by induction hypothesis, $u \in \mathrm{dom}(T \uparrow n)$. Thus, according to the definition of T follows $v \in \mathrm{dom}(T \uparrow n+1)$. □

THEOREM 9.4. *If $v \in \mathrm{dom}(T \uparrow n)$, then*

$$\big(T \uparrow n\big)(v) = \max\Big\{ \sum_{s \in p} \Delta_s - w(p) \;\Big|\; v \overset{p}{\rightsquigarrow} \bot_{\text{init}} \Big\}.$$

Proof. By induction. *Base Case* ($n = 0$): As $\mathrm{dom}(T \uparrow 0) = \emptyset$ there is no $v \in \mathrm{dom}(T \uparrow 0)$ and thus the implication is trivially satisfied.

Inductive Case ($n \to n+1$): According to the definition of T, from $v \in \mathrm{dom}(T \uparrow n+1)$ follows

$$\mathrm{var}(\mathrm{pre}(v)) \subseteq \mathrm{dom}(T \uparrow n)$$

Therefore, for all $u \in \mathrm{var}(\mathrm{pre}(v))$ follows by induction hypothesis that

$$\big(T \uparrow n\big)(u) = \max\Big\{ \sum_{s \in p} \Delta_s - w(p) \;\Big|\; u \overset{p}{\rightsquigarrow} \bot_{\text{init}} \Big\}.$$

Without loss of generality $\mathrm{pre}(v) = \{v_1 + d_1, \ldots, v_k + d_k\}$ and as $d_i = -w(v, v_i)$ it follows

$$\begin{aligned}
(T \uparrow n+1)(v) &= \max\Big\{(T \uparrow n)(\mathrm{pre}(v))\Big\} + \Delta_v \\
&= \max_{1 \leqslant i \leqslant k} \Big\{(T \uparrow n)(v_i) + d_i\Big\} + \Delta_v \\
&= \max_{1 \leqslant i \leqslant k} \Big\{ \max\Big\{ \sum_{s \in p} \Delta_s - w(p) \;\Big|\; v_i \overset{p}{\rightsquigarrow} \bot_{\mathrm{init}}\Big\} + d_i\Big\} + \Delta_v \\
&= \max\Big\{ \sum_{s \in p} \Delta_s + \Delta_v - w(p) + d_i \;\Big|\; v \to v_i \overset{p}{\rightsquigarrow} \bot_{\mathrm{init}}\Big\} \\
&= \max\Big\{ \sum_{s \in p} \Delta_s + \Delta_v - w(p) - w(v, v_i) \;\Big|\; v \to v_i \overset{p}{\rightsquigarrow} \bot_{\mathrm{init}}\Big\} \\
&= \max\Big\{ \sum_{s \in p} \Delta_s - w(p) \;\Big|\; v \overset{p}{\rightsquigarrow} \bot_{\mathrm{init}}\Big\}
\end{aligned}$$

□

THEOREM 9.5. *If* $\sigma \subseteq \sigma'$ *then* $T(\sigma) \subseteq T(\sigma')$*, that is,* T *is monotonic [ABW88].*

Proof. Suppose $\sigma \subseteq \sigma'$. If $v \in \mathrm{dom}(T(\sigma))$ then

$$\mathrm{var}(\mathrm{pre}(v)) \subseteq \mathrm{dom}(\sigma)$$

However, as $\sigma' \supseteq \sigma$ it follows

$$\mathrm{var}(\mathrm{pre}(v)) \subseteq \mathrm{dom}(\sigma')$$

and thus $u \in \mathrm{dom}(T(\sigma'))$. Moreover, if $v \in \mathrm{dom}(T(\sigma))$ and $\sigma \subseteq \sigma'$, then, as $\mathrm{pre}(v)$ is independent of σ and σ'

$$\begin{aligned}
T(\sigma)(v) &= \max\{\sigma(\mathrm{pre}(v))\} + \Delta_v \\
&= \max\{\sigma'(\mathrm{pre}(v))\} + \Delta_v \\
&= T(\sigma')(v)
\end{aligned}$$

and therefore $T(\sigma) \subseteq T(\sigma')$. □

THEOREM 9.6. *For all conjunctive formulas* W *the fixpoint* $\mathbf{T}$ *exists and* $\mathbf{T} = T \uparrow n$ *for some* $n \in \mathbb{N}$.

Proof. As T is monotonic it follows that $T \uparrow n \subseteq T(T \uparrow n) = T \uparrow n+1$ and thus $\mathrm{dom}(T \uparrow n) \subseteq \mathrm{dom}(T \uparrow n+1)$.

As furthermore $\mathrm{dom}(T \uparrow n) \subseteq \mathrm{var}(W)$ for all $n \in \mathbb{N}$ and W is a finite conjunctive temporal formula it follows that $T \uparrow n = T \uparrow n+1$ for some n and therefore $\mathbf{T} = T \uparrow n$. □

COROLLARY. $v \in \mathrm{dom}(W)$ *if and only if* $\lambda(v) < \infty$.

COROLLARY. *If* $v \in \mathrm{dom}(W)$*, then*

$$\mathbf{T}(v) = \max\Big\{ \sum_{s \in p} \Delta_s - w(p) \;\Big|\; v \overset{p}{\rightsquigarrow} \bot_{\mathrm{init}}\Big\}.$$

LEMMA 9.7. *Suppose* $v \rightsquigarrow u$ *is a path in* $G_{W_{\mathcal{A}}}$. *If* $u \notin \mathrm{dom}(W)$ *then* $v \notin \mathrm{dom}(W)$.

Proof. If $v \rightsquigarrow u$ is a path in $G_{W_{\mathcal{A}}}$, then $v \rightsquigarrow u \rightsquigarrow \bot_{\mathrm{init}}$ is a path in $G_{W_{\mathcal{A}}}$. Moreover, if $u \notin \mathrm{dom}(W)$ then $\lambda(u) = \infty$ and thus

$$\lambda(v) \geqslant \lambda(u) = \infty.$$

Therefore, $v \notin \mathrm{dom}(W)$. □

LEMMA 9.8. *Suppose* $v \rightsquigarrow u$ *is a path in* $G_{W_{\mathcal{A}}}$ *and* $v, u \in \mathrm{dom}(W)$. *If* $\mathbf{T}(u) = \infty$ *then* $\mathbf{T}(v) = \infty$.

Proof. If $v \rightsquigarrow u$ is a path in $G_{W_{\mathcal{A}}}$ then $v \rightsquigarrow u \rightsquigarrow \bot_{\mathrm{init}}$ is also a path in $G_{W_{\mathcal{A}}}$. Thus,

$$\begin{aligned}
\mathbf{T}(v) &= \max\Big\{ \sum_{s \in p} \Delta_s - w(p) \;\Big|\; v \overset{p}{\rightsquigarrow} \bot_{\mathrm{init}} \Big\} \\
&\geqslant \max\Big\{ \sum_{s \in p} \Delta_s - w(p) \;\Big|\; v \overset{p}{\rightsquigarrow} \bot_{\mathrm{init}} \wedge u \in p \Big\} \\
&\geqslant \max\Big\{ \sum_{s \in p} \Delta_s - w(p) \;\Big|\; u \overset{p}{\rightsquigarrow} \bot_{\mathrm{init}} \Big\} \\
&= \mathbf{T}(u) = \infty
\end{aligned}$$

□

LEMMA 9.9. *For a runtime trace* $\tau_{\Delta, W_{\mathcal{A}}}$ *follows* $\tau_{\Delta, W_{\mathcal{A}}} \models W_{\mathcal{A}}$.

Proof. Suppose $u + d \leqslant v$ is a sub-formula of $W_{\mathcal{A}}$. If $v \notin \mathrm{dom}(W)$ then $\tau(v) = \infty$ and thus $\tau \models u + d \leqslant v$.

If $v \in \mathrm{dom}(W)$ then $\tau(v) = \mathbf{T}(v)$ and as $\mathbf{T}$ is a fixpoint it follows

$$\begin{aligned}
\mathbf{T}(v) = T(\mathbf{T})(v) &= \max\{\mathbf{T}(\mathrm{pre}(v))\} + \Delta_v \\
&\geqslant \mathbf{T}(u) + d
\end{aligned}$$

and therefore $\tau \models u + d \leqslant v$.

As $u + d \leqslant v$ has been arbitrarily chosen it follows that $\tau \models W_{\mathcal{A}}$. □

LEMMA 9.10. *For a temporal formula* W *with* $G_{W_{\mathcal{A}}}$ *acyclic follows if* $\sigma \models W_{\mathcal{A}}$ *then there is a scenario* Δ *such that* $\tau_{\Delta, W_{\mathcal{A}}} = \sigma$.

Proof. Suppose σ is a variable assignment with $\sigma \models W_{\mathcal{A}}$. Because $G_{W_{\mathcal{A}}}$ is acyclic it suffices to show that there is a scenario Δ such that $\sigma = \mathbf{T}_{\Delta, W_{\mathcal{A}}}$. Let Δ_v be determined by

$$\Delta_v = \begin{cases} \sigma(v) - \max\{\sigma(\mathrm{pre}(v))\} & \text{if } \max\{\sigma(\mathrm{pre}(v))\} < \infty \\ 0 & \text{otherwise} \end{cases}$$

For this scenario follows by induction $T \uparrow n + 1 = \sigma|_{\mathrm{dom}(T \uparrow n+1)}$.

Base Case ($n = 0$): Recall that $\max \emptyset = 0$. Thus

$$T \uparrow 1 = \left\{\Delta_{\perp_{\text{init}}} / \perp_{\text{init}}\right\} = \left\{\sigma(\perp_{\text{init}}) / \perp_{\text{init}}\right\} = \sigma|_{\text{dom}(T \uparrow 1)}$$

Inductive Case ($n \to n+1$): Suppose $v \in \text{dom}(T \uparrow n+1)$. Then for all $u \in \text{pre}(v)$ holds $u \in \text{dom}(T \uparrow n)$ thus by induction hypothesis follows for those u that $\big(T \uparrow n\big)(u) = \sigma(u)$. Hence

$$\begin{aligned}\big(T \uparrow n+1\big)(v) &= \max\left\{\big(T \uparrow n\big)(\text{pre}(v))\right\} + \Delta_v \\ &= \max\left\{\sigma(\text{pre}(v))\right\} + \Delta_v = \sigma(v)\end{aligned}$$

Accordingly, $T \uparrow n+1 = \sigma|_{\text{dom}(T \uparrow n+1)}$.

Therefore, if follows from Theorem 9.6 and the premise that $G_{W_{\mathcal{A}}}$ is acyclic that $\text{dom}(W) = \text{var}(W_{\mathcal{A}})$ and thus $\mathbf{T}_{\Delta, W_{\mathcal{A}}} = \sigma|_{\text{dom}(W)} = \sigma$. □

LEMMA 9.11. *Suppose τ is a runtime trace of $C = (W, D, H)$ and $G_{W_{\mathcal{A}}}$ acyclic, $W_{\mathcal{A}} \wedge D$ consistent in $\mathbb{Q}$, and for all variables f_{init} and g_{init} holds $g_{\text{succ}} \notin \rho(f_{\text{init}})$ or $g_{\text{fail}} \notin \rho(f_{\text{init}})$.*

Then for all $s \in \text{var}(W_{\mathcal{A}})$ there is a definite runtime trace τ' of C with $\tau' \models D$ and $\tau'(s) = \tau(s)$ and $\tau'(v) \geqslant \tau(v)$ for all $v \in \text{var}(W_{\mathcal{A}})$.

Proof. As $G_{W_{\mathcal{A}}}$ is acyclic it follows that $\tau_{\Delta, W_{\mathcal{A}}} = \mathbf{T}_{\Delta, W_{\mathcal{A}}}$ and therefore it suffices to prove the propositions for $\mathbf{T}_{\Delta, W_{\mathcal{A}}}$.

Let $\mathbf{T}'$ be determined by $\mathbf{T}' = \mathbf{T}_{\Delta', W_{\mathcal{A}}}$ with

$$\Delta'(v) = \begin{cases} \Delta(v) & \text{if } v \in \rho(s) \\ \infty & \text{otherwise} \end{cases}$$

Then $\forall v \in \rho(s)$, in particular for $v = s$, follows $\Delta'_v = \Delta_v$ and thus

$$\begin{aligned}\mathbf{T}'(v) &= \max\left\{ \sum_{r \in p} \Delta'_r - w(p) \;\middle|\; v \overset{p}{\rightsquigarrow} \perp_{\text{init}} \right\} + \Delta'_s \\ &= \max\left\{ \sum_{r \in p} \Delta_r - w(p) \;\middle|\; v \overset{p}{\rightsquigarrow} \perp_{\text{init}} \right\} + \Delta_s \\ &= \mathbf{T}(v)\end{aligned}$$

Moreover, if $v \notin \rho(s)$ then $\Delta'_s = \infty$ and therefore $\mathbf{T}'(v) = \infty$. Thus $\mathbf{T}'(v) \geqslant \mathbf{T}(v)$.

As by premise $\forall f_{\text{init}} : f_{\text{succ}} \notin \rho(s) \vee f_{\text{fail}} \notin \rho(s)$ holds it follows by construction of Δ' that $\forall f_{\text{init}} : \mathbf{T}'(f_{\text{succ}}) = \infty \vee \mathbf{T}'(f_{\text{fail}}) = \infty$ thus $\mathbf{T}'$ is a definite runtime trace.

Suppose $u + d \leqslant v$ is a sub-formula of D. If $v \notin \rho(s)$ then $\mathbf{T}'(v) = \infty$ and therefore $\mathbf{T}' \models u + d \leqslant v$.

If $v \in \rho(s)$ then $s \rightsquigarrow v \rightsquigarrow \perp_{\text{init}}$ is a path in $G_{W_{\mathcal{A}} \wedge D}$. However, as $u + d \leqslant v$ is a sub-formula of D it follows that $s \rightsquigarrow v \to u \rightsquigarrow \perp_{\text{init}}$ is also a path in $G_{W_{\mathcal{A}} \wedge D}$ and therefore $u \in \rho(s)$.

Accordingly, $\mathbf{T}'(u) = \tau(u)$ and $\mathbf{T}'(v) = \tau(v)$ and as $\tau \models D$ it follows that $\mathbf{T}' \models u + d \leqslant v$. □

LEMMA 9.12. *If G_{W_A} is cyclic, then the complex action $C = (W, D, H)$ is not fair.*

Proof. If G_{W_A} is cyclic, then for each variable v that is contained in a cycle holds $\lambda(v) = \infty$. Therefore $v \notin \mathrm{dom}(W)$ and thus for every runtime trace τ of C holds $\tau(v) = \infty$. □

LEMMA 9.13. *If there are variables f_{init} and g_{init} such that $g_{succ} \in \rho(f_{init})$ and $g_{fail} \in \rho(f_{init})$ it follows that the complex action $C = (W, D, H)$ is not fair.*

Proof. Suppose τ is a definite runtime trace of $C = (W, D, H)$. It thus follows from Lemma 9.9 that $\tau \models W_A$ and thus $\tau \models W_A \wedge D$.

As $g_{succ} \in \rho(f_{init})$ and $g_{fail} \in \rho(f_{init})$ there are paths p, p' in $G_{W_A \wedge D}$ with $f_{init} \overset{p}{\rightsquigarrow} g_{succ}$ and $f_{init} \overset{p'}{\rightsquigarrow} g_{fail}$. It thus follows from Theorem 9.1

$$\tau(g_{succ}) \leqslant \tau(f_{init}) + w(p) \quad \text{and} \quad \tau(g_{fail}) \leqslant \tau(f_{init}) + w(p')$$

And as τ is a definite runtime trace it further follows that $\tau(g_{succ}) = \infty$ or $\tau(g_{fail}) = \infty$ thus $\tau(f_{init}) = \infty$ and therefore C is not fair. □

THEOREM 8.1. *A complex action $C = (W, D, H)$ is not fair only if the graph G_{W_A} is cyclic or there are variables f_{init} and g_{init} with $g_{succ} \in \rho(f_{init})$ and $g_{fail} \in \rho(f_{init})$.*

Proof. Immediate consequence from Lemma 9.12 and 9.13. □

THEOREM 8.2. *A complex action $C = (W, D, H)$ with $W_A \wedge D$ consistent is not fair if the graph G_{W_A} is cyclic or there are variables f_{init} and g_{init} with $g_{succ} \in \rho(f_{init})$ and $g_{fail} \in \rho(f_{init})$.*

Proof. Suppose $\forall f_{init}, g_{init} : g_{succ} \notin \rho(f_{init}) \vee g_{fail} \notin \rho(f_{init})$ and G_{W_A} is acyclic.

As $W_A \wedge D$ is consistent in $\mathbb{Q}$ there is a σ such that $\sigma \models_{\mathbb{Q}} W_A \wedge D$. As in particular $\sigma \models_{\mathbb{Q}} W_A$ it follows from Lemma 9.10 that there is a runtime trace τ with $\tau = \sigma$.

Suppose $h_{init} \in \mathrm{var}(W_A)$. It then follows from Lemma 9.11 that there is a definite runtime trace τ' with $\tau'(h_{init}) = \tau(h_{init})$ and thus with $\tau'(h_{init}) < \infty$.

As h_{init} was arbitrarily chosen it follows that for all $h_{init} \in \mathrm{var}(W_A)$ there is a definite runtime trace τ' with $\tau'(h_{init}) < \infty$. □

9.4 PROPERTIES OF COMPLIANT ACTIONS

DEFINITION 9.7. For a temporal formula W and a runtime trace τ_{Δ, W_A} the corresponding operator T^{fin} is denoted by

$$T^{fin} = T^{T^{\emptyset}_{\Delta^{fin}, W_A^{fin}}} = T^{\mathrm{lfp}(T^{\emptyset}_{\Delta^{fin}, W_A^{fin}} \uparrow)}$$

whereby

$$\Delta^{\text{fin}}(\nu) = \begin{cases} \Delta(\nu) & \text{if } \tau(\nu) < \infty \\ 0 & \text{otherwise} \end{cases}$$

and

$$F^{\text{fin}} = F \wedge \bigwedge_{\tau(\nu)=\infty} \bot_0 + \omega_\tau \leqslant \nu$$

with $\omega_\tau = \max\{p \in \text{im}(\tau) \mid p < \infty\} + 1$ and a variable $\bot_0 \notin \text{var}(W_{\mathcal{A}})$.

DEFINITION 9.8. For the runtime trace τ of a fair complex action $C = (W, D, H)$ the corresponding τ^{fin} is denoted by

$$\tau^{\text{fin}} = \mathbf{T}^{\text{fin}}_{\Delta^{\text{fin}}, W^{\text{fin}}_{\mathcal{A}} \wedge D^\infty}$$

whereby

$$D^\infty = \bigwedge_{\substack{\tau(\nu)=\infty \\ u+d\leqslant\nu \text{ is sub-formula of } D}} u + d \leqslant \nu$$

Note that $\text{dom}(\tau^{\text{fin}}) \supseteq \text{var}(W_{\mathcal{A}})$ and $\text{im}(\tau^{\text{fin}}) \subseteq \mathbb{Q}$. Moreover, be aware that τ^{fin} is the result of three successive fixpoint iterations.

Remark. In the following, the index of the operator T^{fin} and of $T^{\text{fin}} \uparrow$ is always committed to $\Delta^{\text{fin}}, W^{\text{fin}}_{\mathcal{A}} \wedge D^\infty$. Moreover, the index of the fixpoints τ and τ^{fin} is always committed to $\Delta, W_{\mathcal{A}}$.

Thus, for the sake of readability, these indexes may be omitted in the following.

DEFINITION 9.9. For two runtime traces τ, τ' the notion $\tau \succeq \tau'$ denotes that for all $\nu \in \text{dom}(\tau)$ holds

1. $\tau(\nu) < \infty \implies \tau(\nu) = \tau'(\nu)$
2. $\tau(\nu) = \infty \implies \tau'(\nu) \geqslant \omega_\tau$

Note that if $\tau \succeq \tau'$ then $\tau(\nu) \geqslant \tau'(\nu)$ for all ν.

LEMMA 9.14. *For a temporal formula* H *and runtime traces* $\tau \succeq \tau'$ *such that for all* $u \in \text{dom}(\tau)$ *with* $\tau(u) = \infty$ *holds*

$$\tau'(u) \geqslant \omega_\tau - \min\big(\{0\} \cup \{d' \mid u' + d' \leqslant \nu' \textit{ is sub-formula of } H\}\big) \geqslant \omega_\tau$$

follows $\tau' \models H$ *implies* $\tau \models H$.

Proof. Suppose $\tau' \models u + d \leqslant \nu$. If $\tau'(u) = \tau(u)$ then

$$\tau(u) + d = \tau'(u) + d \leqslant \tau'(\nu) \leqslant \tau(\nu)$$

hence $\tau \models u + d \leqslant \nu$.

If $\tau'(u) \neq \tau(u)$ then $\tau(u) = \infty$. Moreover, as

$$d \geqslant \min\big(\{0\} \cup \{d' \mid u' + d' \leqslant v' \text{ sub-formula of } H\}\big)$$

it follows by premise that $\tau'(u) \geqslant \omega_\tau - d$ and thus

$$\omega_\tau \leqslant \tau'(u) + d \leqslant \tau'(v)$$

hence $\tau'(v) \geqslant \omega_\tau$ and thus $\tau(v) = \infty$ which implies $\tau \models u + d \leqslant v$.

Accordingly, in any case holds if $\tau' \models u + d \leqslant v$ then $\tau \models u + d \leqslant v$. Suppose $H = \bigwedge_i c_i$ is an arbitrary temporal formula and $\tau' \models H$. It thus follows for all i that $\tau' \models c_i$. Therefore $\tau \models c_i$ for all i and hence $\tau \models H$. □

LEMMA 9.15. *For a runtime trace τ of a complex action $C = (W, D, H)$ holds $\tau \succeq T^{\text{fin}} \uparrow n$.*

Proof. By induction. *Base Case* ($n = 0$): If $\tau(v) = \infty$ then $\bot_0 + \omega_\tau \leqslant v$ is a sub-formula of $W_{\mathcal{A}}^{\text{fin}}$. Without loss of generality, $v \in \text{dom}(W)$ and thus

$$\begin{aligned}
\big(T^{\text{fin}} \uparrow 0\big)(v) &= \mathbf{T}^{\emptyset}_{\Delta^{\text{fin}}, W_{\mathcal{A}}^{\text{fin}}}(v) \\
&\geqslant \max\left\{\mathbf{T}^{\emptyset}_{\Delta^{\text{fin}}, W_{\mathcal{A}}^{\text{fin}}}(\text{pre}_{W_{\mathcal{A}}^{\text{fin}} \wedge D^\infty}(v))\right\} \\
&\geqslant \mathbf{T}^{\emptyset}_{\Delta^{\text{fin}}, W_{\mathcal{A}}^{\text{fin}}}(\bot_0) + \omega_\tau \\
&\geqslant \omega_\tau
\end{aligned}$$

If $\tau(v) < \infty$ it follows from Lemma 9.8 that for all u with $v \rightsquigarrow u$ in $W_{\mathcal{A}}$ holds $\tau(u) < \infty$. Therefore, for all such u, follows $\Delta^{\text{fin}}(u) = \Delta(u)$ and there is no sub-formula of $W_{\mathcal{A}}^{\text{fin}}$ with $u + \omega_\tau \leqslant v$. Accordingly,

$$\begin{aligned}
\big(T^{\text{fin}} \uparrow 0\big)(v) &= \mathbf{T}_{\Delta^{\text{fin}}, W_{\mathcal{A}}^{\text{fin}}}(v) \\
&= \max\left\{\sum_{s \in p} \Delta_s^{\text{fin}} - w(p) \;\middle|\; v \overset{p}{\rightsquigarrow} \bot_0 \text{ in } G_{W_{\mathcal{A}}^{\text{fin}}}\right\} \\
&= \max\left\{\sum_{s \in p} \Delta_s - w(p) \;\middle|\; v \overset{p}{\rightsquigarrow} \bot_0 \text{ in } G_{W_{\mathcal{A}}}\right\} \\
&= \tau(v)
\end{aligned}$$

Inductive Case ($n \to n+1$): If $\tau(v) = \infty$ then $\bot_0 + \omega_\tau \leqslant v$ is a sub-formula of $W_{\mathcal{A}}^{\text{fin}}$. Without loss of generality, $v \in \text{dom}(W)$ and thus

$$\begin{aligned}
\big(T^{\text{fin}} \uparrow n+1\big)(v) &= \max\left\{\big(T^{\text{fin}} \uparrow n\big)(\text{pre}_{W_{\mathcal{A}}^{\text{fin}} \wedge D^\infty}(v))\right\} + \Delta_v^{\text{fin}} \\
&\geqslant \big(T^{\text{fin}} \uparrow n\big)(\bot_0) + \omega_\tau \\
&\geqslant \omega_\tau
\end{aligned}$$

If $\tau(v) < \infty$ then $\Delta^{\text{fin}}(v) = \Delta(v)$ and $\text{pre}_{W_{\mathcal{A}}^{\text{fin}} \wedge D^\infty} = \text{pre}_{W_{\mathcal{A}}}$. It furthermore follows from Lemma 9.8 that $\tau(u) < \infty$ for all $u \in \text{pre}_{W_{\mathcal{A}}}(v)$

and therefore $\tau(u) < \infty$ for all $u \in \mathrm{pre}_{W_{\mathcal{A}}^{\mathrm{fin}} \wedge D^\infty}(v)$. Thus by induction hypothesis $\big(T^{\mathrm{fin}} \uparrow n\big)(u) = \tau(u)$ for all $u \in \mathrm{pre}_{W_{\mathcal{A}}^{\mathrm{fin}} \wedge D^\infty}(v)$. and therefore

$$\begin{aligned}\big(T^{\mathrm{fin}} \uparrow n+1\big)(v) &= \max\big\{\big(T^{\mathrm{fin}} \uparrow n\big)(\mathrm{pre}_{W_{\mathcal{A}}^{\mathrm{fin}} \wedge D^\infty}(v))\big\} + \Delta_v^{\mathrm{fin}} \\ &= \max\big\{\tau(\mathrm{pre}_{W_{\mathcal{A}}^{\mathrm{fin}} \wedge D^\infty}(v))\big\} + \Delta_v^{\mathrm{fin}} \\ &= \max\big\{\tau(\mathrm{pre}_{W_{\mathcal{A}}}(v))\big\} + \Delta_v \\ &= \tau(v)\end{aligned}$$

□

LEMMA 9.16. *Suppose τ is a runtime trace of the complex action C, then $\big(T^{\mathrm{fin}} \uparrow n+1\big)(v) \geqslant \big(T^{\mathrm{fin}} \uparrow n\big)(v)$.*

Proof. By induction. *Base Case* ($n = 0$): As by definition $T^{\mathrm{fin}} \uparrow 0$ coincides with the fixpoint $\mathbf{T}^{\emptyset}_{\Delta^{\mathrm{fin}}, W_{\mathcal{A}}^{\mathrm{fin}}}$ it follows that

$$\begin{aligned}\big(T^{\mathrm{fin}} \uparrow 0\big)(v) &= \mathbf{T}^{\emptyset}_{\Delta^{\mathrm{fin}}, W_{\mathcal{A}}^{\mathrm{fin}}}(v) \\ &= \big(T^{\emptyset}_{\Delta^{\mathrm{fin}}, W_{\mathcal{A}}^{\mathrm{fin}}}(\mathbf{T}^{\emptyset}_{\Delta^{\mathrm{fin}}, W_{\mathcal{A}}^{\mathrm{fin}}})\big)(v) \\ &= \max\big\{\mathbf{T}^{\emptyset}_{\Delta^{\mathrm{fin}}, W_{\mathcal{A}}^{\mathrm{fin}}}(\mathrm{pre}_{\Delta^{\mathrm{fin}}, W_{\mathcal{A}}^{\mathrm{fin}}}(v))\big\} + \Delta_v^{\mathrm{fin}} \\ &= \mathbf{T}^{\emptyset}_{\Delta^{\mathrm{fin}}, W_{\mathcal{A}}^{\mathrm{fin}}}(u) + d + \Delta_v^{\mathrm{fin}} \\ &= \big(T^{\mathrm{fin}} \uparrow 0\big)(u) + d + \Delta_v^{\mathrm{fin}}\end{aligned}$$

for a sub-formula $u + d \leqslant v$ of $W_{\mathcal{A}}^{\mathrm{fin}}$. Therefore

$$\begin{aligned}\big(T^{\mathrm{fin}} \uparrow 1\big)(v) &= \max\big\{\big(T^{\mathrm{fin}} \uparrow 0\big)(\mathrm{pre}_{W_{\mathcal{A}}^{\mathrm{fin}} \wedge D^\infty}(v))\big\} + \Delta_v^{\mathrm{fin}} \\ &\geqslant \big(T^{\mathrm{fin}} \uparrow 0\big)(u) + d + \Delta_v^{\mathrm{fin}} \\ &= \big(T^{\mathrm{fin}} \uparrow 0\big)(v)\end{aligned}$$

Inductive Case ($n \to n+1$):

$$\begin{aligned}\big(T^{\mathrm{fin}} \uparrow n+1\big)(v) &= \max\big\{\big(T^{\mathrm{fin}} \uparrow n\big)(\mathrm{pre}_{W_{\mathcal{A}}^{\mathrm{fin}} \wedge D^\infty}(v))\big\} + \Delta_v^{\mathrm{fin}} \\ &\geqslant \max\big\{\big(T^{\mathrm{fin}} \uparrow n-1\big)(\mathrm{pre}_{W_{\mathcal{A}}^{\mathrm{fin}} \wedge D^\infty}(v))\big\} + \Delta_v^{\mathrm{fin}} \\ &= \big(T^{\mathrm{fin}} \uparrow n\big)(v)\end{aligned}$$

□

LEMMA 9.17. *Suppose τ is a runtime trace of the fair complex action $C = (W, D, H)$ and $W_{\mathcal{A}} \wedge D$ is consistent in $\mathbb{Q}$. Then the fixpoint τ^{fin} exists and $\tau^{\mathrm{fin}} = T \uparrow n$ for some $n \in \mathbb{N}$.*

Proof. For a v with $\big(T^{\mathrm{fin}} \uparrow i\big)(v) > \big(T^{\mathrm{fin}} \uparrow i-1\big)(v)$ follows that there is a sub-formula $u + d \leqslant v$ of $W_{\mathcal{A}}^{\mathrm{fin}} \wedge D^\infty$ with

$$\big(T^{\mathrm{fin}} \uparrow i\big)(v) = \big(T^{\mathrm{fin}} \uparrow i-1\big)(u) + d + \Delta_v^{\mathrm{fin}}$$

and furthermore $\left(T^{\text{fin}} \uparrow i-1\right)(u) > \left(T^{\text{fin}} \uparrow i-2\right)(u)$.

Suppose there is no n such that $\tau^{\text{fin}} = T^{\text{fin}} \uparrow n$, then for all n there is a v such that $\left(T^{\text{fin}} \uparrow n+1\right)(v) \neq \left(T^{\text{fin}} \uparrow n\right)(v)$. It then follows from Lemma 9.16 that $\left(T^{\text{fin}} \uparrow n+1\right)(v) > \left(T^{\text{fin}} \uparrow n\right)(v)$. Accordingly, for an n larger than the longest acyclic path in $G_{W_{\mathcal{A}}^{\text{fin}} \wedge D^{\infty}}$ there is a v such that $\left(T^{\text{fin}} \uparrow n+1\right)(v) > \left(T^{\text{fin}} \uparrow n\right)(v)$. It thus follows that there are $v = v_{n+1}, \ldots, v_0$ such that for $0 < i \leqslant n+1$ there is a sub-formula $v_{i-1} + d_{i-1} \leqslant v_i$ and

$$\left(T^{\text{fin}} \uparrow i\right)(v_i) = \left(T^{\text{fin}} \uparrow i-1\right)(v_{i-1}) + d_{i-1} + \Delta_{v_i}^{\text{fin}}$$

As n is larger than the longest path it furthermore follows that there is a $0 \leqslant k < n+1$ such that $v_k = v_{n+1}$ and thus that the path $p = \langle v_{n+1}, \ldots, v_k \rangle$ is a cycle in $G_{W_{\mathcal{A}}^{\text{fin}} \wedge D^{\infty}}$. However, it follows from Definition 9.7 and Definition 9.8 that $\bot_0$ cannot be contained in any cycle and therefore it follows that p is actually a cycle in $G_{W_{\mathcal{A}} \wedge D^{\infty}}$.

Moreover

$$\begin{aligned}
\left(T^{\text{fin}} \uparrow n+1\right)(v) &= \left(T^{\text{fin}} \uparrow n\right)(v_n) + d_n \\
&= \left(T^{\text{fin}} \uparrow n-1\right)(v_{n-1}) + d_{n-1} + d_n \\
&\vdots \\
&= \left(T^{\text{fin}} \uparrow k\right)(v_k) + \sum_{i=k}^{n} d_i = \left(T^{\text{fin}} \uparrow k\right)(v) + \sum_{i=k}^{n} d_i
\end{aligned}$$

And as Lemma 9.16 implies $\left(T^{\text{fin}} \uparrow n\right)(v) \geqslant \left(T^{\text{fin}} \uparrow k\right)(v)$ and furthermore $\left(T^{\text{fin}} \uparrow n+1\right)(v) > \left(T^{\text{fin}} \uparrow n\right)(v)$ it follows

$$0 > \left(T^{\text{fin}} \uparrow k\right)(v) - \left(T^{\text{fin}} \uparrow n+1\right)(v) = -\sum_{i=k}^{n} d_i = w(p)$$

Therefore p denotes a negative cycle in $G_{W_{\mathcal{A}} \wedge D^{\infty}}$ and hence also in $G_{W_{\mathcal{A}} \wedge D}$. And thus it follows from Theorem 9.2 that $W_{\mathcal{A}} \wedge F$ is inconsistent in $\mathbb{Q}$. □

LEMMA 9.18. *For a fair complex action* $C = (W, D, H)$ *with* $W_{\mathcal{A}} \wedge D$ *consistent follows* $\tau \succeq \tau^{\text{fin}}$.

Proof. Immediate consequence from Lemma 9.17 and Lemma 9.15. □

LEMMA 9.19. *For the runtime trace* τ *of a fair complex action* $C = (W, D, H)$ *with* $W_{\mathcal{A}} \wedge D$ *consistent follows* $\tau^{\text{fin}} \models_{\mathbb{Q}} W_{\mathcal{A}} \wedge D$.

Proof. Let $u + d \leqslant v$ determine sub-formula of $W_{\mathcal{A}} \wedge D$. If $u + d \leqslant v$ is furthermore sub-formula of $W_{\mathcal{A}}^{\text{fin}} \wedge D^{\infty}$ it follows for the fixpoint τ^{fin} that

$$\begin{aligned}
\tau^{\text{fin}}(v) &= T^{\text{fin}}\left(\tau^{\text{fin}}\right)(v) \\
&= \max\left\{\tau^{\text{fin}}\left(\text{pre}_{W_{\mathcal{A}}^{\text{fin}} \wedge D^{\infty}}(v)\right)\right\} + \Delta_v^{\text{fin}} \\
&\geqslant \tau^{\text{fin}}(u) + d
\end{aligned}$$

thus $\tau^{\text{fin}} \models u + d \leqslant v$.

Otherwise $u + d \leqslant v$ is a sub-formulas of D but not of D^{∞} and hence $\tau(v) < \infty$. As $\tau \models W_{\mathcal{A}} \wedge D$ it further follows $\tau \models u + d \leqslant v$ and therefore $\tau(u) < \infty$. Thus $\tau(v) = \tau^{\text{fin}}(v)$ and $\tau(u) = \tau^{\text{fin}}(u)$ and therefore $\tau^{\text{fin}} \models u + d \leqslant v$.

Accordingly, as $\text{im}(\tau^{\text{fin}}) \subseteq \mathbb{Q}$ and $u + d \leqslant v$ has been arbitrarily chosen it follows $\tau^{\text{fin}} \models_{\mathbb{Q}} W_{\mathcal{A}}^{\text{fin}} \wedge D$. And by definition of $W_{\mathcal{A}}^{\text{fin}}$ follows $W_{\mathcal{A}}^{\text{fin}} \models_{\mathbb{Q}} W_{\mathcal{A}}$ and therefore $\tau^{\text{fin}} \models_{\mathbb{Q}} W_{\mathcal{A}} \wedge D$. □

THEOREM 8.3. *For a fair complex action* $C = (W, D, H)$ *with* $W_{\mathcal{A}} \wedge D$ *consistent holds* $W_{\mathcal{A}} \wedge D \models_{\mathbb{Q}} H$ *only if for all definite runtime traces* τ *of* C *holds* $\tau \models_{\mathbb{P}} H$.

Proof. Suppose τ is a definite runtime trace. So far, the value $\Delta_{\perp_0}$ of the variable $\perp_0$ introduced in Definition 9.7 has not been constrained and thus we can determine

$$\Delta_{\perp_0} = -\min(\{0\} \cup \{d \mid u + d \leqslant v \text{ sub-formula of } H\})$$

From Lemma 9.19 then follows $\tau^{\text{fin}} \models_{\mathbb{Q}} W_{\mathcal{A}}^{\text{fin}} \wedge D$ and thus $\tau^{\text{fin}} \models_{\mathbb{Q}} H$. In addition, it follows from Lemma 9.18 that $\tau \succeq \tau^{\text{fin}}$. Therefore, it follows from $\tau^{\text{fin}} \models_{\mathbb{Q}} W_{\mathcal{A}}^{\text{fin}}$ that $\tau^{\text{fin}}(v) \geqslant \omega_{\tau} + \Delta_{\perp_0}$ for all v with $\tau(v) = \infty$ and therefore the premises of Lemma 9.14 are satisfied and it thus follows $\tau \models_{\mathbb{P}} H$. □

THEOREM 8.4. *For a fair complex action* $C = (W, D, H)$ *with* $W_{\mathcal{A}} \wedge D$ *consistent holds* $W_{\mathcal{A}} \wedge D \models_{\mathbb{Q}} H$ *if for all definite runtime traces* τ *of* C *holds* $\tau \models_{\mathbb{P}} H$.

Proof. If $W_{\mathcal{A}} \wedge D \not\models_{\mathbb{Q}} H$ there is a variable assignment σ and a sub-formula $u + d \leqslant v$ of H such that $\sigma \models_{\mathbb{Q}} W_{\mathcal{A}} \wedge D$ and $\sigma \not\models_{\mathbb{Q}} u + d \leqslant v$.

As $\sigma \models_{\mathbb{Q}} W_{\mathcal{A}}$ it follows from Lemma 9.10 that there is a runtime trace τ with $\sigma = \tau$. It further follows from Lemma 9.11 that there is a definite runtime trace τ' with $\tau'(v) = \tau(v)$ and $\tau'(s) \geqslant \tau(s)$ for all $s \in \text{var}(W_{\mathcal{A}})$. Therefore

$$\tau'(u) + d \geqslant \tau(u) + d > \tau(v) = \tau'(v)$$

and thus $\tau' \not\models_{\mathbb{P}} u + d \leqslant v$. Accordingly, τ' is a definite runtime trace with $\tau' \not\models_{\mathbb{P}} H$. □

Part IV

OPERATIONAL SEMANTICS AND IMPLEMENTATION

10 OPERATIONAL SEMANTICS OF DURA$_C$

The evaluation of Dura is based on the Temporal Stream Algebra (TSA) [BB12a], an extension of the relational algebra adapting it to streams. To this end, Dura queries and rules are represented by means of TSA algebra expressions build from operators as they are known from the relational algebra, such as, cross product, selection, and grouping.

However, to reduce the complexity of the translation Dura is not directly translated to TSA. Instead, we identify a language core of Dura denoted Dura$_C$ (pronounced "Dura Core") that is expressive enough to cover all aspects of Dura and elaborate a translation of Dura$_C$ to TSA. Subsequently, we establish a translation that rewrites Dura queries to Dura$_C$ queries and thus indirectly obtain an operational semantics of (full) Dura.

10.1 PRELIMINARIES AND INFORMAL INTRODUCTION

Before we describe TSA in more detail and elaborate the translation to TSA, we clarify some basic notions and introduce the sublanguage Dura$_C$. Moreover, the basic ideas of the translation to TSA are illustrated by means of simple but distinctive example queries.

10.1.1 *Event Streams*

Event streams resemble relations known from databases. However, in contrast to conventional relations, streams are unbounded and the tuples or events of the stream arrive online as time advances. As a consequence, for each point in time only a prefix of the entire stream, containing the already obtained events, is known and the conceptually unbounded stream is never obtained completely.

In the following, some of the notions from [AHV95] conceived for relations are transferred to streams. In particular, the named perspective of relational algebra is adopted where tuples are viewed as functions mapping attributes to values and the attributes of a stream R are denoted $\text{sort}(R)$. Accordingly, a tuple of R is represented by a total mapping r with $\text{dom}(r) = \text{sort}(R)$.

10.1.2 *Representation of Dura Events*

To be applicable to streams, Dura events need to be represented by means of flat tuples. However, the schema of events resembles records with an inherent structure whereas the schema of streams merely

correspond to a collection of attribute names. Accordingly, there is no direct mapping between events and tuples.

Listing 10.1: Schema of `temp` Events

```
temp{
  id{identifier}, rt{ begin{timestamp}, end{timestamp} },
  area{long}, value{double}, sensor-id{long}
}
```

Nevertheless, recall that the schema of events is represented by means of Xcerpt terms whereby references to nodes as well as equally labeled or ordered siblings are not permitted. As a consequence, the payload of events corresponds to a tree structure and the location of each atomic value is uniquely described by the labels of the path form the root to the leaf containing the respective value. Accordingly, the schema of events can be mapped to the schema of streams by interpreting the paths to all leafs of the event schema as attributes of a stream schema. In this way, events can be translated to flat tuples whereby the structure of the event is preserved in the names of the attributes.

EXAMPLE The schema of `temp` events from Listing 10.1 translates to the stream T with

$$\text{sort}(T) = \{area, id, rt.begin, rt.end, sensor\text{-}id, value\}$$

Note how the attribute `rt{ begin{timestamp}, end{timestamp} }` of the `temp` event translates to the two attributes *rt.begin* and *rt.end* of the stream T. Accordingly, the event

```
temp{ area{31}, value{12.7}, sensor-id{8191}, ... }
```

corresponds to a tuple $t \in T$ represented by the function

$$t = \{area \mapsto 31,\ value \mapsto 12.7,\ sensor\text{-}id \mapsto 8191,\ \ldots\}$$

Be aware that for the sake of conciseness and readability, the label of the composite attribute `reception-time` is abbreviated with `rt` in this and all following examples.

10.1.3 *A Generalization of Relational Algebra for Streams*

TSA proposed by Brodt and Bry [BB12a] is an extension of the relational algebra adapting it to streams, in particular with respect to the incremental evaluation of expressions, which is inevitable due to the unbounded nature of streams.

TSA provides algebra operators, such as, projection (π), selection (σ), rename (ρ), cross product ($\times$), union ($\cup$), and set difference ($\setminus$),

which are closely related to the respective operators of the relational algebra [AHV95]. Composite algebra expressions, in turn representing streams, are built by applying algebra operators to streams.

However, as at runtime merely a prefix of the entire stream is available, in contrast to the relational algebra, there are syntactically correct algebra expressions that can conceptually not be evaluated in an incremental fashion and hence cannot be evaluated on streams. Such kind of expressions usually involve non-monotonic operators, such as, grouping and aggregation, and are recognized by the static analysis described in [BB12a], which determines whether and how algebra expressions can be incrementally evaluated.

REPRESENTATION OF STREAMS IN TSA Streams, as they are introduced in [BB12a], are associated with a sophisticated stream schema, which includes, in addition to the attributes of the stream, further information related to the static analysis and the incremental evaluation of TSA expressions. However, the details on the proposed static analysis and incremental evaluation are out of the scope of this thesis, they are thoroughly elaborated in [BB12a], though.

Accordingly, the stream schema is usually omitted in the following as the additional information is not further relevant for the translation of Dura to TSA. Merely the translation of event schemata described in Section 10.4.1 specifies stream schemata as they are introduced in the original work on TSA.

RELATIONS BETWEEN DURA AND TSA TSA and Dura have been elaborated in close collaboration within the EMILI project. Accordingly, both formalisms have been designed to be suited to each other. As a consequence, some of the considerable features of Dura, in particular the support of user defined times and multiple independent time lines and the versatile grouping and negation capabilities, are actually inherited from TSA, which forms the basis for the evaluation of Dura queries.

To discriminate between Dura and TSA the notions event, tuple, query, and (algebra) expression are strictly separated in the following. The notions event and query are only used when referring to Dura whereas the notion tuple and expression is only used when referring to TSA. Nevertheless, the notions are closely related as Dura events correspond to TSA tuples and Dura queries correspond to TSA expressions. In addition, arithmetic expressions from Dura and TSA are referred to as terms to distinguish them from algebra expressions. Note, however, that the notions may be named differently elsewhere, in particular in [BB12a].

10.1.4 *The Sub-language Dura$_C$*

Informally speaking, Dura$_C$ is the subset of Dura that is limited to declarative event queries and deductive rules. In this way, Dura$_C$ is limited to constructs that can be translated more or less directly. In contrast, stateful objects, complex actions, and reactive rules cannot be easily expressed in TSA and are thus not included in Dura$_C$.

More formally, Dura$_C$ queries are inductively defined as follows.

1. each atomic event query *event* `e: t` is a Dura$_C$ query
2. if `q` is a Dura$_C$ query, then
   `q` **`where`** `c`, `q` **`let`** `l`, and `q` **`group by`** `g` **`aggregate`** `a` are Dura$_C$ queries
3. if $q_1, \ldots, q_k$ are Dura$_C$ queries, then
   `and`$\{q_1, \ldots, q_k\}$, **`or`**$\{q_1, \ldots, q_k\}$, and **`not`**$\{q_1\}$ are Dura$_C$ queries.

Moreover, if `q` is a Dura$_C$ query, then `DETECT t ON q END` is a Dura$_C$ rule.

10.1.5 *Basic Ideas of the Translation*

In the following, queries from Chapter 5 are used to illustrate the basic ideas of the translation of Dura queries to TSA expressions by means of examples. The generic translation from Dura to TSA are subject to Section 10.4.

The queries are based on `temp`, `smoke`, `sensor-msg`, `temp-value`, and `alarm` events which are represented by the streams T, S, M, V, and A, respectively. The corresponding attributes of those streams are listed in Table 10.1.

$$
\begin{aligned}
\text{sort}(\mathsf{T}) &= \{\mathit{area}, \mathit{id}, \mathit{rt.begin}, \mathit{rt.end}, \mathit{sensor\text{-}id}, \mathit{value}\} \\
\text{sort}(\mathsf{S}) &= \{\mathit{area}, \mathit{conc}, \mathit{id}, \mathit{rt.begin}, \mathit{rt.end}\} \\
\text{sort}(\mathsf{M}) &= \{\mathit{area}, \mathit{id}, \mathit{rt.begin}, \mathit{rt.end}, \mathit{sensor}, \mathit{type}, \mathit{value}\} \\
\text{sort}(\mathsf{V}) &= \{\mathit{aid}, \mathit{id}, \mathit{rt.begin}, \mathit{rt.end}, \mathit{sid}, \mathit{val}\} \\
\text{sort}(\mathsf{A}) &= \{\mathit{area}, \mathit{id}, \mathit{rt.begin}, \mathit{rt.end}\}
\end{aligned}
$$

Table 10.1: Stream Attributes

ATOMIC QUERIES AND QUERY SUPPLEMENTS Atomic queries are translated by mapping the type of the queried event to the corresponding stream. In addition, a prefix coinciding with the event identifier of the query is added to the attributes of the stream by means of a *rename* (ρ) operator. In this way, the attributes of each stream are unambiguously determined when multiple streams are considered, eg, for the translation of composite queries, and thus name conflicts between equally named attributes of different streams are avoided.

Listing 10.2: An Atomic Query with Supplement

```
event e: temp{ area{var A}, value{var T} }
  where { var T > 400 }
  let { var Min = ceil(end(e), 1min) }
```

Moreover, the `where` part of queries is translated to *selections* (σ) by canonically converting the formulas specified in the `where` part to composite TSA formulas. Thereby, references are replaced with the respective attributes they are referring to. For instance, the atomic formula `var A > 400` is translated to $e.area > 400$.

In addition, variables introduced by `let` constructs are translated to designated attributes that are added to the stream by means of *embeddings* (ι).[1] Eg, the `let` part from above is translated to an embedding which adds the attribute *.var.Min* to the stream, notice the leading period, whose value is determined by the term $\text{ceil}(e.rt.end, 60)$. Thereby, *.var* is a prefix that is not valid for variable names in Dura and thus name conflicts between existing attributes and attributes caused by the translation of `let` parts are avoided.

Accordingly, the query from Listing 10.2 translates to the following TSA expression.

$$\begin{aligned}&\iota[.var.Min \leftarrow \text{ceil}(e.rt.end, 60)\big(\\ &\quad\sigma[e.value > 400]\big(\\ &\qquad\rho[* \rightarrow e.*]\big(\mathsf{T}\big)\big)\big)\end{aligned}$$

CONJUNCTIVE QUERIES Conjunctive queries are translated to TSA by a combination of *cross products* ($\times$) and *selections* (σ).

Listing 10.3: A Conjunctive Query

```
and{
  event e: temp{ area{var A}, value{var T} },
  event f: smoke{ area{var A}, conc{var C} }
} where { {e,f} within 1min, var T > 100, var C > 0.1 }
```

To translate a conjunctive query, all (unnegated) sub-queries are translated to the corresponding TSA expression and subsequently connected by means of a cross product. Moreover, the join conditions contained in the supplementary `where` part are translated by means of a selection. Thereby, implied join conditions, caused by the usage of equally named variables, need to be explicitly added to the condition of the selection.

For the given conjunctive query, the atomic formula $e.area = f.area$, caused by the implied join over the variable `A`, is added to the condi-

1 Embeddings correspond to extended projections of the relational algebra that do not omit values.

tion of the selection. Note that the reception time of events is explicitly contained in the payload of events and that thus even temporal conditions between events are translated to conditions contained in selections.

$$\begin{aligned}&\sigma[e.area = f.area \wedge e.value > 100 \wedge f.conc > 0.1 \wedge \\ &\quad \text{greatest}(e.rt.end, f.rt.end) - \text{least}(e.rt.begin, f.rt.begin) \leqslant 60]\big(\\ &\quad\quad \times\big(\rho[* \to e.*](T), \\ &\quad\quad\quad \rho[* \to f.*](S)\big)\big)\end{aligned}$$

NEGATED SUB-QUERIES Negated sub-queries contained in a conjunctive query are translated by means of *anti-semi-joins* ($\overline{\ltimes}$).

Listing 10.4: A Query with Negation

```
and{
  event e: temp{ sensor-id{var Id} },
  not event f: temp{ sensor-id{var Id} }
} where { f during from-end(e, 1min) }
```

In contrast to a conventional join $R \bowtie S$, which combines tuples from R with matching tuples from S, that is, which only preserves tuples from R that have a join partner in S, an anti-semi-join $R \mathbin{\overline{\ltimes}} S$ *filters* all tuples from R that have a join partner in S. Note that therefore $\text{sort}(R \mathbin{\overline{\ltimes}} S) = \text{sort}(R)$ and that hence the join condition needs to be specified in conjunction with the anti-semi-join operator and cannot be applied in a supplemental selection afterwards, as it is realized for the translation of conjunctive queries without negations.

As a consequence, a conjunctive query with negated sub-queries is translated by independently translating the unnegated and negated part of the query to TSA expressions and by connecting them by means of an anti-semi-join operator. Thereby, join conditions contained in the supplemental `where` part referring to the negated subquery, are specified in association with the anti-semi-join operator.

$$\begin{aligned}&\overline{\ltimes}[e.sensor\text{-}id = f.sensor\text{-}id \wedge \\ &\quad e.rt.end < f.rt.begin \wedge f.rt.end < e.rt.end + 60]\big(\\ &\quad\quad \rho[* \to e.*](T), \\ &\quad\quad \rho[* \to f.*](T)\big)\end{aligned}$$

GROUPING AND AGGREGATION Grouping and aggregation from Dura is translated by means of *groupings* (γ). Thereby, the first argument of γ corresponds to the grouping attributes from the `group by` part whereas the second argument introduces new attributes based on terms applying the aggregation functions that are specified in the `aggregate` part of the query.

Listing 10.5: A Query with Grouping

```
event e: temp{ area{var A}, value{var T} }
  let { var Min = ceil(end(e), 1min) }
  group by { var Min, var A } aggregate { var Tavg = avg(T) }
```

If event identifiers are specified in the `group by` part, they are replaced by all basic attributes of the stream corresponding to the event that is referenced by the identifier.

$$\gamma[.var.Min, e.area][.var.Tavg \leftarrow \text{avg}(e.value)]\big(\\ \iota[.var.Min \leftarrow \text{ceil}(e.rt.end, 60)\big(\\ \rho[* \rightarrow e.*]\big(\text{T}\big)\big)\big)$$

DISJUNCTIVE QUERIES Disjunctions are translated by means of *unions* ($\cup$) in TSA. However, the union operator can only be applied to streams with identical attributes. Therefore, a *projection* (π) is used to eliminate attributes that are not common to both streams.

Listing 10.6: A Disjunctive Query

```
or{
  event e: sensor-msg{ area{var A}, value{var Tkel} }
    let { var Tcel = var Tkel - 273.15 }

  event e: temp-value{ aid{var A}, val{var Tfah} }
    let { var Tcel = (var Tfah - 32)/1.8 }
}
```

As the projection also eliminates attributes that are referenced by variables, all variables that have positive polarity in the disjunction are explicitly added to the stream by means of embeddings. For instance, the value of the variable `A` is in both streams copied to the attribute *.var.A* and thus remains in the stream determined by the union operator whereas the primal attributes *e.area* and *e.aid* are eliminated by the projections.

$$\cup\big(\pi[e.id, e.rt.begin, e.rt.end, .var.A, .var.Tcel]\big(\\ \iota[.var.A \leftarrow e.area,\ .var.Tcel \leftarrow e.value - 273.15]\big(\\ \rho[* \rightarrow e.*]\big(\text{M}\big)\big)\big),\\ \pi[e.id, e.rt.begin, e.rt.end, .var.A, .var.Tcel]\big(\\ \iota[.var.A \leftarrow e.aid,\ .var.Tcel \leftarrow (e.val - 32)/1.8]\big(\\ \rho[* \rightarrow e.*]\big(\text{V}\big)\big)\big)\big)$$

EVENT CONSTRUCTION Event construction from rule heads is realized in TSA by means of *embeddings* (ι) and *projections* (π).

Listing 10.7: A Declarative Rule

```
DETECT
  alarm{ area{var A} }
ON
  and{
    event e: temp{ area{var A}, value{var T} },
    event f: smoke{ area{var A}, conc{var C} }
  } where { {e,f} within 1min, var T > 100, var C > 0.1 }
END
```

The attributes of derived tuples need to coincide with the attributes of the stream corresponding to the event type specified in the rule head. Accordingly, the schema of the TSA expression needs to be adapted appropriately. To this end, redundant attributes are eliminated by means of projections and terms determining the value of attributes in the query head are translated by means of embeddings. Moreover, the implicit attributes for the key and the reception time of the event that are not explicitly determined in the rule head are added to the stream by means of further embeddings.

Note that because of the implied join caused by the variables `A` in the body of the rule, either of both attributes that are referenced by `A` can be used in the rule head as a translation of `A`. Accordingly, in the rule head, *e.area* as well as *f.area* are valid substitutes for *var* `A`.

$$
\begin{aligned}
A \leftarrow \pi[&\mathit{area}, \mathit{id}, \mathit{rt.begin}, \mathit{rt.end}]\big(\\
&\iota[\mathit{area} \leftarrow \mathit{e.area}]\big(\\
&\quad \iota[\mathit{id} \leftarrow \text{sequence.next}(),\\
&\quad\ \mathit{rt.begin} \leftarrow \text{least}(\mathit{e.rt.begin}, \mathit{f.rt.begin}),\\
&\quad\ \mathit{rt.end} \leftarrow \text{greatest}(\mathit{e.rt.end}, \mathit{f.rt.end})]\big(\\
&\quad\quad \ldots)))
\end{aligned}
$$

10.2 A GENTLE INTRODUCTION TO TSA

This section contains an informal introduction of TSA that is derived from [BB12a]. Thereby, many details of TSA, in particular those related to the stream schema and the evaluation of TSA expressions, are omitted or presented in a simplified manner. For a comprehensive and precise description of TSA refer to [BB12a].

10.2.1 *Basic Algebra Operators*

TSA provides the following seven basic operators which closely resemble the operators known from relational algebra [AHV95]. However, note that the operators include a cross product but no join and that the definition of new attributes is separated from the projection

operator. This design decision has been made in [BB12a] to obtain orthogonal operators that are easier to analyze and better suited for optimizations of query plans. Nevertheless, a combination of the TSA operators can be used to obtain the functionality of joins and other conventional operators known from relational algebra.

Note that the following representation of streams and operators deviates from the one in [BB12a]. In particular, the sophisticated schema of operators determining composite streams is omitted as it is not relevant for the translation of Dura queries and rules.

SELECTION (σ) A selection filters tuples of a stream that do not meet a given condition. Suppose R is a stream and C is a formula solely referring to the attributes of R, then

$$\sigma[C](R) = \{r \mid r \in R \text{ and } r \text{ satisfies } C\}$$

EMBED (ι) An embedding determines values for new attributes according to a term t, which is identified by means of a function f_t in the following.

Suppose R is a stream, $a_i \in \text{sort}(R)$, and $a' \notin \text{sort}(R)$, then

$$\begin{aligned}\iota[a' \leftarrow f_t(a_1,\ldots,a_k)](R) = \{r' \mid\ & \text{dom}(r') = \text{sort}(R) \uplus \{a'\} \text{ and} \\ & \text{exists } r \in R \text{ such that} \\ & r'(a) = r(a) \text{ for } a \in \text{sort}(R) \text{ and} \\ & r'(a') = f_t(r(a_1),\ldots,r(a_k))\}\end{aligned}$$

PROJECTION (π) A projection omits attributes from a stream R. Note that in TSA, in contrast to the classical projection from relational algebra, new attributes can only be introduced by means of embeddings.

Suppose R is a stream and $A \subseteq \text{sort}(R)$, then

$$\begin{aligned}\pi[A](R) = \{r' \mid\ & \text{dom}(r') = A \text{ and} \\ & \text{exists } r \in R \text{ such that } r'(a) = r(a) \text{ for } a \in A\}\end{aligned}$$

GROUPING (γ) A grouping aggregates the values of multiple tuples. To this end, the tuples of R are partitioned into groups so that all tuples of each group have the identical values with respect to the set of grouping attributes A. Subsequently, an aggregation function f is applied to the tuples of each group, determining a single value for the attribute a' for each group.

Suppose R is a stream, $A \subseteq \text{sort}(R)$, $a \in \text{sort}(R)$, $a' \notin \text{sort}(R)$, and $f \in \{\min, \max, \text{avg}, \text{sum}, \text{count}\}$, then

$$\gamma[A][a' \leftarrow f(a)](R) = \Big\{ r' \;\Big|\; \begin{array}{l} \text{dom}(r') = A \uplus \{a'\} \text{ and} \\ \quad \text{exists } r \in R \text{ such that} \\ \quad r'(a) = r(a) \text{ for } a \in A \text{ and} \\ \quad r'(a') = f\big(\langle s(a) \rangle_{s \in R_r}\big) \end{array} \Big\}$$

whereby $R_r = \{t \in R \mid t(a) = r(a) \text{ for } a \in A\}$ and $\langle \cdot \rangle_\cdot$ denotes a duplicate preserving sequence.

CROSS PRODUCT ($\times$) A cross product between R and S pairwise combines all tuples of R with all tuples of S.

Suppose R and S are streams with $\text{sort}(R) \cap \text{sort}(S) = \emptyset$, then

$$R \times S = \Big\{ r' \;\Big|\; \begin{array}{l} \text{dom}(r') = \text{sort}(R) \uplus \text{sort}(S) \text{ and} \\ \quad \text{exists } r \in R \text{ and } s \in S \text{ such that} \\ \quad r'(a) = r(a) \text{ for } a \in \text{sort}(R) \text{ and} \\ \quad r'(a) = s(a) \text{ for } a \in \text{sort}(S) \end{array} \Big\}$$

UNION ($\cup$) The union of two streams R and S merges all tuples of R and S into a single stream.

Suppose R and S are streams with $\text{sort}(R) = \text{sort}(S)$, then

$$R \cup S = \Big\{ r \;\Big|\; r \in R \text{ or } r \in S \Big\}$$

SET DIFFERENCE ($\setminus$) The set difference of two streams R and S omits all tuples from R that are also contained in S.

Suppose R and S are streams with $\text{sort}(R) = \text{sort}(S)$, then

$$R \setminus S = \Big\{ r \in R \;\Big|\; r \notin S \Big\}$$

SYNTACTIC VARIATIONS It is furthermore legitimate that embeddings and groupings specify more than one attribute to be added to the stream. Moreover, as cross product and union are associative, parentheses around multiple cross products and unions can be omitted.

10.2.2 *Composite Algebra Operators*

For convenience, the following two additional composite TSA operators have been introduced which are based on the operators introduced above and which are used as syntactic sugar in the following.

RENAME (ρ) A renaming adapts the names of attributes. It can be obtained by applying an embedding and projection to the stream.

Suppose R is a stream, $a \in \text{sort}(R)$, and $a' \notin \text{sort}(R)$, then

$$\rho[a \to a'](R) = \pi[A]\big(\iota[a' \leftarrow a](R)\big)$$

whereby $A = \big(\text{sort}(R) \setminus \{a\}\big) \uplus \{a'\}$.

PREFIX (ρ) A slight variation of the rename operator can furthermore be used to add a prefix to the attributes of a stream.

Suppose R is a stream with $\text{sort}(R) = \{a_1, \ldots, a_k\}$, then

$$\rho[* \to p.*](R) = \rho[a_1 \to p.a_1, \ldots, a_k \to p.a_k](R)$$

ANTI-SEMI-JOIN ($\overline{\ltimes}$) An anti-semi-join between R and S filters all tuples of R that have a join partner in S according to a join condition C. Be aware that consequently $\text{sort}(R \mathbin{\overline{\ltimes}}[C]\ S) = \text{sort}(R)$.

Suppose R and S are streams and C is a formula referring solely to the attributes of R and S, then

$$\begin{aligned} &R \mathbin{\overline{\ltimes}}[C]\ S = \\ &\quad R \setminus \pi[\text{sort}(R)]\big(\\ &\qquad \sigma[C](R \times S)\big) \end{aligned}$$

10.3 NORMALIZATION OF QUERIES

Prior to their translation to TSA, Dura queries are normalized in a preprocessing step to facilitate their translation.

Dura provides many high-level concepts, eg, pattern matching with variables, that are not available in TSA and need to be expressed with equivalent but more basic means. In addition, even basic operators of Dura, eg, the unary negation `not`, do not have a direct correspondence in TSA so that structural modifications of the queries are required to facilitate their translation to TSA expressions. Moreover, the implied conditions that are not explicitly specified in Dura, eg, joins over equally named variables specified in different locations of the query, need to be made explicit.

The following translations are intended to be applied in the order they are presented in this section. Thereby, valid Dura queries and rules are transformed to semantically equivalent but more basic Dura queries and rules. Each translation is successively applied where applicable starting at the inner most operator, formula, and function, respectively. In addition, the translations expect that queries are weakly range restricted as described in Section 5.2.1 and that furthermore the conditions on progressing attributes described in Section 5.2.5 are satisfied.

10.3.1 *Eliminating Literals in Query Terms*

Literals, that is, the textual representation of values in Dura, specified in query terms are intended to constrain the values of events matching the query. However, the concept of pattern matching is not available in TSA. Nevertheless, the same effect is achieved by adding a condition to the corresponding atomic query specifying the path to the literal and constraining the referenced value accordingly, as it is illustrated in Figure 10.1.

```
event e: temp{ sensor-id{496} }
```

```
event e: temp{}
 where { e.sensor-id = 496 }
```

Figure 10.1: Eliminating Literals

Accordingly, after this transformation has been applied, conditions on the values of attributes specified by means of patterns containing literals have been replaced by equivalent conditions in the `where` part of the corresponding event queries.

10.3.2 *Eliminating Temporal Relations and Functions*

Interval functions are not available in TSA but can be translated to composite terms by successively replacing the inner most composite function by the corresponding definition summarized in Table 10.2.

```
where {
  f during from-end(e, 1min),
  {f,g} apart-by 2min
}
```

```
where {
  e.rt.end < f.rt.begin,
  f.rt.end < e.rt.end + 1min,

  or{
    f.rt.begin - e.rt.end >= 2min,
    e.rt.begin - f.rt.end >= 2min
  }
}
```

Figure 10.2: Translation of Relations and Functions

Simultaneously, interval functions operating on event identifiers, namely, `time(e)`, `begin(e)`, and `end(e)`, are translated by replacing them with `[e.rt.begin, e.rt.end]`, `e.rt.begin`, and `e.rt.end`, respectively. In addition to the transformation of composite functions, temporal relations are translated to composite formulas in a similar manner. The corresponding definition can be found in Table 10.2.

`begin([l,u])`	$\equiv$	`l`
`end([l,u])`	$\equiv$	`u`
`extend(e, d)`	$\equiv$	`[begin(e), end(e) + d]`
`shorten(e, d)`	$\equiv$	`[begin(e), end(e) - d]`
`extend-begin(e, d)`	$\equiv$	`[begin(e) - d, end(e)]`
`shorten-begin(e, d)`	$\equiv$	`[begin(e) + d, end(e)]`
`shift-forward(e, d)`	$\equiv$	`[begin(e) + d, end(e) + d]`
`shift-backward(e, d)`	$\equiv$	`[begin(e) - d, end(e) - d]`
`from-end(e, d)`	$\equiv$	`[end(e), end(e) + d]`
`from-end-backward(e, d)`	$\equiv$	`[end(e) - d, end(e)]`
`from-begin(e, d)`	$\equiv$	`[begin(e), begin(e) + d]`
`from-begin-backward(e, d)`	$\equiv$	`[begin(e) - d, begin(e)]`
`e` *before* `f` $\equiv$ `f` *after* `e`	$\equiv$	`end(e) < begin(f)`
`e` *meets* `f` $\equiv$ `f` *met-by* `e`	$\equiv$	`end(e) = begin(f)`
`e` *overlaps* `f` $\equiv$ `f` *overlapped-by* `e`	$\equiv$	`and{ begin(e) < begin(f), begin(f) < end(e), end(e) < end(f) }`
`e` *during* `f` $\equiv$ `f` *contains* `e`	$\equiv$	`and{ begin(f) < begin(e), end(e) < end(f) }`
`e` *starts* `f` $\equiv$ `f` *started-by* `e`	$\equiv$	`and{ begin(e) = begin(f), end(e) < end(f) }`
`e` *finishes* `f` $\equiv$ `f` *finished-by* `e`	$\equiv$	`and{ end(e) = end(f), begin(e) > begin(f) }`
`e` *equals* `f`	$\equiv$	`and{ begin(e) = begin(f), end(e) = end(f) }`
`{e,f}` *apart-by* `d`	$\equiv$	`or{ begin(f) - end(e) >= d, begin(e) - end(f) >= d }`
$\{e_1, \ldots, e_k\}$ *within* `d`	$\equiv$	`greatest(end(`e_1`), ..., end(`e_k`)) - least(begin(`e_1`), ..., begin(`e_k`)) <= d`

Table 10.2: Translation of Relations and Functions whereby the expressions `l` and `u` correspond to timestamps, `d` corresponds to a duration, and `e`, $e_1, \ldots, e_k$ and `f` correspond to identifiers or intervals.

10.3.3 *Eliminating Identifiers in Groupings*

Event identifiers have, similar to variables, no direct correspondence in TSA and thus every event identifier specified in the `group by` part of a query is replaced with the set of all basic attributes of the event that is referenced by the identifier.

```
and{
  event e: alarm{},
  ...
} group by { event e }
```

```
and{
  event e: alarm{},
  ...
} group by {
  e.area, e.id, e.rt.begin, e.rt.end
}
```

Figure 10.3: Translation of Event Identifiers: Groupings

This transformation eliminates all remaining event identifiers in the supplement of a query, as all identifiers contained in `where` and `let` parts have been eliminated by the previous transformation. Note, however, that the identifiers preceding atomic queries, which are references by paths, are preserved so as to obtain valid Dura queries.

10.3.4 *Adding Value Definitions in Disjunctions*

Disjunctions are translated to unions, which can only be applied to streams with equal sets of attributes. As a consequence, only attributes that are common to all events contained within a disjunction are available in the attributes of the resulting stream. However, attributes referenced by variables may be absent in the stream corresponding to the translation of the disjunctive query.

```
or{
  event e: sensor-msg{ area{var A} }
  event e: temp-value{ aid{var A} }
}
```

```
or{
  event e: sensor-msg{}
    let { var A = e.area }

  event e: temp-value{}
    let { var A = e.aid }
}
```

Figure 10.4: Adding Value Definitions

To avoid the omission of attributes referenced by variables, variables with positive polarity in the disjunctive query are transformed to variable definitions in a `let` part, as it is illustrated in Figure 10.4. In this way, the value of each variable with positive polarity is copied to a distinguished attribute with the same name in each sub-expression contributing to the union eg, *.let.A* in case of the query above. Accordingly, attributes referenced by variables with positive polarity are

copied to attributes that remain in the resulting stream whereas the attributes they actually refer to are eliminated.

10.3.5 *Eliminating Redundant Variable Definitions*

Variables are a concept of Dura that is not available in TSA and therefore variables need to be eliminated in order to facilitate the transformation of Dura to TSA. Variables specified in a query term correspond to variable definitions basically associating a name with an attribute of an event. Therefore, variables are simply translated to the attributes they refer to.

However, equally named variables occurring in different locations of a query imply a join between the attributes they are referring to. Accordingly, to eliminate multiple definitions of equally named variables without affecting the semantics of queries, the join conditions need to be explicitly specified in the `where` part of the query, as it is illustrated in Figure 10.5. Note that the variable definition may be referenced in other query parts and therefore, similar to the elimination of identifiers in Section 10.3.3, one variable definition is preserved so as to obtain valid Dura queries.

```
and{
  event e: temp{ area{var A} },
  event f: smoke{ area{var A} }
}
```

```
and{
  event e: temp{ area{var A} },
  event f: smoke{}
} where { e.area = f.area }
```

Figure 10.5: Materializing Implied Join Conditions

A similar transformation is illustrated in Figure 10.6 to proceed with implied conditions involving variables introduced in `let` parts. Note, however, that the condition added to the query corresponds to *var* `C = f.value-273.15`, including the variable `C`, instead of the path to its definition. Variables solely defined in `let` are in general not referring to the attributes of an event and thus cannot be expressed by means of a path.

```
and{
  event e: temp-value{ val{var F} }
    let { var C = (F-32)/1.8 }

  event f: sensor-msg{}
    let { var C = f.value-273.15 }
}
```

```
and{
  event e: temp-value{ val{var F} }
    let { var C = (F-32)/1.8 }

  event f: sensor-msg{}
} where { var C = f.value-273.15 }
```

Figure 10.6: Eliminating Redundant Variable Definitions

As a result of these transformations, implied join conditions caused by equally named variables are explicitly specified as a condition in

the `where` part of queries. Moreover, each remaining variable definition contained in a conjunctive query is unambiguous and refers to the attributes of one distinguished event or to a term specified in the `let` part of a query.

10.3.6 *Establishing Range Restriction*

The positioning of references is quite liberal in Dura as queries are merely required to be weakly range restricted. In consequence, the `where` part of queries may contain references with weak positive polarity. However, the translation of Dura queries to TSA expressions often involves projections omitting attributes of the stream. As a consequence, it is mandatory for the translation of queries that references appear at the right position, that is, queries need to be properly range restricted whereas being weakly range restricted is no longer sufficient. Consequently, queries and their conditions may need to be restructured in order to adapt the polarity of references appropriately.

In the following, a condition of a query denotes one conjunct of the conjunctive normal form of the formula specified in the supplemental `where` part of the query.[2] Moreover, before any condition is moved to another location by one of the following translations, the formula of the corresponding `where` part is firstly converted into conjunctive normal form.

CONDITIONS WITH POSITIVE POLARITY First of all, conditions containing only references that have positive polarity in a particular sub-query `q` are moved to the `where` part in the supplement of `q`. This applies, eg, to the conditions in Figure 10.7 which are moved to the atomic event queries they refer to. Note that, although this transformation may improve the efficiency of the evaluation, its major intention is to prepare the queries for the following transformation.

As a result, all remaining conditions containing references with weak positive polarity necessarily also contain references with positive polarity. However, there is no common sub-query `q` so that all references have positive polarity in `q`. Those kind of conditions are further treated by the following two transformations.

EXPANDING DISJUNCTIONS This transformation applies for conjunctive queries containing conditions referring to a disjunctive subquery. Consider the query `q` determined by

$$\texttt{and\{ } c_1\texttt{, ..., } c_k\texttt{, or\{ } d_1\texttt{, ..., } d_\ell\texttt{ \} \}}$$

If there is a condition of `q` containing a reference with positive polarity in a c_i as well as a reference with positive polarity in a d_j, then the

2 Note that for formulas solely composed from atomic formulas and conjunctions, each atomic sub-formula corresponds to a condition.

```
and{
  event e: alarm{}

  or{
    event f: temp{}
    event g: smoke{}
  }
} where {
  f.value > 50, g.conc > 0.1
}
```

```
and{
  event e: alarm{}

  or{
    event f: temp{}
      where { f.value > 50 }

    event g: smoke{}
      where { g.conc > 0.1 }
  }
}
```

Figure 10.7: Relocating Conditions with Positive Polarity

```
and{
  event e: alarm{}

  or{
    event f: temp{}
    event g: smoke{}
  }
} where {
  f.rt.end < e.rt.end + 1min,
  g.rt.end < e.tr.end,
}
```

```
or{
  and{
    event e: alarm{}
    event f: temp{}
  }

  and{
    event e: alarm{}
    event g: smoke{}
  }
} where {
  f.rt.end < e.rt.end + 1min,
  g.rt.end < e.tr.end,
}
```

Figure 10.8: Expanding Disjunctions

```
and{
  event e: temp{}

  not event f: temp{}
} where {
  e.id = f.id,
  e.rt.end < f.rt.begin,
  f.rt.end < e.rt.end + 1min
}
```

```
and{
  event e: temp{}

  not {
    event f: temp{}
  } where {
    e.id = f.id,
    e.rt.end < f.rt.begin,
    f.rt.end < e.rt.end + 1min
  }
}
```

Figure 10.9: Relocating Conditions with Weak Positive Polarity

disjunctive query is expanded using the distributivity law as it is known from mathematical logic and illustrated in Figure 10.8. Note that in this example both atomic formulas satisfy the properties that entail the expansion.

As a result, for all conditions containing references with weak positive polarity there is either a sub-query `q` so that all references of the condition have positive polarity in `q` or there is a sub-query `not q` so that all references of the condition have positive or weak positive polarity in the query comprising `not q`. Those conditions are subsequently relocated in a manner similar to the one described above.

CONDITIONS WITH WEAK POSITIVE POLARITY The remaining conditions containing references without positive polarity can be relocated to their proper location by means of the following transformation.

Suppose there is a condition containing a reference `r` without positive polarity. If `r` refers to an atomic query nested within a negation, then the condition is added to the `where` part of the negated query. This applies, for instance, to the conditions in Figure 10.9 which are hence moved to the `where` part of the negation.

Otherwise, the condition is moved to the supplement of the particular sub-query `q` which satisfies that `r` has positive polarity in `q`. This applies to the conditions in Figure 10.8 which need to be moved to the `where` part of the respective conjunctive sub-queries.

10.3.7 *Summary*

In result of the applied transformations the obtained queries are satisfying the following properties.

All rules are range restricted rather than weakly range restricted. Moreover, as a consequence of rules being range restricted, negated queries are directly contained in conjunctive queries.

Join conditions caused by the usage of equally named variables as well as restrictions imposed by literals in query terms are explicitly expressed by appropriate formulas in the `where` part of queries. Accordingly, all references can be directly translated to an attribute of a stream.

Atomic formulas have the form $t_1\ \theta\ t_2$ with $\theta \in \{<, <=, =, >=, >, !=\}$ whereas composite formulas are composed from the n-ary connectives `and` and `or` and the unary connective `not`.

Terms in `aggregate` parts of queries are composed from references and the aggregation functions `min`, `max`, `avg`, `sum`, `count` whereas all other terms are composed from literals, variables and paths, the nullary functions `system-time.now()` and `sequence.next()`, the binary functions `+`, `-`, `*`, `/`, `**`, `floor` and `ceil`, and the n-ary functions `least` and `greatest`.

Finally, the conditions on progressing attributes are not affected by the transformations as the existing conditions are not eliminated but merely expressed in an equivalent manner and relocated to different parts of the query in order to establish range restriction. Accordingly, the conditions on progressing attributes are preserved by the transformations.

10.4 TRANSLATING DURA_C TO TSA

Eventually, Dura queries and rules are translated to composite TSA algebra expressions. To this end, queries are translated to expressions by recursively applying the following transformations. Thereby syntactically correct and range restricted queries satisfying the conditions on progressing attributes discussed in Section 5.2.5 yield valid TSA expressions, that is, syntactically correct expressions that can be incrementally evaluated and pass the static analysis of TSA.

10.4.1 *Event Schemata*

The schema of streams as they are introduced in [BB12a] corresponds to a triple $S = (A, G, H)$. In simplified terms, A is the set of attributes of the stream, G is a formula providing information on temporal dependencies between attributes, and H is a formula designating the progressing attributes of a stream.

An event schema is translated to the corresponding TSA stream schema $S = (A, G, H)$ in the following way. The attributes of the event are mapped to the set of attributes A by interpreting the paths to all leaves of the event schema as attributes. Temporal dependencies between attributes, which coincide with atomic formulas composed from inequations in the `where` part of the event schema, are mapped to a formula G by means of a conjunction. And finally, atomic formulas designating the progressing attributes of the event, which coincide with atomic formulas composed from the `progressing` relations in the `where` part of the event schema, are mapped to a formula H by means of a disjunction.[3]

EXAMPLE The event schema from Listing 10.8 is translated to the stream schema $sch(T) = (A, G, H)$ of the stream T with

$$\begin{aligned} A &= \{area, id, rt.begin, rt.end, sensor\text{-}id, value\} \\ G &= rt.begin \leqslant rt.end \\ H &= progressing(id) \vee progressing(rt.end) \end{aligned}$$

Note that the stream schema is of particular importance for the static analysis that is carried out on composite TSA expressions corre-

3 A disjunction is used to specify that it is sufficient to maintain progress information for any of those attributes (see Section 5.2.5).

Listing 10.8: Schema of `temp` Events from Listing 5.21

```
temp{
  id{identifier}, rt{ begin{timestamp}, end{timestamp} },
  area{long}, value{double}, sensor-id{long}
} where {
  progressing(id), progressing(rt.end),
  rt.begin <= rt.end
}
```

sponding to entire Dura queries. The analysis determines properties of streams that are required for the evaluation of TSA expressions and identifies inconsistent temporal dependencies [BB12a].

For the translation of Dura to TSA the schema and in particular the formulas G and H are only relevant in the base case, that is, for the translation of stream schemata, and thus merely the attributes of streams are considered in the following.

10.4.2 *Atomic Queries*

An atomic Dura query

```
event i: e{ ... }
```

translates to the TSA expression

$$\rho[* \to i.*]\big(\mathsf{E}\big)$$

whereby E corresponds to the stream containing all event of type `e`.

10.4.3 *Terms and Formulas*

ATOMIC TERMS The representation of literals is identical in Dura and TSA with the exception of booleans and durations. The boolean literals `true` and `false` are translated to $\top$ and $\bot$, respectively, whereas Dura durations, which are independently specifying the hours, minutes, and so on, are translated to the equivalent amount of seconds.

Constants are translated to the corresponding literal they represent.

Paths in Dura coincide with the attribute names of the corresponding TSA streams.

Variables specified in a query term are translated to attributes corresponding to the path that specifies their location in the query term. Recall that during the normalization of queries ambiguous variable definitions are eliminated and that hence each variable translates to an unambiguous attribute.

Variables introduced by means of `let` and `aggregate` are translated to the attribute name *.var.V* whereby *V* is the name of the variable and *.var* is a prefix that cannot occur in variable names in Dura.

COMPOSITE TERMS AND FORMULAS Composite terms and formulas of normalized Dura queries are canonically mapped to TSA terms and formulas. They merely differ slightly in their syntax and structure, eg, `<=` as opposed to $\leqslant$ and the n-ary connective `and` instead of multiple binary connectives $\wedge$.

10.4.4 *Query Supplements*

SELECTION A `where` part

```
q where c
```

is translated to a selection

$$\sigma[C](Q)$$

whereby Q corresponds to the translation of the query `q` and C corresponds to the translation of the formula `c`.[4]

EMBEDDING A `let` part

```
q let { var v1 = expr1, ..., var vk = exprk }
```

is translated to an embedding

$$\iota[.var.v_1 \leftarrow expr_1, \ldots, .var.v_k \leftarrow expr_k](Q)$$

whereby Q is the translation of the query `q` and $expr_i$ denotes the translation of the term $\texttt{expr}_i$.[5]

GROUPING AND AGGREGATION A generic `group by` and `aggregate` part is determined by

```
q
group by { r1, ..., rk }
aggregate { var v1 = fn1(s1), ..., var vℓ = fnℓ(sℓ) }
```

whereby $\texttt{r}_i$ and $\texttt{s}_i$ denote references and $\texttt{fn}_i$ denotes an aggregation function. The query is translated to the TSA expression

$$\gamma[R_1, \ldots, R_k][.var.v_1 \leftarrow fn_1(S_1), \ldots, .var.v_\ell \leftarrow fn_\ell(S_\ell)](Q)$$

whereby Q corresponds to the translation of `q`. Moreover R_i and S_i correspond to the attributes referenced by $\texttt{r}_i$ and $\texttt{s}_i$, respectively.

Note that in Dura the `aggregate` part is optional and if omitted the second argument of γ remains empty.

4 Note that `where { ... }` is actually syntactic sugar for `where and{ ... }`.

5 Note that in this context $\texttt{v}_i$ is meta variable and not a Dura variable.

10.4.5 *Disjunctive Queries*

A disjunctive query

```
or{ q1, ..., qk }
```

is translated by means of unions and projections

$$\pi[A](Q_1) \cup \pi[A](Q_2) \cup \cdots \cup \pi[A](Q_k)$$

whereby Q_i corresponds to the translation of the sub-query q_i and $A = \bigcap_i \text{sort}(Q_i)$.

10.4.6 *Conjunctive Queries*

A generic conjunctive query is determined by

```
and{
  q1, ..., qk,

  not { n1 } where c1,
  ...,
  not { nℓ } where cℓ
}
```

Recall that Dura is a declarative language and that hence the sub-queries can be arbitrarily reordered as opposed to, eg, Prolog, where a rearrangement of subgoals may affect the semantics and the termination behavior of programs.

The conjunctive query translates to the following TSA expression composed from anti-semi-joins and cross products

$$\begin{array}{l}
\overline{\ltimes}[C_\ell](\\
\quad \cdots \\
\qquad \overline{\ltimes}[C_2](\\
\qquad\quad \overline{\ltimes}[C_1](Q_1 \times Q_2 \times \cdots \times Q_k,\ N_1), \\
\qquad\quad N_2), \\
\qquad \cdots \\
\quad N_\ell)
\end{array}$$

whereby C_i corresponds to the translation of the formula c_i and Q_i and N_i correspond to the translation of the queries q_i and n_i, respectively.

Note that if the conjunctive query does not contain any negated sub-queries, its translation degenerates to the cross product

$$Q_1 \times Q_2 \times \cdots \times Q_k$$

10.4.7 *Deductive Rules*

A deductive rule

```
DETECT type{ ... } ON q END
```

translates to the TSA expression

$$
\begin{aligned}
\mathsf{T} \leftarrow \pi[\mathrm{sort}(\mathsf{T})]\big(&\\
&\iota[\mathsf{P}_1 \leftarrow \mathsf{E}_1, \ldots, \mathsf{P}_k \leftarrow \mathsf{E}_k]\big(\\
&\quad \iota[\mathit{id} \leftarrow \mathrm{sequence.next}(),\\
&\qquad \mathit{rt.begin} \leftarrow \mathrm{least}(e_1.\mathit{rt.begin}, \ldots, e_\ell.\mathit{rt.begin}),\\
&\qquad \mathit{rt.end} \leftarrow \mathrm{greatest}(e_1.\mathit{rt.end}, \ldots, e_\ell.\mathit{rt.end})](\mathsf{Q})\big)\big)
\end{aligned}
$$

whereby T is the stream that corresponds to the event type `type` and Q is the translation of the query `q` and the prefixes $\mathtt{e}_1, \ldots, \mathtt{e}_\ell$ coincide with the identifiers having positive polarity in `q`. Moreover, E_i corresponds to the translation of an (arithmetic) expression in the rule head whereby P_i corresponds to the path describing the location of the expression in the construct term.

Note that furthermore the definition of the attributes *id*, *rt.begin*, and *rt.end* are omitted from the inner most embedding, if they coincide with one of the P_i, that is, if they are explicitly specified in the head of the rule.

10.5 WALK THROUGH OF AN ENTIRE TRANSLATION

Consider the rules from Listing 10.9 which are based on previously introduced queries and rules from Chapter 5. The first rule derives `temp-exceeded-stat` events which determine the number of areas that exceed an average temperature of 100 degrees Celsius within one minute after the occurrence of an alarm. The second rule derives `broken-temp-sensor` events whenever the latest received `temp` event of a certain sensor has been obtained more than one minute ago.

BASIC STREAMS The given rules are based on the events `alarm`, `temp`, `temp-exceeded-stat`, and `broken-temp-sensor` events which correspond to the TSA streams A, T, E, and B with the schemata

$$
\begin{aligned}
\mathit{sch}(\mathsf{A}) &= (\{\mathit{area}, \mathit{id}, \mathit{rt.begin}, \mathit{rt.end}\}, \mathsf{G}, \mathsf{H})\\
\mathit{sch}(\mathsf{T}) &= (\{\mathit{area}, \mathit{id}, \mathit{rt.begin}, \mathit{rt.end}, \mathit{sensor\text{-}id}, \mathit{value}\}, \mathsf{G}, \mathsf{H})\\
\mathit{sch}(\mathsf{E}) &= (\{\mathit{count}, \mathit{id}, \mathit{rt.begin}, \mathit{rt.end}\}, \mathsf{G}, \mathsf{H})\\
\mathit{sch}(\mathsf{B}) &= (\{\mathit{id}, \mathit{rt.begin}, \mathit{rt.end}, \mathit{sensor\text{-}id}\}, \mathsf{G}, \mathsf{H})
\end{aligned}
$$

whereby

$$
\begin{aligned}
\mathsf{G} &= \mathit{rt.begin} \leqslant \mathit{rt.end}\\
\mathsf{H} &= \mathit{progressing}(\mathit{id}) \vee \mathit{progressing}(\mathit{rt.end})
\end{aligned}
$$

Listing 10.9: Rules Derived from Listing 5.12 and Listing 5.17

```
DETECT
  temp-exceeded-stat{ count{var Acount} }
ON
  and{
    event e: alarm{},
    event f: temp{ area{var A}, value{var T} }
  } where { f during from-end(e, 1min) }
    group by { e, var A } aggregate { var Tavg = avg(var T) }
    where { var Tavg > 100 }
    group by { e } aggregate { var Acount = count(var A) }
END

DETECT
  broken-temp-sensor{
    sensor-id{var Id}, rt{ end{ end(e)+15sec } }
  }
ON
  and{
    event e: temp{ sensor-id{var Id} }
    not event f: temp{ sensor-id{var Id} }
  } where { f during from-end(e, 1min) }
END
```

Listing 10.10: Normalization of Rules from Listing 10.9

```
DETECT
  temp-exceeded{ count{var Acount} }
ON
  and{
    event e: alarm{},
    event f: temp{ area{var A}, value{var T} }
  } where { e.rt.end < f.rt.begin, f.rt.end < e.rt.end+1min }
    group by { e.area, e.id, e.rt.begin, e.rt.end, var A }
    aggregate { var Tavg = avg(var T) }
    where { var Tavg > 100 }
    group by { e.area, e.id, e.rt.begin, e.rt.end }
    aggregate { var Acount = count(var A) }
END

DETECT
  broken-temp-sensor{
    sensor-id{var Id}, rt{ end{ e.rt.end+15sec } }
  }
ON
  and{
    event e: temp{ sensor-id{var Id} }
    not { event f: temp{} } where {
      e.rt.end < f.rt.begin, f.rt.end < e.rt.end+1min,
      e.sensor-id = f.sensor.id
    }
  }
END
```

NORMALIZATION OF RULES The normalization of the rules from Listing 10.9 is illustrated in Listing 10.10:

The temporal condition `f during from-end(e, 1min)` is replaced with the formula `and{ e.rt.end < f.rt.begin, f.rt.end < e.rt.end + 1min }`. In case of the second rule, the formula is subsequently relocated to the `where` part of the negation. Finally, in both `group by` parts of the first rule the event identifier `e` is replaced by `e.area, e.id, e.rt.begin, e.rt.end`.

TRANSLATION TO TSA Subsequently, the normalized queries from Listing 10.10 are recursively translated to TSA expressions. Note that thereby, each sub-query can be individually translated as all potential interdependencies between sub-queries have been eliminated by the preceding normalization of queries.

In case of the first rule deriving `temp-exeeded` events the translation to TSA yields the expression

$$
\begin{aligned}
&\mathrm{E} \leftarrow \pi[\mathit{count}, \mathit{id}, \mathit{rt.begin}, \mathit{rt.end}]\,(\\
&\quad \iota[\mathit{count} \leftarrow .\mathit{var.Acount}]\,(\\
&\quad\quad \iota[\mathit{id} \leftarrow \text{sequence.next}(),\\
&\quad\quad\ \mathit{rt.begin} \leftarrow \text{least}(\mathit{e.rt.begin}),\\
&\quad\quad\ \mathit{rt.end} \leftarrow \text{greatest}(\mathit{e.rt.end})]\,(\\
&\quad\quad\quad \gamma[\mathit{e.area}, \mathit{e.id}, \mathit{e.rt.begin}, \mathit{e.rt.end}]\\
&\quad\quad\quad\ [.\mathit{var.Acount} \leftarrow \text{count}(\mathit{f.area})]\,(\\
&\quad\quad\quad\quad \sigma[.\mathit{var.Tavg} > 100]\,(\\
&\quad\quad\quad\quad\ \gamma[\mathit{e.area}, \mathit{e.id}, \mathit{e.rt.begin}, \mathit{e.rt.end}, \mathit{f.area}]\\
&\quad\quad\quad\quad\quad [.\mathit{var.Tavg} \leftarrow \text{avg}(\mathit{f.value})]\,(\\
&\quad\quad\quad\quad\quad\quad \sigma[\mathit{e.rt.end} < \mathit{f.rt.begin},\ \mathit{f.rt.end} < \mathit{e.rt.end} + 60]\,(\\
&\quad\quad\quad\quad\quad\quad\quad \times\,(\rho[* \to \mathit{e}.*]\,(\mathrm{A}),\ \rho[* \to \mathit{f}.*]\,(\mathrm{T}))\cdots)
\end{aligned}
$$

whereas the second rule deriving `broken-temp-events` is eventually translated to the expression

$$
\begin{aligned}
&\mathrm{B} \leftarrow \pi[\mathit{id}, \mathit{rt.begin}, \mathit{rt.end}, \mathit{sensor\text{-}id}]\,(\\
&\quad \iota[\mathit{sensor\text{-}id} \leftarrow \mathit{e.sensor\text{-}id},\ \mathit{rt.end} \leftarrow \mathit{e.rt.end} + 15]\,(\\
&\quad\quad \iota[\mathit{id} \leftarrow \text{sequence.next}(),\\
&\quad\quad\ \mathit{rt.begin} \leftarrow \text{least}(\mathit{e.rt.begin})]\,(\\
&\quad\quad\quad \overline{\ltimes}[\mathit{e.rt.end} < \mathit{f.rt.begin},\ \mathit{f.rt.end} < \mathit{e.rt.end} + 60,\\
&\quad\quad\quad\ \mathit{e.sensor\text{-}id} = \mathit{f.sensor\text{-}id}]\,(\\
&\quad\quad\quad\quad \rho[* \to \mathit{e}.*]\,(\mathrm{T}),\ \rho[* \to \mathit{f}.*]\,(\mathrm{T})))))
\end{aligned}
$$

11 TRANSLATING DURA TO DURA$_C$

Having established an operational semantics for Dura$_C$ in the previous chapter, we now elaborate a translation of generic Dura rules to Dura$_C$ rules in order to obtain an operational semantics for (full) Dura. To this end, all constructs that are not available in Dura$_C$ need to be expressed by means of Dura$_C$ event queries and deductive rules. This includes in particular queries for stateful objects; internal, external, and complex actions; and reactive and complex action rules.

The translation of Dura to Dura$_C$ is separated into aspects concerned with the execution of external and complex actions, which is subject to Section 11.1, and aspects concerned with queries of stateful objects and internal actions on stateful objects, which is is subject to Section 11.2.

In the following, the translations are illustrated by means of relevant excerpts of the resulting deductive rules. For the sake of completeness and further reference, the entire rule sets are contained in Appendix B.

11.1 ACTIONS IN DURA$_C$

Actions have no direct representation in Dura$_C$ and thus they need to be modeled by means of events. As a consequence, reactive and complex action rules need to be translated to deductive rules that realize the intended effect of the specified actions by deriving appropriate events.

The translation of complex action and reactive rules is realized by means of several translation steps that are subsequently applied to a program. Each step is devoted to a certain aspect of the translation and produces intermediate rules that are the basis for the next translation step.

An early draft for the translation of complex actions was elaborated by Scherr in his Diploma thesis [Sch11] supervised by François Bry and the author of this thesis. However, as the language definition and the basic assumptions have substantially changed in the mean time, only the basic ideas for the translation of complex actions can be adopted from [Sch11].

11.1.1 *Informal Introduction*

Information on action instances is internally represented by means of events. To this end, the invocation of an action is indicated by the

occurrence of a corresponding «name»$requested event, whereas the actual initiation and the success or failure of an action is indicated by «name»$initiated, «name»$succeeded, and «name»$failed events.

For instance, the immediate-reactions action from Listing 11.2 is represented by four events whose schema is indicated in Listing 11.1 whereby «type» corresponds to requested, initiated, succeeded, and failed. Thereby, the end of the reception time of, eg, succeeded events indicates the success of the action instance referred to by the respective payload.id attribute.

Listing 11.1: Events Associated with the immediate-reactions Action

```
immediate-reactions$«type»{
  pl{ id{identifier}, ref{identifier}, area{long} }
}
```

Be aware that, for the sake of conciseness and readability, the label of the composite attribute payload is abbreviated with pl in this and all following examples. Moreover, the prefix of event names referring to actions may be omitted in the following, if the complete name is clear from the context.

TRANSLATING REACTIVE RULES When an atomic action is triggered by means of a reactive rule, a requested event is derived internally that indicates the invocation of the respective action. This is realized by translating the reactive rule to a deductive rule as it is illustrated in Listing 11.3. In case of external actions, the occurrence of a requested event causes furthermore the delivery of a message to the respective actuator where it triggers the actual execution of the action.

Listing 11.2: A Reactive Rule

```
ON
  event e: certain-fire-alarm{ area{var A} }
DO
  action a: immediate-reactions{ area{var A} }
END
```

Listing 11.3: Translation of the Reactive Rule from Listing 11.2

```
DETECT
  immediate-reactions$requested{ pl{
    id{sequence.next()}, area{var A}
  } }
ON
  event e: certain-fire-alarm{ area{var A} }
END
```

TRANSLATING COMPLEX ACTIONS Similarly to the invocation of actions by means of reactive rules, the invocation of sub-actions specified in a complex action is realized by the derivation of appropriate `requested` events. Therefore, the invocation of each sub-action is realized by means of a separate rule that derives an appropriate `requested` event. However, thereby the temporal dependencies that are specified in the `where` part of the complex action need to be properly taken into account for the invocation of each sub-action. To this end, the body of each rule deriving `requested` events contains event queries that are used to determine the earliest time for the invocation of the sub-action according to the temporal conditions of the complex action.

Consider, for instance, the complex action rule in Listing 11.4 containing the `activate-ventilators` sub-action that is referenced by the identifier `b`. Obviously `b` must not be initiated before the initiation of the complex action `v` of which it is a sub-action. Moreover, both temporal dependencies in the `where` part further constrain the initiation of `b` such that `b` must not be initiated until the success of the sub-action `a` nor until one minute after the initiation of `c`. Accordingly, `b` must not be initiated before `greatest(init(v), succ(a), init(c)+1min)` whereby the time points `init(v)`, `succ(a)`, and `init(c)` can be obtained by means of queries to

```
adapt-ventilation$initiated
open-fire-dampers$succeeded
warn-of-smoke$initiated
```

events that have been caused by the execution of the complex action. Note that, with slight abuse of notation, the earliest time for the initiation of `b` coincides with `greatest(pre`$_{W_A}$`(init(b)))`.[1]

The deductive rule responsible for the invocation of the sub-action `b` is illustrated in Listing 11.5. Note how the area parameter `A` of the complex action is obtained in the deductive rule by means of a query to the `adapt-ventilation$initiated` event. Moreover, the reception time of derived events is determined by the earliest possible time for the initiation of `b` in order to respect the temporal dependencies that are specified in the complex action.

TRANSLATING STATUS QUERIES The translation of status queries is closely related to the translation of action invocations in complex actions. Status queries are basically translated to deductive rules, eg, deriving `succeeded` events in case of a `succeeds on` part, and references to the times of actions and their parameters are substituted with references to appropriate event queries.

The translation of the `succeeds on` part of the complex action in Listing 11.4 is illustrated in Listing 11.6.

1 Recall from Section 7.3.2 that $\text{pre}_{W_A}(v) = \{u + d \mid u + d \leqslant v \text{ is sub-formula of } W_A\}$ whereby W_A is determined by the axiomatic closure of the complex action's temporal dependencies.

Listing 11.4: A Complex Action Rule

```
FOR
  action v: adapt-ventilation{ area{var A} }
DO
  compound{
    action a: open-fire-dampers{ area{var A} }
    action b: activate-ventilators{ area{var A} }
    action c: warn-of-smoke{ area{var A} }
  } where { succ(a) <= init(b), init(c)+1min <= init(b) }
    succeeds on and{
      event e: smoke{ area{var A}, amount{var C} }
    } where { var C < 0.1, init(v) < end(e), end(e) < init(v)+2min }
      group by { v }
END
```

Listing 11.5: Translation of the Invocation of `b` from Listing 11.4

```
DETECT
  activate-ventilators$requested{
    reception-time{ end{
      greatest(end(v$init), end(a$succ), end(c$init)+1min)
    } }
    pl{ area{var A}, id{sequence.next()}, ref{v$init.pl.id} }
  }
ON
  and{
    event v$init: adapt-ventilation$initiated{ pl{ area{var A} } }
    event a$succ: open-fire-dampers$succeeded{}
    event c$init: warn-of-smoke$initiated{}
  } where {
    v$init.pl.id = a$succ.pl.ref, v$init.pl.id = c$init.pl.ref
  }
END
```

Listing 11.6: Translation of the `succeeds on` Part from Listing 11.4

```
DETECT
  adapt-ventilation$succeeded{ pl{ v$init.pl } }
ON
  and{
    event e: smoke{ area{var A}, amount{var C} }
    event v$init: adapt-ventilation$initiated{ pl{ area{var A} } }
  } where {
    var C < 0.1, end(v$init) < end(e), end(e) < end(v$init)+2min
  } group by { v$init }
END
```

STEPWISE TRANSLATION The translation of complex actions and reactive rules is carried out in a stepwise manner. Initially, anonymous complex actions are eliminated by assigning them a (generated) name. Subsequently, complex action rules are translated to reactive rules and status queries of complex action rules and external actions are translated to deductive rules. Finally, reactive rules are translated to deductive rules deriving `requested` events.

11.1.2 *Representation of Actions*

Each action is internally represented by means of the four event types illustrated in Listing 11.7. Thereby, `«parameters»` corresponds to the attributes `id{`*`identifier`*`}` and `ref{`*`identifier`*`}` and the explicit parameters of the action.

Listing 11.7: Schema of Events Representing Actions

```
«name»$requested{ pl{ «parameters» } }
«name»$initiated{ pl{ «parameters» } }
«name»$succeeded{ pl{ «parameters» } }
«name»$failed{ pl{ «parameters» } }
```

All events that are associated with a certain action instance refer to the same key in their `pl.id` attribute. Moreover, all events refer to the key of the complex action that caused their invocation in their `pl.ref` attribute. In addition, the explicit parameters passed to an action are found in the `pl` attribute and the end of the reception time of each event coincides with the respective time of the action. Eg, if `«id»` is an action identifier referring to an action `«name»`, then the time point referenced by `succ(«id»)` coincides with the one referenced by `end(«name»$succeeded)`.

11.1.3 *Eliminating Anonymous Complex Actions*

To begin with, all anonymous complex actions, that is, complex actions that are directly specified in the head of a reactive rule, are eliminated by transforming them to a complex action rule with an (arbitrary) generated name. Subsequently, the head of the reactive rule is replaced with an invocation of the newly introduced (named) complex action.

The corresponding translation is illustrated in Listing 11.8 and Listing 11.9. Thereby, `«generated parametes»` is determined by the set of variables that occur in `«complex action»` and have positive polarity in `«event query»`. In result of the transformation, all reactive rules specify a single action in their head and all complex actions are associated with a proper name. Both of these aspects are exploited by subsequent transformations.

Listing 11.8: A Reactive Rule Specifying an Anonymous Action

```
ON
  «event query»
DO
  «complex action»
END
```

Listing 11.9: Translation of the Reactive Rule from Listing 11.8

```
ON
  «event query»
DO
  action «generated id»: «generated name»{ «generated parameters» }
END

FOR
  action «generated id»: «generated name»{ «generated parameters» }
DO
  «complex action»
END
```

11.1.4 *Translating Complex Action Rules*

The translation of complex actions divides each complex action into several separate reactive rules, namely, one reactive rule for each sub-action that is invoked by the complex action. Thereby, each reactive rule is intended to trigger the execution of one sub-action according to the (temporal) dependencies specified for the complex action.

The generic scheme for the translation of action invocation is illustrated in Listing 11.11. Thereby, the set of queries from `IF` parts that constrain the sub-action execution corresponds to the queries of all `IF` parts the sub-action is nested within. It is naturally empty if the sub-action is not nested within any conditional action. Moreover, W_A corresponds to the axiomatic closure of the `«temporal dependencies»` specified in the `where` part of the complex action.

Note how in the reactive rule in Listing 11.11 the `ref` parameter of the invoked action is determined by the key of the complex action instance that caused its invocation. In this way, sub-actions that are invoked by one particular complex action instance can be discriminated from equally named sub-actions that happen to be invoked by other rules that are not related to the considered complex action instance. This is important, as it would be incorrect in the context of the complex action `adapt-ventilation` from Listing 11.4 to initiate a `activate-ventilators` action in response to the success of a `open-fire-damper` action that has not been caused by the complex action `adapt-ventilation`.

Note that a similar technique is required if the same action is specified multiple times within a complex action. This can be simply real-

Listing 11.10: A Generic Complex Action Rule

```
FOR
  action «complex action identifier»: «complex action name»{
    «parameters»
  }
DO
  compound{
    action «sub-action identifier>: «sub-action name»{
      «sub-action parameters»
    }
    ...
  } where { «temporal dependencies» }
    fails on { «fails query» }
    succeeds on { «succeeds query» }
END
```

Listing 11.11: Translation Scheme for Sub-Action Invocation

```
ON
  and{
    event: «complex action name»$initiated{ pl{«parameters»} }
    «queries from IF parts that constrain the sub-action execution»
  }
DO
  action: «sub-action name»{
    «sub-action parameters»
    ref{ id(«complex action identifier») }
    request-time{ greatest(pre_{W_A}(init(«sub-action identifier»))) }
  }
END
```

Listing 11.12: Translation Scheme for `initiated` Queries

```
DETECT
  «action name»$initiated{ pl{«id».pl} }
ON
  and{
    «initiates query»
    event «id»: «action name»$requested{ pl{«parameters»} }
  }
END
```

Listing 11.13: Translation Scheme for `succeeded` and `failed` Queries

```
DETECT
  «action name»$«status»{ pl{«id».pl} }
ON
  and{
    «status query»
    event «id»: «action name»$initiated{ pl{«parameters»} }
  }
END
```

ized by adding the (unique) name of the respective sub-action identifier to the parameters of the invoked sub-action, similar as it is accomplished by the `ref` attribute. However, for the sake of simplicity, this aspect is not further considered in the following.

11.1.5 *Translating Status Queries*

Status queries specified in complex action rules or in the schema of external actions are translated into deductive rules as it is illustrated in Listing 11.12 and Listing 11.13.

Thereby, «`status query`» corresponds to the event query that is either specified in the `succeeds on` or `fails on` part of the corresponding action and «`id`» is an identifier that does not occur in «`status query`». Note that because `initiates on` can only be specified for external actions the «`initiated query`» in Listing 11.12 is empty for the translation of complex action rules.

11.1.6 *Illustrative Examples*

Applying the preceding two transformations to the complex action rule from Listing 11.4 yields the rules in Listing 11.14.

Note that most of these intermediate rules are not range restricted, as they contain action identifiers that are not, and cannot be, positive in the respective rule bodies as queries in rule bodies cannot contain actions. Moreover, values for `request-time` are specified which is no proper parameter of any of the actions.

These issues are addressed by the following translations. However, the translations are only illustrated for some of these intermediate rules. The complete translation is contained in Appendix B, though.

11.1.7 *Translating Action Identifier*

The deductive and reactive rules generated by the previous translation step contain references to actions, in particular to their times. To obtain valid Dura$_C$ rules, these references need to be expressed by means of appropriate event queries that provide the referenced information in their payload.

The mapping of action identifiers to attributes of the corresponding events is illustrated in Table 11.1. Thereby, «`id`» may be chosen from among any identifier `a$req`, `a$init`, `a$succ`, or `a$fail` that has positive polarity in the body of the rule. It defaults to `a$init` if no such identifier exists. In addition, for each event identifier that is introduced by this translation appropriate event queries are added to the rule body.[2]

2 Note that it is thus convenient to begin with the translation of `init(a)`, `succ(a)`, `fail(a)` and afterward translate the remaining action identifiers.

Listing 11.14: Intermediate Translation of the Rule from Listing 11.4

```
ON
  event v$init: adapt-ventilation$initiated{ pl{ area{var A} } }
DO
  action a: open-fire-dampers{
    area{var A},
    ref{id(v)}, request-time{ greatest(init(v)) }
  }
END

ON
  event v$init: adapt-ventilation$initiated{ pl{ area{var A} } }
DO
  action a: activate-ventilators{
    area{var A},
    ref{id(v)},
    request-time{ greatest(init(v), succ(a), init(c)+1min) }
  }
END

ON
  event v$init: adapt-ventilation$initiated{ pl{ area{var A} } }
DO
  action a: warn-of-smoke-emission{
    area{var A},
    ref{id(v)}, request-time{ greatest(init(v)) }
  }
END

DETECT
  adapt-ventilation$initiated{ pl{ v$req.pl } }
ON
  event v$req: adapt-ventilation$requested{ }
END

DETECT
  adapt-ventilation$succeeded{ pl{ v$init.pl } }
ON
  and{
    event e: smoke{ area{var A}, amount{var C} }
    event v$init: adapt-ventilation$initiated{ pl{ area{var A} } }
  } where { C < 0.1, init(v) < end(e), end(e) < init(v)+2min }
    group by { v }
END
```

`a`	≡	`«id»`
`id(a)`	≡	`«id».pl.id`
`a.«path»`	≡	`«id».pl.«path»`
`req(a)`	≡	`end(a$req)`
`init(a)`	≡	`end(a$init)`
`succ(a)`	≡	`end(a$succ)`
`fail(a)`	≡	`end(a$fail)`

Table 11.1: Translation of Action Identifiers to Attributes of Events

For instance, when `succ(a)` is translated to `end(a$succ)` the query

```
event a$succ: «name»$succeeded{}
  where { «complex action id»$init.pl.id = a$succ.pl.ref }
```

is added to the body of the rule (unless it is already existing there of course) whereby `«name»` coincides with the name of the action that is referenced by `a`. Moreover, `«complex action id»` corresponds to the identifier of the complex action whose translation caused the respective declarative rule. The condition thus ensures that only events that are caused by the same complex action instance, and thus provide the same key in their `pl.ref` attribute, are considered by the respective query.

When the null-ary variant of these functions is used in the schema of external actions, the mapping applies accordingly whereby `«name»` corresponds to the name of the uniquely determined action from the respective schema and the condition of added event queries is dropped.

The translation is illustrated in Listing 11.15 for the reactive rule from Listing 11.14 that is invoking the `activate-ventilators` action. Note that the resulting reactive rule is weakly range restricted. It still determines implicit attributes of the actions which is not legitimate for user defined actions but tolerated for these particular internal rules.

11.1.8 *Translating Reactive Rules*

Reactive rules triggering external or complex actions are simply translated to deductive rules that derive the respective `requested` events. Be aware that for rules triggering internal actions, which are modifying stateful objects, a particular translation is applied that is introduced in Section 11.2.

The respective transformation is illustrated in Listing 11.17. Thereby, $e_1, \ldots, e_k$ correspond to the event identifiers that have positive polarity in `«event query»`. If there is no `request-time` and `ref` attribute in the head of the reactive rule, ie, when the reactive rule is specified

Listing 11.15: Translation of Identifiers of a Rule from Listing 11.14

```
ON
  and{
    event v$init: adapt-ventilation$initiated{ pl{ area{var A} } }
    event a$succ: open-fire-dampers$succeeded{}
    event c$init: warn-of-smoke$initiated{}
  } where {
    v$init.pl.id = a$succ.pl.ref, v$init.pl.id = c$init.pl.ref
  }
DO
  action a: activate-ventilators{
    area{var A},
    ref{v$init.pl.id},
    request-time{greatest(end(init$v), end(a$succ), end(c$init)+1min)}
  }
END
```

Listing 11.16: A Generic Reactive Rule

```
ON
  «event query»
DO
  action «id»: «action name»{
    «parameters»
    ref{«ref»}, request-time{«request time»}
  }
END
```

Listing 11.17: Translation Scheme for Reactive Rules

```
DETECT
  «action name»$requested{
    reception-time{greatest(e1, ..., ek, «request time»)},
    pl{ id{sequence.next()}, ref{«ref»}, «parameters» } }
ON
  «event query»
END

DETECT
  «external action name»{ «id».pl }
ON
  event «id»: «external action name»$requested{}
END
```

by a programmer and not generated by the preceding transformations, then «request time» remains empty and «ref» is determined by sequence.next(). Furthermore, the second rule only applies for external actions, as it is merely responsible for the derivation of the message that is sent to the actuator where it causes the actual invocation of the action.

Note how the end of the reception-time of the derived requested event is determined by greatest(e_1, ..., e_k, «request time») and is thus greater than or equal to the «request time» that is specified for the reactive rule. This is in particular relevant for the invocation of external actions, as in this way the derivation of the «external action name» message that is sent to the actuator is deferred until the determined «request time» is exceeded.

11.2 STATEFUL OBJECTS IN DURA$_C$

Stateful objects have no direct representation in Dura$_C$ and thus need to be represented by means of events. Therefore, queries for stateful objects need to be expressed by means of event queries, and reactive rules modifying stateful objects need to be expressed by means of deductive rules that, by deriving appropriate events, affect the result of event queries representing queries for stateful objects.

The basic ideas for the translation of stateful objects, which are discussed in Section 11.2.1, have been elaborated in collaboration with Simon Brodt.

11.2.1 *Informal Introduction*

The basic idea to model stateful objects in Dura$_C$ is to represent the values of a stateful object by means of appropriate «name»$created and «name»$terminated events that characterize their payload and valid time. Note that as before the label payload is abbreviated by pl and the prefix «name»$ of event names is omitted in textual descriptions.

Listing 11.18: Schema of created and terminated Events

```
operation-mode{ area{long}, mode{int} }
on conflict select { max mode, min id }

operation-mode$created{ pl{ id{identifier}, area{long}, mode{int} } }
operation-mode$terminated{ pl{ id{identifier}, area{long}, mode{int} } }
```

More precisely, a value of a stateful object becoming valid is indicated by means of a created event whereas the validity of a value coming to an end, eg, when the stateful object is updated, is indicated by means of a terminated event that is referring to the respective created event. Thereby, the actual data of a value is contained in

the `pl` attribute of the respective events and the end of their reception times corresponds to the time the value becomes valid and invalid, respectively.

TRANSLATING QUERIES Based on the representation of stateful objects by means of events, queries for stateful objects are translated to conventional event queries. Consider, for instance, the composite query in Listing 11.19 which contains a query for the stateful object `operation-mode`.

Listing 11.19: A Stateful Object Query

```
and{
  event e: certain-fire-alarm{ subarea{var S} }
  state s: operation-mode{ area{var S}, mode{var M} }
} where { s valid-at «tp» }
```

Listing 11.20: Basic Translation of the Query from Listing 11.19

```
and{
  event e: certain-fire-alarm{ subarea{var S} }

  event s$cre: operation-mode$created{ pl{ area{var S}, mode{var M} } }
  not event s$term: operation-mode$terminated{}
} where {
  s$cre.pl.id = s$term.pl.id,
  end(s$cre) < «tp», end(s$term) < «tp»
}
```

When a certain fire alarm occurs, the query matches all values of the stateful object that are valid at time «tp» and refer to the subarea `S`. It furthermore binds the variable `M` to the respective mode of matching values. In other words, the query looks for a value of the stateful object that began to be valid before «tp» and did not become invalid as early as «tp». And as values becoming valid and invalid is represented by means of `created` and `terminated` events, the query for the stateful object can be equivalently expressed by means of the Dura$_C$ query in Listing 11.20.

TRANSLATING REACTIVE RULES Reactive rules that trigger actions altering stateful objects are translated to rules deriving `created` and `terminated` events. Consider, for instance, the rule in Listing 11.21. When a certain fire alarm occurs, the operation mode of all value of the stateful object that are matching the subarea `S` and are valid at time «tp» is set to `OPM_EMERGENCY`.

Accordingly, every matching value becomes invalid at time «tp» and in turn a new value carrying the updated data begins to be valid after «tp». Similar as before, this is realized by rules deriving `created` and `terminated` events as it is illustrated in Listing 11.22. Note that

Listing 11.21: Updating a Stateful Object

```
ON
  and{
    event e: certain-fire-alarm{ subarea{var S} }
    state s: operation-mode{ area{var S} }
  } where { s valid-at end(e) }
DO
  action a: operation-mode$update{
    query{id(s)}, set{ area{var S}, mode{OPM_EMERGENCY} }
  }
END
```

Listing 11.22: Basic Translation of the Rule from Listing 11.21

```
DETECT
  operation-mode$updated{
    old-payload{ s$cre.pl }
    new-payload{ area{var S}, mode{OPM_EMERGENCY} } }
  }
ON
  and{
    event e: certain-fire-alarm{ subarea{var S} }

    event s$cre: operation-mode$created{ pl{ area{var S} } }
    not event s$term: operation-mode$terminated{}
  } where {
    s$cre.pl.id = s$term.pl.id,
    end(s$cre) < end(e), end(s$term) < end(e)
  }
END

DETECT
  operation-mode$created{ pl{var P} }
ON
  event ud: operation-mode$updated{ new-payload{var P} }
END

DETECT
  operation-mode$terminated{ pl{var P} }
ON
  event ud: operation-mode$updated{ old-payload{var P} }
END
```

the body of the reactive rule closely resembles the query from Listing 11.19 and therefore the translation of the body is adopted from Listing 11.20 with minimal adaptations.

MEETING CONDITIONS ON CYCLIC RULES Although the previous examples are convenient to illustrate the basic idea for modeling stateful objects by means of events, the given rules are unsound as they are cyclic and do not satisfy the conditions on cyclic rules.[3] Recall from Section 5.2.4 that these conditions require that the end of the reception time of a derived event exceeds the end of the reception time of recursively queried events by a constant duration.

However, the rule in Listing 11.22 deriving `updated` events queries `created` and `terminated` events and at the same time it (indirectly) derives events of the same type. Thereby, the end of the reception time of the queried events is merely constrained by the two conditions `end(s$cre) < «tp»` and `end(s$term) < «tp»`. Thus, with a properly selected `«tp»` such as `end(e)`, the reception time of derived `created` and `terminated` events coincides with `«tp»`, which comes arbitrarily close to `end(s$cre)` and `end(s$term)`. Therefore, the rule does not conform to the requirements on recursive rules.

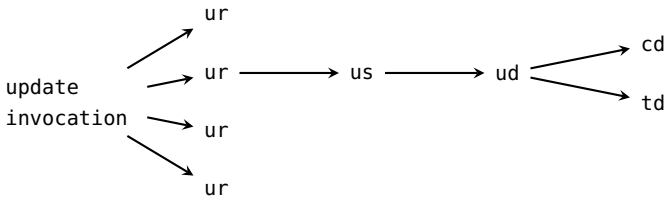


Figure 11.1: Events Caused by the Invocation of an Update Action

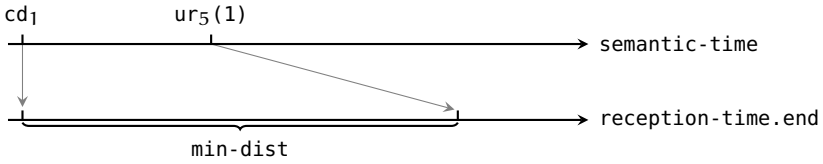
So in order to meet the conditions for cyclic rules, the translation of update invocations is adapted as follows. Instead of directly entailing `updated` events (`ud`), the invocation of an update action entails `update$requested` events (`ur`) whose reception time may be shifted forward by an arbitrary but fixed duration `min-dist`. Moreover, the rule deriving `update$requested` events (`ur`) matches only `terminated` events (`td`) that occur at least `min-dist` before the time of the update invocation. In this way, the conditions on cyclic rules are satisfied but there may be `update$requested` events that would not have been derived if all `terminated` events were queried. Therefore, `terminate$requested` and `update$requested` events that occurred prior to the invocation of the update are queried to determine if an `update$requested` event is actually justified so that eventually an `update$succeeded` event (`us`) can be derived. Finally, each identified `update$succeeded` event (`us`) results in a corresponding `updated` event (`ud`) which in turn causes the derivation of `terminated` and `created` events (`td` and `cd`).

3 Note that the rules are neither stratifiable [ABW88] nor local stratifiable [Prz88].

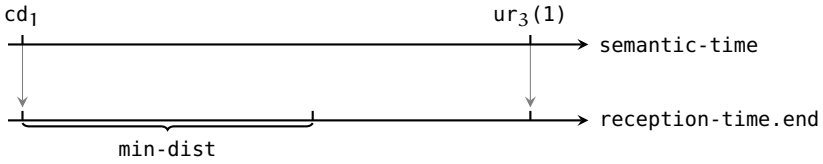
The translation of rules triggering a `terminate` actions is almost identical to the translation of `update` actions whereas the translation of rules triggering `created` actions is straight forward as it does never involve recursion.

SHIFTING THE RECEPTION TIME The end of the reception time of derived `update$requested` events is shifted forward until the time difference between the end of its reception time and the end of the reception time of the `created` event it refers to exceeds `min-dist`. If, however, this time difference is already large enough, the end of the reception time remains unaffected. Moreover, the primal end of the reception time is preserved in the `semantic-time` attribute of the event.

In this way, the end of the reception time of `created` events entailed by the `updated$requested` event exceed the end of the reception time of the queried `created` event by the duration `min-dist`. Note, however, that the shifted time of the `update$requested` event also affects the reception time of `update$suceeded`, `updated`, `created`, and `terminated` events, as these events are derived from `update$requested` events.



(a) An Update Request with Shifted Reception Time



(b) An Update Request without Shifted Reception Time

Figure 11.2: Adaptation of Times of Update Request Events

The adaptation of the reception time is illustrated in Figure 11.2. The labels `cd` and `ur` refer to `created` and `update$requested` events and indicate the respective values of their `reception-time.end` and `semantic-time` attributes. The index of each label coincides with the value of their `pl.id` attribute and the number inside the brackets of the `update$requested` event refers to the key of the value that is to be updated, that is, to the index of a `cd` label. Moreover, the gray arrow indicates the value of the `reception-time.end` attribute relative to the `semantic-time` of each event.

MASKING TERMINATED EVENTS A temporal condition is added to the body of the rule deriving `update$requested` events so that only `terminated` events occurring at least `min-dist` before the time of the

update invocation are matched. That is, `terminated` events that are temporarily to close to the update invocation are masked.

In this way, the end of the reception time of `terminated` events entailed by the `updated$requested` event exceed the end of the reception time of the queried `terminated` event by the duration `min-dist`. However, in consequence, there may occur `update$requested` that would not have been derived if all `terminated` events were considered. Naturally, these events must not cause a `updated` event because they refer to a value that is already terminated. Accordingly, these kind of events are called false positives whereas `update$request` events that refer to a value that can be updated are called true positives.

Figure 11.3: A False Positive Update Request

The masking of events is illustrated in Figure 11.3. Similar to before, the labels `cd`, `td`, and `ur` refer to events and their times. The gray area indicates the time interval of duration `min-dist` in which `terminated` are masked for the rule deriving the `update$requested`, ie, for the rule deriving the event represented by the label $\texttt{ur}_7\texttt{(1)}$.

Note that the value represented by the `created` event is actually terminated by the $\texttt{td}_3\texttt{(1)}$ event. It is thus no longer valid when the `update$requested` event is derived. Accordingly, $\texttt{ur}_7\texttt{(1)}$ is a false positive `update$requested` event, as it would not have been derived, if the $\texttt{td}_3\texttt{(1)}$ event were not masked in the rule deriving the event.

11.2.2 *Representing Values of Stateful Objects*

Recall from Section 5.3.1 that whenever a stateful object is modified, its current value is not overwritten but instead merely marked as obsolete and remains, at least conceptually, in the system. To this end, a stateful object is actually determined by a series of successive values whereby each value is associated with a unique key and only valid within a certain time interval.

Each of these values is represented in Dura_C by means of `created` and `terminated` events whose schema is illustrated in Listing 11.23. Thereby, the unique key of a value and its actual data is located in the `pl` attribute. Moreover, the validity of a value is determined by the `semantic-time` of the two events, more precisely, the value is valid from the `semantic-time` of the `created` event through the `semantic-time` of the `terminated` event. Note, however, that as long as a certain value is valid it is merely represented by a `created` event.

Both attributes, `semantic-time` and `reception-time.end`, are distinguished as progressing attributes in the `where` part of the schema. Moreover, the value of `semantic-time` is bounded above and below by the value of `reception-time.end` and the duration `min-dist`.

11.2.3 *Translating Stateful Object Queries*

Queries for stateful objects are simply translated by replacing them with queries for the respective `created` and `terminated` events and by adapting the conditions in the `where` part appropriately.

The scheme for the translation of queries for stateful objects is illustrated in Listing 11.25 whereas the translation of conditions is illustrated in Table 11.2. Be aware that the query for `terminated` events is negated in Listing 11.25 which affects the translation of the temporal dependencies. The translation of conditions is thereby deliberately restricted to the two temporal conditions *`valid-at`* `«tp»` and *`valid-during`* `«ti»` here. For a generalization that covers temporal conditions built from formulas based on the terms `cre(s)` and `term(s)` refer to Section 13.2.

```
                    s ≡ s$cre
              s.«path» ≡ s$cre.pl.«path»

    s valid-at «tp» ≡ and{ s$cre.semantic-time < «tp»,
                           s$term.semantic-time < «tp» }
s valid-during «ti» ≡ and{ s$cre.semantic-time < end(«ti»),
                           s$term.semantic-time < begin(«ti»)}
```

Table 11.2: Translation of Temporal Conditions on Stateful Objects

Note that the translations of *`valid-at`* and *`valid-during`* both specify an upper bound on the progressing attribute `semantic-time` of the negatively queried `terminated` event. Thus, to enable the incremental evaluation of queries for stateful objects it is for instance sufficient to specify a `«tp»` or a `«ti»` that are depending on the time of an event that is positive in `«queries»`. For further details on how to facilitate the (incremental) evaluation of queries check for Section 5.2.5.

11.2.4 *Translating Reactive Rules*

Actions modifying stateful objects are translated in the following way. A special translation for reactive rules triggering such actions is applied that yields rules deriving `update$requested` events on the invocation of an update action. Thereby it is assumed that the query for the stateful object that is to be modified is directly contained in the body of the respective reactive rule. This limitation on the structure of rules is required to keep the effect of falsely derived `update$requested`

Listing 11.23: Complete Schema of `created` and `terminated` Events

```
«name»$«created or terminated»{
  id{identifier}
  semantic-time{timestamp}
  reception-time{ begin{timestamp}, end{timestamp} }
  pl{ id{identifier}, «attributes» }
} where {
  progressing(id),
  progressing(semantic-time),
  progressing(reception-time.end),

  reception-time.begin <= reception-time.end,
  reception-time.end - min-dist <= semantic-time,
  semantic-time <= reception-time.end
}
```

Listing 11.24: A Generic Stateful Object Query

```
and{
  «queries»

  state s: «name»{ «attributes» }
} where { «conditions» }
```

Listing 11.25: Translation of the Query from Listing 11.24

```
and{
  «queries»

  event s$cre: «name»$created{ pl{ «attributes» } }
  not event s$term: «name»$terminated{ }
} where {
  s$cre.pl.id = s$term.pl.id

  «translation of conditions»
}
```

events local to internal rules so that false positives can be detected and thus do not cause an effect that is visible to programmers.

Accordingly, reactive rules modifying stateful objects as they are considered by this transformation have the syntactical shape as illustrated in Listing 11.26. The translation of these reactive rules to deductive rules is illustrated in Listing 11.27. Thereby, the translation of queries of stateful objects discussed in Section 11.2.3 is applied yielding «translation of queries» and «translation of conditions» to translate further queries to stateful objects that are contained in «queries». Note that the rule deriving terminate$requested events is obtained by simply changing the event name of the rule deriving update$requested events and by omitting the parameter set.

Note how the rule in Listing 11.27 determines values of the attributes query-time, semantic-time, and reception-time.end by means of the meta variables «query time», «semantic time», and «reception time». Thereby, with $e_1,\ldots,e_k$ corresponding to the event identifiers positive in «translation of queries», the meta variable «query time» is determined by

```
greatest(end(e1), ..., end(ek), «tp»)
```

or

```
greatest(end(e1), ..., end(ek), begin(«ti»), end(s$cre))
```

depending on whether *valid-at* «tp» or *valid-during* «ti» is used in the temporal dependencies to constrain the validity of the stateful object s.[4] Moreover, semantic-time is determined by

```
greatest(«query time», «request time»)
```

and «reception time» is determined by

```
greatest(«semantic time», end(s$cre)+min-dist).
```

In this way, the «query time» corresponds to the time the query in the body matches and the update is invoked. It thus corresponds to the time the key of the currently valid value of the stateful object that is to be updated is obtained. The «semantic time» corresponds to the point in time at which the update action is deemed to be initiated. Note that it corresponds to the «query time», unless the initiation is further deferred by «request time», which can only be caused by the temporal conditions of a complex action. Finally, «reception time» basically corresponds to the semantic time but may be shifted to maintain a minimal duration of min-dist to the occurrence of the queried created event. Accordingly, derived updated events, and also the resulting terminated events, may occur at «semantic time». Thus, this time determines the end of the time interval in which terminated events need to be masked.

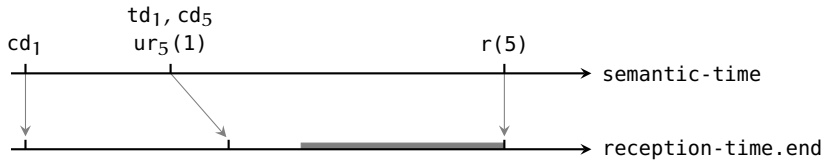
4 Note that the first case is actually already covered by the second case if one determines begin(«tp») = «tp».

Listing 11.26: Reactive Rules Triggering Stateful Object Actions

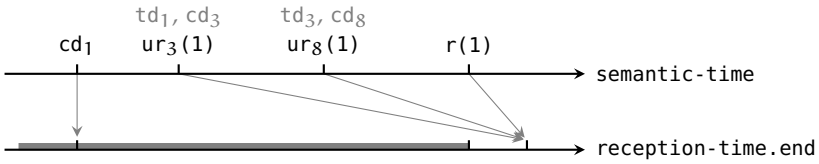
```
ON
  «queries»
DO
  action a: «name»$create{
    request-time{«request time»}, «attributes»
  }
END

ON
  and{
    «queries»
    state s: «name»{ «query terms» }
  } where { «conditions» }
DO
  action a: «name»$update{
    query{id(s)}, set{«attributes»}, request-time{«request time»}
  }
END
```

Listing 11.27: Translation Scheme for Rules in Listing 11.26

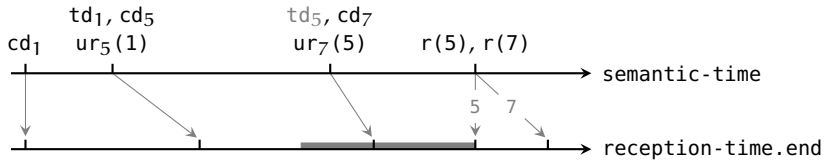
```
DETECT
  «name»$create$requested{
    semantic-time{«request time»}
    reception-time{ end{«request time»} }
    pl{ id{sequence.next()}, «attributes» }
  }
ON
  «translation of queries»
END

DETECT
  «name»$update$requested{
    query-time{«query time»}
    semantic-time{«semantic time»}
    reception-time{ end{«reception time»} }
    pl{ query{s$cre.pl.id}, set{«attributes»}, id{sequence.next()} }
  }
ON
  and{
    «translation of queries»

    event s$cre: operation-mode$created{ pl{ «query terms» } }
    not event s$term: operation-mode$terminated{}
  } where {
    «translation of conditions»

    s$cre.pl.id = s$term.pl.id,
    end(s$cre) <= «query time»,
    end(s$term) <= «query time» - min-dist
  }
END
```



(a) `min-dist` Exceeded: `end(r(5)) - end(cd5) > min-dist`



(b) `min-dist` not Exceeded: `end(r(1)) - end(cd1) <= min-dist`



(c) `min-dist` Exceeded and not Exceeded: `end(r(5)) - end(cd5) > min-dist` and `end(r(7)) - end(cd7) <= min-dist`

Figure 11.4: Possible Situations for the Derivation of Update Requests

Figure 11.4 illustrates the rule deriving `update$requested` events from Listing 11.27 by means of two different situations that may arise, namely, whether the time difference between the «`semantic time`» and the end of the reception time of the referenced `created` event exceeds or falls below `min-dist`. Thereby, the index of each label corresponds to the `pl.id` attribute of the respective event and the number inside the brackets refers to the `pl.id` attribute of the `created` event representing the value that is to be updated. The gray area illustrates the time interval that ends at «`semantic time`» and goes backward for a duration of `min-dist`. Accordingly, in the gray area `terminated` events are masked for the query of the rule deriving `update$request` events.

Note that in Figure 11.4b the `update$requested` event `r(1)` should actually reference cd_8 rather than cd_1 and likewise $ur_8(1)$ should reference cd_3. However, the end of the reception time of cd_8 exceeds «`query time`» and thus it does not match the temporal conditions specified in the body of the rule deriving `update$requested` events. Moreover, note that in Figure 11.4c both situations apply for the same `update` instance and that the derived `update$requested` event `r(5)` is a false positive as the validity of cd_5 has already been terminated by td_5. Figure 11.4a is actually free of special cases.

11.2.5 *Deriving Succeeded Events*

To begin with, we focus on the derivation of `update$succeeded` events and assume that updates are not issued concurrently. Aspects related to the conflict resolution and to the derivation of `create$succeeded` and `terminate$succeeded` events are discussed subsequently.

Note that, with the exception of the translation of the conflict resolution, all following rules are independent of any user defined queries and rules and thus apply for the translation of any stateful object. Merely the prefix of the event names differs for the translation of different stateful object, but the rest of the rules remains unaffected.

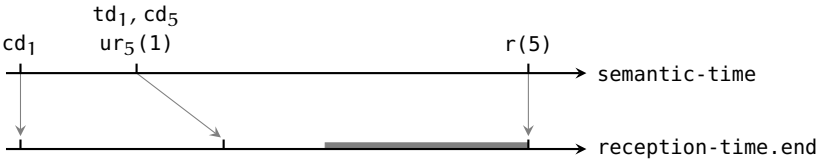
DERIVING UPDATE SUCCEEDED EVENTS To determine whether an `update$requested` event is a true or false positive two different cases are distinguished, namely, whether the time difference of the end of the reception time of the `update$request` and the referenced `created` event exceeds `min-dist` or not. The two situations are illustrated in Figure 11.5 and Figure 11.6. This time, the gray area illustrates the time interval that ends at `end(r)` and reaches backwards for a duration of `min-dist`. Accordingly, in the gray area `created` as well as `terminated` events need to be masked, as `update$requested` events potentially entail `created` and `terminated` events with a reception time that coincides with `end(r)`.

Figure 11.5 illustrates a situation where the time difference exceeds `min-dist`, that is, `end(r(5)) - end(cd`$_5$`) > min-dist`. By looking at the `created` and `terminated` events, it is obvious that the update request `r(5)` is a true positive request in Figure 11.5a whereas it is a false positive request in Figure 11.5b and in Figure 11.5c. However, in the gray area `terminated` events are masked and thus the information provided by the `terminated` events needs to be deduced from the available `terminate$request` and `update$request` events. Accordingly, an update request `r(i)` is a true positive request if there is no `td`$_i$ with `end(td`$_i$`) + min-dist <= end(r(i))` and if neither a `tr`$_i$ nor a `ur`$_i$ occurs with

```
end(r(i)) - min-dist <= end(tr_i), end(tr_i) <= end(r(i))
end(r(i)) - min-dist <= end(ur_i), end(ur_i) <= end(r(i))
tr_i.semantic-time <= r(i).semantic-time
ur_i.semantic-time < r(i).semantic-time
```

Note that the case `ur`$_i$`.semantic-time = r(i).semantic-time` is subject to the conflict resolution which is subsequently addressed.

Figure 11.6 illustrates a situation where the time difference falls below `min-dist`, that is, `end(r(1)) - end(cd`$_1$`) <= min-dist`. In this situation, the initiation of the `update` action occurs before any `created` and `terminated` event referring to `cd`$_1$, at least with respect to the end to the respective receptions times. Accordingly, in this situation `ur(1)` actually refers to the `created` event with the largest semantic time that is smaller than its own, that is, `ur(1)` actually identifies `ur(8)`. However,

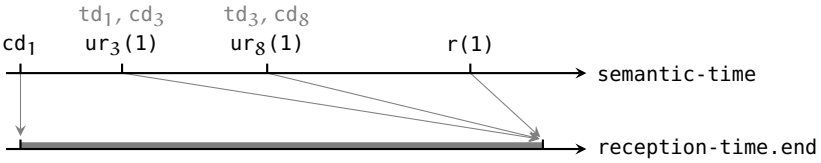


(a) True Positive Update Request `r(5)`

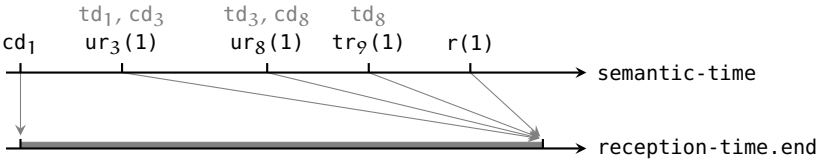
(b) False Positive Update Request `r(5)`

(c) False Positive Update Request `r(5)`

Figure 11.5: `min-dist` Exceeded: `end(r(5))` - `end(cd5)` > `min-dist`



(a) True Positive Update Request `r(1)`



(b) False Positive Update Request `r(1)`

Figure 11.6: `min-dist` not Exceeded: `end(r(1))` - `end(cd1)` <= `min-dist`

in Figure 11.6b the validity of cd_8 is already terminated by tr_9(1), which actually identifies tr_9(8). Thus ur(1) is a true positive request in Figure 11.6a whereas it is a false positive request in Figure 11.6b. Accordingly, an update request r(i) is a true positive request if there is no td_i with end(td_i) + min-dist <= end(r(i)) and if no tr(i) occurs with

```
end(r(i)) - min-dist <= end(tr_i) <= end(r(i))
tr_i.semantic-time <= r(i).semantic-time
```

Note that these conditions are actually included in the conditions of the previous case. Combining the elaborated aspects yields the deductive rule in Listing 11.28 deriving update$succeeded events for each true positive ur event.

CONFLICT RESOLUTION In case two updates are issued concurrently, the conflict resolution specified in the on conflict select part associated with the schema of a stateful object applies.

Eg, the conflict resolution of the stateful object operation-mode coincides with on conflict select { max mode, min id }. Accordingly, when two updates are issued concurrently the one with the higher mode and, in case both modes are equal, with the lower id is selected, which translates to the following condition.

```
or{
  ur.pl.mode > r.pl.mode,
  and{ ur.pl.mode = r.pl.mode, ur.pl.id < r.pl.id }
}
```

Be aware that the query for update$requested events is negated in Listing 11.28 which needs to be taken into account for the specification of conditions. In analogy, the generic conflict resolution

```
on conflict select { c_1 p_1, ..., c_k p_k }
```

translates to the condition

```
or{
  ur.pl.p_1 r_1 r.pl.p_1,
  and{
    ur.pl.p_1 = r.pl.p_1,
    or{
      ur.pl.p_2 r_2 r.pl.p_2,
      and{ ...,
        or{
          ur.pl.p_k-1 r_k-1 r.pl.p_k-1,
          and{ ur.pl.p_k-1 = r.pl.p_k-1, ur.pl.p_k r_k r.pl.p_k }
} ... }
```

whereby r_i is determined by the relation > if c_i coincides with max and by < otherwise.

If the schema of the corresponding stateful object specifies a conflict resolution for concurrent updates, the respective condition needs to be substituted in Listing 11.28 for false.

Listing 11.28: Deductive Rule Deriving update$succeeded Event

```
DETECT
 «name»$update$succeeded{
  semantic-time{r.semantic-time}
  payload{r.payload}
 }
ON
 and{
  event r: «name»$update$requested{},

  event cd: «name»$created{},
  not event td: «name»$terminated{},

  not event tr:  «name»$terminate$requested{}

  not {
   event ur:  «name»$update$requested{}
  } where or{
   and{ end(r) > end(cd)+min-dist, ur.semantic-time < r.semantic-time }
   and{ ur.semantic-time = r.semantic-time, false }
  }
 } where {
  end(cd) <= end(r) - min-dist,
  end(td) <= end(r) - min-dist,
  end(r) - min-dist <= end(tr), end(tr) <= end(r),
  end(r) - min-dist <= end(ur), end(ur) <= end(r),

  cd.payload.id = td.payload.id,
  cd.payload.id = r.payload.query,

  tr.payload.query = r.payload.query,
  tr.semantic-time <= r.semantic-time,

  ur.id != r.id,
  ur.payload.query = r.payload.query,
 }
END
```

DERIVING REMAINING SUCCEEDED EVENTS The deductive rule deriving `create$succeeded` events is straight forward, as it does not involve recursion and `create` actions cannot fail. The respective rule just derives `create$succeeded` events whenever a `create$requested` event occurs. For the sake of completeness, the generic scheme of the rule is specified in Listing 11.29.

The basic approach for the derivation of `terminate$succeeded` events is equivalent to the one for `update$succeeded` events. Consequently, both rules are very similar and differ only in two minor aspects. First, if two `terminate` actions are issued concurrently, both actions succeed, as the respective value is indeed terminated, and thus a conflict resolution as for `update` actions is not required. Therefore, the condition `tr.semantic-time <= r.semantic-time` from Listing 11.28 is replaced by `tr.semantic-time < r.semantic-time` in Listing 11.30. Second, the conflict resolution for `update` actions is omitted, as it is only relevant if there is not at least one `update$requested` events. If there are multiple, it is irrelevant which of them is successful as at least one of them is successful and therefore the conditions for the conflict resolution can be safely omitted.

11.2.6 *Deriving Failed Events*

When a `update$requested` event occurs whereas the corresponding `update$succeeded` event remains absent, this can either mean that that the request is a false positive or that the update failed. Failed updates are either caused by concurrently issued updates or if the initiation of the update is delayed.

Both cases can be distinguished by comparing the `query-time` of the `update$requested` event with the `semantic-time` of the corresponding `terminated` event. If the `semantic-time` of the `terminated` event is equal or greater than the `query-time`, then the corresponding `update$request` would have been successful if its initiation was not deferred. However, if the `semantic-time` of the `terminated` is lower than the `query-time` then the `terminated` event occurs in the time interval that is masked and thus the `update$request` event is a false positive. The respective rule is given in Listing 11.31. Note that that in contrast to `update$succeeded` events, `update$failed` events do not entail `created` and `terminated` events and therefore `terminated` events can be queried without further limitations.

The same conditions naturally also apply for `terminate$requested` events. Thus, by substituting `terminate` for `update` one obtains the rule deriving `terminate$failed` events. Moreover, the creation of values of stateful objects cannot fail and thus there is simply no rule deriving `create$failed` events.

Listing 11.29: Deductive Rule Deriving `create$succeeded` Event

```
DETECT
  «name»$create$succeeded{
    semantic-time{r.semantic-time}
    payload{r.payload}
  }
ON
  event r: «name»$create$requested{}
END
```

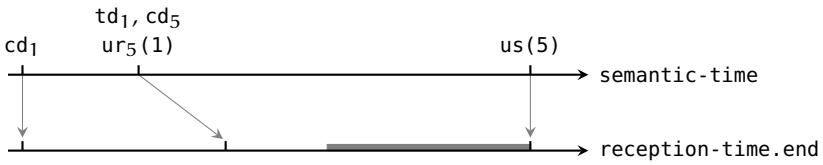
Listing 11.30: Deductive Rule Deriving `terminate$succeeded` Event

```
DETECT
  «name»$terminate$succeeded{
    semantic-time{r.semantic-time}
    payload{r.payload}
  }
ON
  and{
    event r: «name»$terminate$requested{},

    event cd: «name»$created{},
    not event td: «name»$terminated{},

    not event tr: «name»$terminate$requested{}

    not {
      event ur: «name»$update$requested{}
    } where and{
      end(r) > end(cd)+min-dist,
      ur.semantic-time < r.semantic-time
    }
  } where {
    end(cd) <= end(r) - min-dist,
    end(td) <= end(r) - min-dist,
    end(r) - min-dist <= end(tr), end(tr) <= end(r),
    end(r) - min-dist <= end(ur), end(ur) <= end(r),

    cd.payload.id = td.payload.id,
    cd.payload.id = r.payload.query,

    tr.id != r.id,
    tr.payload.query = r.payload.query,
    tr.semantic-time < r.semantic-time,

    ur.payload.query = r.payload.query,
  }
END
```

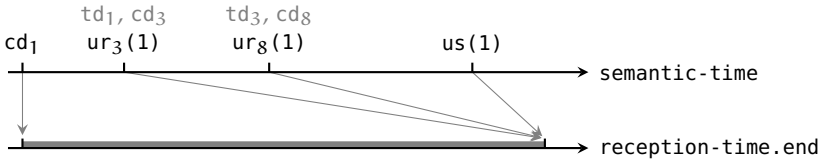
Listing 11.31: Deductive Rule Deriving `update$failed` Events

```
DETECT
  «name»$update$failed{
    semantic-time{r.semantic-time}
    payload{r.payload}
  }
ON
  and{
    event r: «name»$update$requested{}
    not event us: «name»$update$succeeded{}

    event td: «name»$terminated{}
  } where {
    ur.payload.id = us.payload.id,
    ur.payload.query = td.payload.id,

    ur.semantic-time = us.semantic-time,
    td.semantic-time <= ur.semantic-time,

    ur.observation-time <= td.semantic-time
  }
END
```

Once an `update$succeeded` event is derived, the next step is to derive a corresponding `updated` event which contains the old and new values in its payload. As before there are two cases, namely, whether the time difference between the end of the reception time of the `update$succeeded` event and the referenced `created` event exceeds or falls below `min-dist`. The two situations are illustrated in Figure 11.7, whereby the gray area determines the time interval in which `created` and `terminated` events are masked.



(a) `min-dist` Exceeded: `end(us(5)) - end(cd5) > min-dist`



(b) `min-dist` not Exceeded: `end(us(1)) - end(cd1) <= min-dist`

Figure 11.7: Determining the Payload of the Updated Value

In Figure 11.7a, the time difference exceeds the duration `min-dist`, that is, `end(us(5)) - end(cd5) > min-dist`. Accordingly, the referenced `created` event can be queried, which is required to obtain the values that need to be provided in the `old-payload` attribute of the `updated` event.

In Figure 11.7b, the time difference falls below `min-dist`, that is, `end(us(5)) - end(cd5) <= min-dist`, and thus the `created` event cd_5 is masked. However, similar as for the derivation of `update$succeeded` events, the desired information can be equivalently obtained from the latest `update$requested` event that refers to the cd_1 and occurred before `us(1)` with respect to the semantic time.

The respective rules deriving `updated` events are illustrated in Listing 11.32. Note that the key of the `update` action instance is reused to determine the key of the new value which is specified in the `new-payload` attribute. It is therefore no coincidence that in the previous figures the index of `update$requested` and `created` events coincides if they occur at the same instant with respect to the semantic time.

Listing 11.32: Deductive Rule Deriving `updated` Events

```
DETECT
  «name»$updated{
    semantic-time{r.semantic-time},
    old-payload{cd.payload},
    new-payload{
      id{r.payload.id},
      area{r.payload.set.area}, mode{r.payload.set.mode}
    }
  }
ON
  and{
    event r: «name»$update$succeeded{}
    event cd: «name»$created{}
    not event us: «name»$update$succeeded{}
  } where {
    cd.payload.id = r.payload.query,
    cd.payload.id = us.payload.query,

    end(cd) + min-dist <= end(r),
    end(r) - min-dist <= end(us), end(us) <= end(r),

    us.semantic-time < r.semantic-time
  }
END

DETECT
  «name»$updated{
    semantic-time{r.semantic-time},
    old-payload{
      id{us.payload.id},
      area{us.payload.set.area}, mode{us.payload.set.mode}
    }
    new-payload{
      id{r.payload.id},
      area{r.payload.set.area}, mode{r.payload.set.mode}
    }
  }
ON
  and{
    event r: «name»$update$succeeded{}
    event us: «name»$update$succeeded{}
    not event other-us: «name»$update$succeeded{}
  } where {
    r.payload.query = us.payload.query,
    r.payload.query = other-us.payload.query,

    end(r) - min-dist <= end(us), end(us) <= end(r),
    end(r) - min-dist <= end(other-us), end(other-us) <= end(r),

    us.semantic-time < r.semantic-time,
    us.semantic-time < other-us.semantic-time,
    other-us.semantic-time < r.semantic-time
  }
END
```

Listing 11.33: Deductive Rule Deriving `terminated` Events

```
DETECT
  «name»$terminated{ semantic-time{r.semantic-time}, pl{cd.pl} }
ON
  and{
    event r: «name»$terminate$succeeded{}
    event cd: «name»$created{}
    not event ts: «name»$terminate$succeeded{}
  } where {
    end(cd) + min-dist <= end(r),
    end(r) - min-dist <= end(ts), end(ts) <= end(r),

    r.pl.query = cd.pl.id,
    r.pl.query = ts.pl.query,

    r.semantic-time = ts.semantic-time,
    ts.pl.id > r.pl.id
  }
END

DETECT
  «name»$terminated{
    semantic-time{r.semantic-time},
    pl{ area{us.pl.set.area}, mode{us.pl.set.mode}, id{us.pl.id} }
  }
ON
  and{
    event r: «name»$terminate$succeeded{}
    not event ts: «name»$terminate$succeeded{}
    event us: «name»$update$succeeded{}
    not event other-us: «name»$update$succeeded{}
  } where {
    end(r) - min-dist <= end(ts), end(ts) <= end(r),
    end(r) - min-dist <= end(us), end(us) <= end(r),
    end(r) - min-dist <= end(other-us), end(other-us) <= end(r),

    r.pl.query = us.pl.query,
    r.pl.query = ts.pl.query,
    r.pl.query = other-us.pl.query,

    r.semantic-time = ts.semantic-time,
    ts.pl.id > r.pl.id,

    us.semantic-time < r.semantic-time,
    us.semantic-time < other-us.semantic-time,
    other-us.semantic-time < r.semantic-time
  }
END

DETECT
  «name»$terminated{ semantic-time{ud.semantic-time}, pl{ud.old-pl} }
ON
  event ud: «name»$updated{}
END
```

11.2.8 *Deriving Terminated Events*

The basic approach for the derivation of `terminated` events is very similar to the derivation of `updated` events and, except for slightly varying payload, only differs in two aspects.

First, recall from Section 11.2.5 that when two `terminate` actions are issued concurrently, both of them are successful. Nevertheless, only one `terminated` event should be derived. Therefore, the rule in Listing 11.33 picks (more or less randomly) the `terminate$succeeded` event with the larges id to derive the respective `terminated` event.

Second, in contrast to `updated` events, `terminated` events are not only entailed by `terminate$succeeded` events but also by `update$succeeded` events. Accordingly, a third rule is required for the derivation of `terminated` events.

11.2.9 *Deriving Created and Initiated Events*

As `create` actions cannot fail, a `created` event is simply derived whenever a `create$requested` event occurs. Moreover, an `updated` event also entails the creation of a new value of the stateful object and thus a `created` event is derived whenever an `updated` event occurs.

Each action triggered by a reactive rule needs to entail an `initiated` event. To this end, `terminate$initiated` and `update$initiated` events are derived from the respective `succeeded` and `failed` events whereas `created$initiated` events are simply derived on the occurrence of the respective `create$requested` events as the creation of values is always successful.

For the sake of completeness, the rather basic rules are illustrated in Listing 11.34 and Listing 11.35 on the next page.

11.2.10 *Concluding Remarks*

The translation of stateful objects relies on the presence of the duration `min-dist` which is artificially impacting the time of events that are representing stateful objects. Although this does not affect the semantics of queries to stateful objects as the translation is carefully designed to account for the shifted times in a transparent manner, it affects the evaluation of Temporal Stream Algebra (TSA) expressions with respect to the latency that is caused at runtime.[5] As a consequence, programmers need to balance two different effects that are impacted by their choice of `min-dist`.

With the proposed translation, the earliest possible time for the sound evaluation of a (translated) query to a stateful object corresponds to `end(cd) + min-dist`, whereby `cd` refers to the `created` event

5 Details on runtime effects related to the evaluation of TSA expressions by means of Event-Mill are out of the scope of this thesis. For details thereon refer to [BB12a].

Listing 11.34: Deductive Rules Deriving `created` Events

```
DETECT
  «name»$created{
    semantic-time{r.semantic-time},
    payload{r.payload}
  }
ON
  event r: «name»$create$requested{}
END

DETECT
  «name»$created{
    semantic-time{ud.semantic-time}
    payload{ud.new-payload}
  }
ON
  event ud: «name»$updated{}
END
```

Listing 11.35: Deductive Rules Deriving `initiated` Events

```
DETECT
  «name»$create$initiated{
    semantic-time{r.semantic-time}
    payload{r.payload}
  }
ON
  event r: «name»$create$requested{}
END

DETECT
  «name»$terminate$initiated{
    semantic-time{d.semantic-time}
    payload{d.payload}
  }
ON
  or{
    event d: «name»$terminate$succeeded{}
    event d: «name»$terminate$failed{}
  }
END

DETECT
  «name»$update$initiated{
    semantic-time{d.semantic-time}
    payload{d.payload}
  }
ON
  or{
    event d: «name»$update$succeeded{}
    event d: «name»$update$failed{}
  }
END
```

representing a matching value. In consequence, the evaluation of queries for stateful objects is deferred by `min-dist`.[6] Accordingly, to obtain a small latency for the evaluation of queries, `min-dist` should be chosen relatively small.

In contrast, the Event-Mill engine, which is used to evaluate TSA expressions, continuously triggers the evaluation of expressions contributing to the derivation of `created` and `terminated` events in intervals of size `min-dist`. Hence, the smaller `min-dist` is chosen the more often the Event-Mill engine needs to trigger the evaluation of expressions. Accordingly, to keep the base load of the engine low, `min-dist` should be chosen relatively large.

All in all, the duration `min-dist` needs to be chosen as small as possible to keep the latency for the evaluation of queries to stateful objects low but still large enough so that the base load of the evaluation engine remains reasonable. Consequently, there is no generic value for `min-dist` that applies similarly for any setting. Hence, the duration `min-dist` needs to be adapted empirically for each individual rule set based on a characteristic sample of the expected event stream and the performance of the machine running Event-Mill. It defaults to one second, though, which is still tolerable for Emergency Management (EM) purposes and yet still reasonable even for current desktop computers.

6 By borrowing the query of the rule deriving `update$succeeded` events, it seems feasible to obtain a translation of queries for stateful objects that defers only queries issued between `end(cd)` and `end(cd)+min-dist`.

12 IMPLEMENTATION PROTOTYPE

The evaluation or rather execution of Dura programs is based on the Event-Mill engine [BHB12], which is capable of evaluating Temporal Stream Algebra (TSA) expressions that are obtained by means of the translation described in Chapter 10 and Chapter 11. The Event-Mill engine is designed and implemented by Simon Brodt, a colleague of this dissertation's author. For the theoretical details on the evaluation of TSA refer to [BB12a].

Following is an illustration of the implementation of the Dura compiler and of the compilation process which focuses on the basic set up of the Event-Mill engine and the evaluation of Dura programs. Moreover, we briefly introduce a convenient and full-fledged Dura editor that seamlessly integrates with the Eclipse IDE [Eclb].

12.1 A PRAGMATIC MODULE MECHANISM

To be suitable for large rule sets that may easily occur in practice, the Dura compiler is based on a pragmatic module mechanism for Dura that facilitates the separation of rule sets into modules of manageable size.

Thereby, the basic ideas for modules in Dura are borrowed from object-oriented languages, such as, Java [Oraa]. In Java, namespaces are used to avoid name conflicts between different classes. Moreover, member variables and functions associated with a certain class are are specified in a common class definition. Likewise, in Dura, the schema of a stream and rules that reference the stream in their head are specified in a common stream definition and module names are used to avoid name conflicts between streams.

The following description of the module mechanism focuses on aspects that are relevant for the compilation of Dura programs, in particular on stream definitions. Further aspects, such as, information hiding, are elaborated in Hausmann, Brodt, and Bry [HBB12].

12.1.1 *Stream Definitions*

Events, stateful objects, and action are specified by means of stream definitions that contain the respective stream schema in combination with rules that are related to the corresponding stream.

```
EVENT «event schema» WITH «deductive rules» END
STATE «stateful object schema» END
ACTION «action schema» WITH «complex action rules» END
```

Thereby the `WITH` part is optional and may be omitted, eg, to specify the schema of events that are received by the Event Processing System (EPS) from external sensors.

Within each module, the type of each stream definition must be unique, that is, there must not be two definitions for the same type. Accordingly, all rules that derive a particular event type or specify a particular complex action are specified in a single stream definition. This seem convenient as all essential properties of a stream are determined in a single part of the program and cannot be scattered among a (potentially huge) program.

12.1.2 *Stream Modifier*

Different modifiers are available that are specified prior to the stream definition they apply for. They determine which events are read from or sent to the Event Service Bus (ESB) connected to the Event-Mill engine and whether a stream is static.

Thereby, the modifiers are mainly intended for event streams. Modifies specified for stateful objects and actions are transferred to the implicit events that are used for their internal representation.

BUFFER TYPE There are three different buffer types that affect how the Event-Mill engine processes and stores event internally, namely, *`input`*, *`output`*, and *`log`*.

The modifiers *`input`* and *`output`* specify that events are read from and sent to the ESB connected to the Event-Mill engine, respectively. Moreover, *`log`* specifies that all events of a certain type are store in a particular database relation where they are *not* subject to garbage collection.

STATIC If resent, the *`static`* modifier determines that no events can be added to a stream after a Dura program has been initialized. As a consequence, the attributes of static streams cannot be identified as progressing attributes.

Note that it is thus reasonable to use *`static`* only in conjunction with *`input`*. Static streams cannot be populated by means of rules at runtime and thus their content needs to be provided as input in the initiation phase of the engine.

Note that the module system can be naturally extended by means realizing information hiding, for instance, by *`public`* and `privat` modifiers. These and further extensions are elaborated in [HBB12] but are not implemented by the current version of the compiler.

Filename: `metro.operation-mode.modes.dura`

```
MODULE metro.operation-mode.modes

CONST OPM_NORMAL = 0
CONST OPM_EXCEPTIONAL = 1
CONST OPM_EMERGENCY = 2
CONST OPM_EMERGENCY_MAJOR = 3
```

Filename: `metro.operation-mode.dura`

```
MODULE metro.operation-mode

input
STATE
  operation-mode{ area{long}, mode{int} }
  on conflict select { max mode, min id }
END

output
EVENT
  operation-mode-escalation{ area{long}, mode{int} }
WITH
  DETECT
    operation-mode-escalation{ area{var A}, mode{var Mnew} }
  ON
    event e: operation-mode$updated{
      old-payload{ area{var A}, mode{var Mold} },
      new-payload{ area{var A}, mode{var Mnew} }
    } where { var Mold < var Mnew }
  END
END
```

Filename: `metro.alarm-detection.dura`

```
MODULE metro.alarm-detection

IMPORT metro.operation-mode.modes.*
IMPORT metro.operation-mode.operation-mode-escalation

output
EVENT
  uncertain-alarm{ area{long} }
WITH
  DETECT
    uncertain-alarm{ area{var A} }
  ON
    event e: operation-mode-escalation{ new-payload{ area{var A} } }
  END

  ...
END
```

Figure 12.1: An Exemplary Module Structure

12.1.3 *Modules*

A module is determined by a unique name, a set of stream and constant definitions, and a set of reactive rules.

The module name is indicated in the beginning of each file by means of the `MODLUE` keyword followed by the respective name of the module. Thereby, periods can be used to realize a hierarchical nesting of modules. A module can be scattered across different files and there are no regulations on how to store the files on the file system. However, it seems reasonable to use only one file per module and to reflect the nesting of modules either in the respective filenames or in the file system structure.

Within a module, definitions can be referenced by the respective type they specify. Definitions from other modules need to be referenced by their full qualified name, composed from the module name and their basic name, unless the definition is imported by means of the `IMPORT` statement. For instance, the constant of `OPM_NORMAL` from Figure 12.1 can always be referenced by its full qualified name, ie, `metro.operation-mode.modes.OPM_NORMAL`. If the definition is imported properly by either of the following statements it can be referenced by its basic name `OPM_NORMAL`.

```
IMPORT metro.operation-mode.modes.OPM_NORMAL
IMPORT metro.operation-mode.modes.*
```

Thereby, $*$ imports all definitions of the respective module.

12.2 THE DURA COMPILER

The Dura compiler and its sources are integrated into the Event-Mill release available from

`http://www.pms.ifi.lmu.de/cep/releases/`

The compiler takes as input a set of source files and yields a set of TSA expressions that can be evaluated by means of the Event-Mill engine. To this end, the compiler basically applies the transformations and normalizations that are discussed in Chapter 10 and Chapter 11.

12.2.1 *Source Code Overview*

The compiler is written in Java [Oraa] and all relevant classes are located in the package `de.lmu.ifi.pms.cep.durang.compiler` and the following subpackages:

- The package `analysis` contains classes for the static analysis of complex actions.

- The package `transformations` contains classes for the translation of Dura to Dura_C.
- The package `tsa` contains classes for the translation of Dura_C to TSA.
- The package `exceptions` contains Dura specific exceptions that are related to the semantic analysis of programs, eg, type checks and the static analysis of actions.
- The package `utils` contains auxiliary classes that provide means to traverse and modify the data structures that model Dura in addition to means for debugging and unit testing purposes.

The actual compilation is realized by the classes `DuraCompiler` and `DuraCComplier` which are based on the functionality of the given packages.

12.2.2 *Compilation Phases*

The compilation process is split into several phases. Initially, the Dura code is parsed by means of the Xtext parser [Xte]. Subsequently, the static analysis is applied to the complex actions. Unless the analysis fails, the Dura program is translated to an equivalent Dura_C program which is eventually translated to a set of TSA expressions.

PARSING The parsing of Dura source code is realized by means of the Xtext parser [Xte]. The parser receives multiple Dura source files as input and yields an Abstract Syntax Tree (AST) in form of an instantiated EMF model [EMF] as output.

Moreover, the Xtext parser identifies and reports syntactical errors, such as, references to undefined variables or streams and references to non-existing attributes of streams, and validates the weak range restriction of rules.

A simplified version of the applied Xtext grammar is included in Appendix A.

STATIC ANALYSIS The class `ComplexActionAnalysis` accomplishes the static analysis of complex actions as it is described in Section 8.3.

Thereby the verification of the compliance of complex actions relies on functionality that is implemented by the analysis of TSA expressions. More precisely, the entailment of formulas that indicates the compliance of complex actions is verified by means of the on the `GenericFormulaAnalyser` class which basically implements the method described in [BB12a].

TRANSLATING DURA TO DURA$_C$ The class `DuraCompiler` realizes the transformation of a Dura program to a Dura$_C$ program as it is described in Chapter 11.

To this end, the AST of the parsed Dura program is modified by means of the following classes that are part of the `transformations` package:

- The class `ActionCompositonTransformation` contains methods for the translation of action invocations of complex action rules as it is described in Section 11.1.4.
- The class `ActionStatusQueryTransformation` contains methods for the translation of status queries of complex action rules and atomic actions as it is described in Section 11.1.5.
- The class `ReactiveRuleTransformation` contains methods for the translation of reactive rules as it is described in Section 11.1.8.
- The class `StatefulObjectQueryTransformation` contains methods for the translation of queries of stateful objects to event queries as it is described in Section 11.2.3.

TRANSLATING DURA$_C$ TO TSA The AST resulting from the previous phase is subsequently translated to a set of TSA expressions by means of the `DuraCCompiler` class.

To this end, the Dura$_C$ rules are normalized as it is described in Section 10.3 and subsequently the normalized AST is translated to TSA expressions by means of the following auxiliary classes that are contained in the `tsa` package:

- The class `StreamCompiler` contains methods for the translation of Dura streams to TSA streams as it is described in Section 10.4.1.
- The two classes `ExpressionCompiler` and `FormulaComplier` contain methods for the translation of Dura formulas, eg, from query supplements, to TSA formulas as it is described in Section 10.4.3.
- The class `QueryComiler` contains methods for the translation of atomic and conjunctive event queries as it is described in Section 10.4.2 and Section 10.4.6.
- The class `SupplementCompiler` contains methods for the translation of query supplements as it is described in Section 10.4.4.
- The class `RuleCompiler` contains methods for the translation of deductive rules as it is described in Section 10.4.7.

12.2.3 *Manual Compilation*

The Dura compiler is integrated into the Event-Mill engine where it is automatically applied when Dura programs are initiated for evaluation. Nevertheless, for debugging purposes it is convenient to merely compile a Dura program and to skip its evaluation.

To this end, the method `DuraTestSuite.compile(String...)` is available which provides information on every compilation phase. The method takes as input a set of files containing a Dura program and outputs the Dura_C translation of the Dura program in addition to the normalization of the Dura_C program. Moreover, the resulting TSA expressions are output and the static analysis of complex actions as well as the temporal analysis of TSA expressions is carried out.

12.3 EVALUATION OF DURA PROGRAMS

The evaluation of Dura programs by means of the Event-Mill engine is realized either by means of a command line interface to Event-Mill or by means of a Java API.

Both means are equally expressive, in fact, the command line interface is realized by means of the API, but intended for different purposes. The command line interface is suited for humans to interactively control and explore the evaluation of programs whereas the API is suited to connect generic ESBs to the Event-Mill engine.

For the sake of simplicity, we only present the command line interface in the following. For details on the Java API refer to [BHB12].

12.3.1 *Event-Mill Setup*

The Event-Mill engine requires Java 6 or higher and a relational database, namely, MonetDB [Mon], PostgreSQL [Pos], H2 [H2], or HyperSQL [HSQ], to be installed.

The current Event-Mill release is available from the website

`http://www.pms.ifi.lmu.de/cep/releases/`

which offers zip archives containing an executable jar file, all required libraries, and various Dura example programs. Each archive furthermore contains the source code of Event-Mill and the Dura compiler which are provided under the terms of the Eclipse Public License [EPL].

The engine is set up by downloading and extracting the most recent `event-mill-and-examples-X.X.X.zip` archive from the website. Subsequently, the database connection, in particular the user credentials and the database name, need to be configured by adapting the configuration file `event-mill.configuration.xml` as well as the configuration file `event-mill.io.configuration.xml`. Both files are included in the

archive and contain detailed templates for the currently supported databases.

Establishing the database connection is basically sufficient to completely set up the engine. Yet, various further parameters can be adjusted in the configuration files which substantially affect the compilation and evaluation of programs. However, these aspects are not further elaborated here, for details refer to [BHB12].

12.3.2 *Event-Mill Command Line Interface*

The command line interface serves two different purposes. First and foremost, it facilitates the compilation and evaluation of Dura programs. In addition, the interface allows to interactively send events to the engine and to read derived events from the engine.

Both aspect are illustrated in the following by means of the Dura program from Figure 12.2. Note that its source code and the source code of more sophisticated sample programs are included in the Event-Mill release in the `examples` directory.

COMPILATION AND INSTANTIATION To open the command line interface, simply run the Event-Mill jar file from the archive. Note that we assume in the following that the `java` command is executed from the directory the archive has been extracted to.

```
$ java -jar event-mill-X.X.X.jar
```

To begin with the Dura program needs to be compiled and properly instantiated. To this end, the variable `source` is assigned the file containing the Dura program and the variable `examples.dir` is assigned the `examples` directory from the Event-Mill release which contains required auxiliary files.

```
> set %{examples.dir} = 'examples'
> set %{source} = 'examples/simple-access-control.dura'
```

Subsequently, the compilation is accomplished by the following command.

```
> inject %{examples.dir}/init-example.es
```

If no compilation errors occur and the instantiation of the program was successful, the engine replies with a listing of instantiated programs. Thereby, further previously successfully compiled and instantiated programs may be listed as well.

```
The following initialized programs are available:
example.simple-access-control:test
...
```

Eventually, the program instance is loaded and finally set up for execution by means of the `run` command.

```
> run example.simple-access-control:test
```

Filename: `simple-access-control.dura`

```
MODULE example.simple-access-control

input
EVENT
  request-access{ person{string} }
END

static input
STATE
  staff{ person{string} }
END

output log
EVENT
  deny-access{ person{string} }
WITH
  DETECT
    deny-access{ person{var P} }
  ON
    and{
      event e: request-access{ person{var P} },
      not state s: staff { person{var P} }
    }
  END
END

output log
EVENT
  grant-access{ person{string} }
WITH
  DETECT
    grant-access{ person{ var P} }
  ON
    and{
      event e: request-access{ person{ var P} },
      not event f: deny-access{ person{ var P} }
    } where { end(e) - 5sec <= end(f), end(f) <= end(e) }
  END
END
```

Figure 12.2: A Simple Dura Program

INITIATION OF STATEFUL OBJECTS At this point, initial data for stateful objects, in particular for static stateful objects, can be provided. To this end, the respective values, specifying the full qualified name of the stateful object, are parsed and subsequently sent to the engine.

```
> parse example.simple-access-control.staff{ person{ "Alice" } }
> parse example.simple-access-control.staff{ person{ "Bob" } }
> write
```

Note that thereby the whitespaces, in particular those before and after braces, are mandatory and cannot be omitted. Moreover, attributes need to be specified according to their lexicographical order.

Finally, the compiled program is initiated and ready for execution.

```
> init
```

Note that from now on, input for static streams is ignored by the engine.

EVALUATION OF PROGRAMS The program has been successfully loaded and the values of the static stateful object `staff` have been initialized. The next step is to start continuous evaluation for the respective queries.

```
> start
```

At this point, events that are marked as *`input`* can be sent to the engine. Similar to before, events are specified by means of their full qualified name and sent to the engine by means of the `parse` and `write` commands.

```
> parse example.simple-access-control.request-access{ person{ "Bob" } }
> parse example.simple-access-control.request-access{ person{ "Eve" } }
> write
```

Derived events and actions with an *`output`* modifier can be printed on the command line interface using a combination of `read` and `print`.

```
> read
> print
```

The prior command loads all newly derived events into an internal buffer whereas the latter outputs and removes all events of the buffer. In the given example, this causes the following output.

```
Read example.simple-access-control.grant-access{
  id{ 58 }, person{ "Bob" },
  reception-time{ begin{ 1365127620001 }, end{ 1365127620001 } }
} from message bus.
Read example.simple-access-control.deny-access{
  id{ 59 }, person{ "Eve" },
  reception-time{ begin{ 1365127620001 }, end{ 1365127620001 } }
} from message bus.
```

Finally, the execution of the engine is stopped and the command line interface is terminated by means of the commands `stop` and `quit`.

```
> stop
> quit
```

Further commands are available which are in particular required to compile Dura programs that are scattered over multiple files. For thorough details on the command line interface and its commands refer to [BHB12].

12.3.3 *Executing Sample Sessions*

In addition to the interactive conduction of sessions as it is described in the previous section, entire predefined sessions can be automatically executed. In fact, each example program in the `example` folder comes with such an example session.

To this end, each Dura program comes with two additional files, eg, `simple-access-control.init.es` and `simple-access-control.es`. The prior file contains commands that are used to initialize the stateful objects whereas the latter provides the commands that parse events and send them to the engine and reads events from the engine. Note that the commands that are provided by both files almost exactly coincide with the ones that are specified in an interactive session. Merely, the `pause «duration»` command, causing a delay of `«duration»` milliseconds, is additionally used to realize a certain timing of the injected events.

For instance, the sample session for the simple access control example, is executed by means of the following three commands.

```
> set %{examples.dir} = 'examples'
> set %{source} = '%{examples.dir}/simple-access-control.dura'
> inject %{examples.dir}/run-example.es
```

Note that thereby only the last command differs from the compilation of the program.

The sample session causes the exact same command sequence as in the previous example. Thus similar events, only differing in their key and reception time are output.

12.4 THE DURA EDITOR

The Dura Editor is realized as a plugin for the popular Eclipse IDE [Eclb] which is generated from the formal grammar by means of the Xtext framework [Xte]. The plugin is available from the update site

`http://www.pms.ifi.lmu.de/cep/plugins/duraeditor/`

and can be conveniently installed as Eclipse plugin by means of the "Install New Software" wizard as it is described in [Ecla].

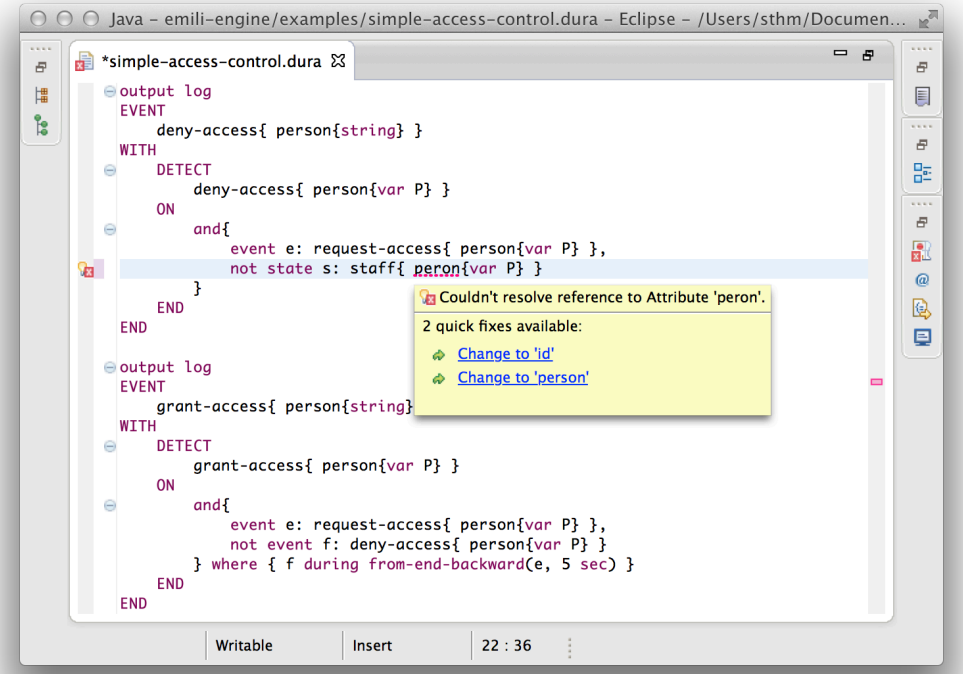


Figure 12.3: Validation and Quick Fixes

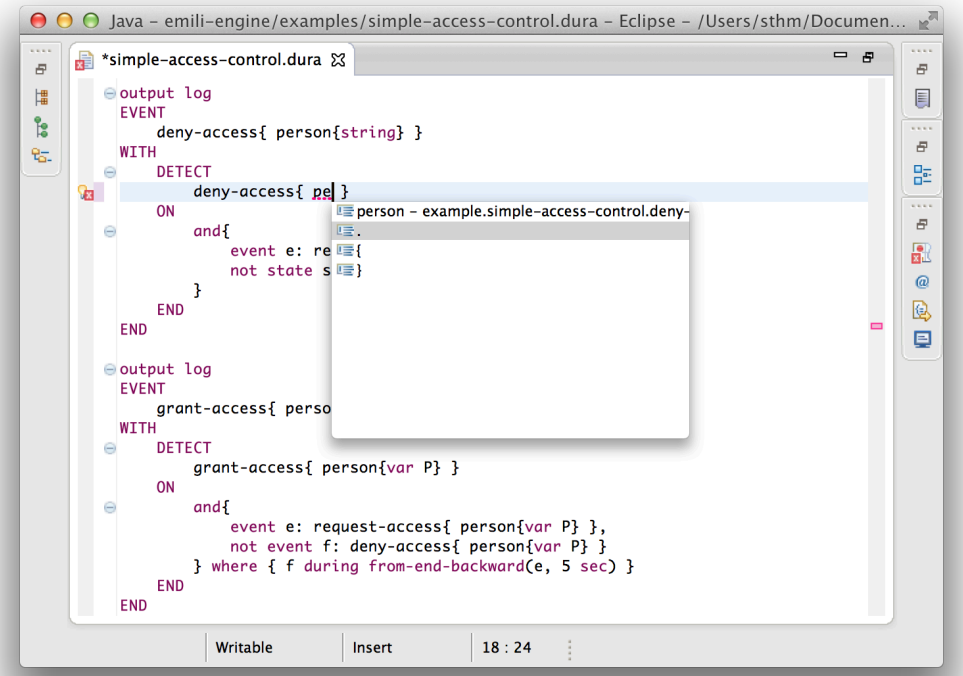


Figure 12.4: Content Assist

The editor is well integrated with the Eclipse IDE and provides syntax coloring and several advanced features such as the syntactical validation of programs; quick fixes that suggest how to resolve errors; content assist for stream names, stream attributes, and variables; an outline of the program for quick navigation in modules; and the capability to fold and hide certain program parts. Moreover, the integration in the Eclipse IDE provides various searching and navigation capabilities in addition to sophisticated refactoring capabilities, such as, the global renaming of stream names and attributes.

12.5 CURRENT LIMITATIONS

The implementation prototype does not fully cover all aspects that have been elaborated for Dura. However, it is complete in the sense that all following issues can be avoided by adapting the rules by hand.

KEYWORDS AS LABELS Keywords of Dura, such as, `begin` and `end`, cannot be directly used as attribute names. Instead, keywords used as attribute names need to be preceded with a hat sign (^).

For instance, to determine the end of the reception time

```
reception-time{ ^end{ end(e)+1min } }
```

must be used instead of

```
reception-time{ end{ end(e)+1min } }
```

DISJUNCTIVE QUERIES The current prototype misses support of disjunctive event queries. Although inconvenient for programmers, queries with disjunctions can be expanded by hand using the distributive law as it is known from mathematical logic.

STATEFUL OBJECTS Currently only static stateful objects are supported by the implementation prototype. Although (dynamic) stateful objects can be expressed by means of event queries as it is elaborated in Section 11.2, it seems desirable to directly integrate the translation into the compiler.

WEAKLY RANGE RESTRICTION The parser implements a notion of weakly range restriction that slightly deviates from the one elaborated in Section 5.2.1. In particular, the first supplement of a conjunctive query with negation must be a `where` part, although sometimes it is desirable to specify a `let` part followed by a `where` part. The expressions bound to variables in a `let` part can be directly specified in the `where` part, though.

ELIMINATION OF ANONYMOUS COMPLEX ACTIONS The current implementation prototype merely supports the compilation of reac-

tive rules with a single action in their head. Accordingly, anonymous complex actions specified in the head of a reactive rule need to be manually eliminated by introducing an appropriate complex action rule and by substituting the head with an invocation of the added complex action.

Part V

CONCLUSION AND OUTLOOK

13 FUTURE WORK AND PERSPECTIVES

Emergency Management (EM) and especially the EM related use cases of the EMILI project have substantially influenced the design of Dura. However, not all reasonable aspects could have been thoroughly considered for the initial design of Dura. Thus, in particular aspects that are not directly related to the considered EM use cases were given less consideration.

Accordingly, various extensions, ranging from basic syntactic sugar to more generic aspects that are relevant for Event Query Languages (EQLs) in general, can be elaborated and studied more intense in the future.

13.1 EXTENSIONS FOR EVENT QUERIES

Event queries can be extended in several manners including syntactic sugar and a generalization of weak range restriction.

OPTIONAL QUERIES FOR AGGREGATION The following query is intended to count the number of uncertain alarms in case a certain alarm occurs in an area.

```
and{
  event e: certain-alarm{ area{var A} }
  event f: uncertain-alarm{ area{var A} }
} where { f during from-end-backward(e, 1min) }
  group by { e } aggregate { var Count = count(f) }
```

Thereby, the conjunctive query matches only if at least one uncertain alarm occurs in the respective time interval. Accordingly, the variable `Count` is always greater than but never equal to zero. As a consequence, the case when no uncertain alarm occurs needs to be covered with an additional query. In practice, this often leads to multiple more or less redundant rules.[1]

It seems desirable to introduce an `optional` operator that is specified before the query for `uncertain-alarms` and causes the conjunctive query to match in presence and absence of uncertain alarms. In this way, the sub-query for uncertain alarms can be adapted to

```
optional event f: uncertain-alarm{ area{var A} }
```

and thus the entire query matches in presence and absence of uncertain alarms and thus facilitates a `Count` equal to zero. Note that the

1 Note that this behavior is common for query languages, including, SQL and SPARQL.

conjunctive query thus resembles an outer join rather than a natural join. A similar construct is proposed for Xcerpt [FBS07; BS02] where it is mainly intended to cope with the heterogeneous nature of semi-structured data.

GENERALIZATION OF WEAK RANGE RESTRICTION The notion of weak range restriction has been elaborated to provide a more liberal variation of range restriction that is convenient for programmers. Thereby, the conditions for a query or rule to be weakly range restriction have been determined in such a way that a weakly range restricted query or rule can be converted into a range restricted one. However, the elaborated conditions are sufficient but not necessary. They can thus be further refined so that more queries and rules would be accepted.

Listing 13.1: A Query Which is Not Weakly Range Restricted

```
and{
  event e:

  or{
    not { event f: ... } where { f during from-end(e, 1min) }
    ...
  }
}
```

For instance, the query in Listing 13.1 is not weakly range restricted because of the (mandatory) temporal condition in the `where` part. This still holds if the condition is moved to the supplement of the disjunction and conjunction, respectively. However, by expanding the disjunction one obtains a semantically equivalent query that is not only weakly range restricted but also range restricted.

Listing 13.2: Another Query Which is Not Weakly Range Restricted

```
and{
  event e: ...

  not and{
    event f: ...

    not { event g: ... } where { g during from-end(e, 1min) }
  }
}
```

Similarly, the query in Listing 13.2 is not weakly range restricted. However, it can be made weakly range restricted without affecting its semantics in the following way. The query `event e` is duplicated and added as `event e'` to the inner conjunctive query. Moreover, inside the

inner conjunctive query references to `e` are replaced by references to `e'` and the condition `id(e) = id(e')` is added.

Both illustrated transformations can be naturally integrated into the normalization of Dura_C queries described in Section 10.3 and the notion of weak range restriction can be adapted accordingly.

IDENTIFIERS AS SYNTACTIC SUGAR FOR NAMES When stateful objects are modified by triggering a `terminate` or `update` action, the key of the currently valid value needs to be passed to the respective action as parameter. Thus, to formulate the update

```
action a: «name»$update{ query{id(s)}, set{ «new values» } }
```

the identifier `s` needs to be bound by means of a query to the stateful object `«name»`. However, as the identifier `s` determines the name of the stateful object, the invocation of `terminate` and `update` actions can be specified in a syntactically more concise manner that uses the identifier `s` to refer to the action `«name»`.

```
action a: s$terminate{}
action a: s$update{ «new values» }
```

This applies similarly to the supplements of complex actions that contain queries for events entailed by actions. Thereby, the queried events often need to be related to an action instance that has been caused by the particular complex action. It therefore seems reasonable to introduce

```
succeeds on { a$succeeded{} }
```

as syntactic shortcut for

```
succeeds on { «name»$succeeded{ payload{ id{id(a)} } } }
```

whereby `«name»` corresponds to the name of the action that is referenced by the action identifier `a`.

13.2 EXTENSIONS FOR STATEFUL OBJECTS

Stateful objects as they are introduced for Dura have been substantially inspired by the representation of operation modes. Accordingly, some aspects of stateful objects that are not directly relevant for this purpose have not been thoroughly considered in the first place.

VIEWS ON STATEFUL OBJECTS It seems natural to introduce views on stateful objects that resemble deductive rules but entail stateful objects instead of events. Views on stateful objects are for instance convenient determine the operation mode of a station by accumulating the highest operation mode of any of its subareas.

Listing 13.3: A View on a Stateful Object

```
DERIVE
  aggregated-operation-mode{ area{var S}, mode{var Mmax} }
ON
  and{
    state s: station-area{ area{var S}, area{var A} }
    state t: operation-mode{ area{var A}, mode{var M} }
  } group by { var S } aggregate { var Mmax = max(var M) }
END
```

Views on stateful objects can be realized as it is illustrated in Listing 13.4. Thereby, reactive rules are used to update values of the derived stateful object whenever the value of one of the queried stateful objects changes. However, this only indicates the principle feasibility of views on stateful objects. Beyond that it seems desirable to elaborate a more direct translation that only triggers updates for values that actually changed.

Listing 13.4: Possible Realization of the View from Listing 13.3

```
ON
  and{
    or{
      event s$mod: operation-mode$created{}
      event s$mod: operation-mode$terminated{}
    }

    state s: station-area{ area{var S}, area{var A} }
    state t: operation-mode{ area{var A}, mode{var M} }
    state u: aggregated-operation-mode{ area{var S} }
  } where {
    t valid-at s$mod.semantic-time,
    u valid-at s$mod.semantic-time
  } group by { end(s$mod), var S } aggregate { var Mmax = max(var M) }
DO
  action a: aggregated-operation-mode$update{
    query{id(u)}, set{ area{var S}, mode{ var Mmax } }
  }
END
```

GENERIC TEMPORAL CONDITIONS FOR STATEFUL OBJECTS The translation of actions modifying stateful objects as it is described in Section 11.2 is restricted to the translation of the two temporal relations `valid-at` and `valid-during`.

The limitation to this particular two relations enables the translation of queries for stateful objects by means of a negated query for `terminated` events:

```
and{
  ...
  event s$cre: «name»$created{ ... }
  not event s$term: «name»$terminated{}
} where { ... }
```

In this way, a query for the stateful object «name» with the temporal conditions `s valid-at «tp»` matches when «tp» is exceeded. By contrast, if the negation is omitted, the query matches after the respective value is terminated, which is independent from the time «tp».[2]

However, generic temporal formulas may specify upper bounds on the termination of the stateful object, eg, `term(s) + 1min <= «tp»`. In this case, the query for the `terminated` event must not be negated as it is required to determine the value of `term(s)`. Accordingly, to support generic temporal dependencies, the translation of a query for a stateful object `s` needs to take the temporal dependencies of `s` into account. Based on whether or not they specify an upper or lower bound on `term(s)` the query for the respective `terminated` event needs to be negated. Moreover, in case disjunctive temporal conditions are specified the translation of the query for the stateful object needs to result in a disjunctive event query.

MORE VERSATILE CONFLICT RESOLUTION The currently available conflict resolution of Dura is intended to select a single value in case several updates are concurrently invoked. Although this is convenient for the adaptation of operation modes, in other situations it may be more suitable to accumulate the effect of concurrently issued updates.

A natural extension is, for instance, provided by an additional `group by` and `aggregate` construct that facilitates the aggregation of several concurrently issued updates.

```
operation-mode{ area{long}, mode{int} }
on conflict group by { area } aggregate { mode = avg(mode) }
```

In this way, it is for instance feasible to determine the (rounded) average operation mode in case of conflicting updates. Although this behavior is not desired for the considered EM use cases, it may be required in other domains.

To support this way of conflict resolution, the translation of stateful objects to reactive rules needs to be adapted. In particular, the derivation of `updated` events needs to be extended by means of a grouping over `update$request` events with coinciding semantic times.

GARBAGE COLLECTION FOR STATEFUL OBJECTS As a matter or principle, the maximum duration for the validity of values representing a stateful object is unbounded. As a consequence, the difference

2 Note that this may defer the evaluation of queries for a period of arbitrary length as the validity of a value may never end, eg, when the stateful object is never updated.

between the reception time of related `created` and `terminated` events is unbounded as well, which effectively prevents garbage collection on these event types.

However, if `created` events are repeated in regular intervals, eg, of duration `max-dist`, whereby only their reception time changes, the time of relevant `created` events can be bound below. Accordingly, as illustrated in Figure 13.1, it is sufficient to query `created` events that occur in the gray area, which designates the time interval going backward from `update` for a duration of `max-dist`. As all valid `created` events are repeatedly copied they necessarily occur in this interval.

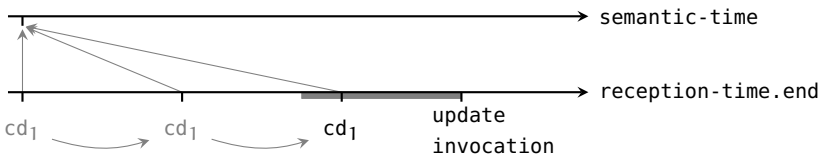


Figure 13.1: Enabling Garbage Collection by Copying Events

In this way, the reception time of `created` and `terminated` events can be bound below and thus garbage collection for these event types becomes feasible. This comes at the price of copying events, which causes additional overhead, though.

13.3 Extensions for Complex Actions

Extensions of complex actions are mainly concerned with the structure of temporal conditions which is currently limited to conjunctive formulas.

Disjunctive Domain Knowledge and Assertions Temporal formulas related to actions are restricted to conjunctive formulas. Disjunctive formulas for the specification of domain knowledge and assertions seem desirable as they facilitate, for instance, the specification of a maximal duration of an action.

```
open-fire-dampers{ area{long} }
where or{ succ() <= init()+1min, fail() <= init()+1min }
```

To support disjunctive formulas the semantic analysis algorithm Algorithm 1 from Section 8.3.1 needs to be adapted appropriately. To be applicable to a complex action $C = (W, D, H)$ with disjunctive formulas W, D, or H, the algorithm needs to be adapted so that instead of $W_A \wedge D$ each disjunct of the disjunctive normal form of $W_A \wedge D$ is independently considered by the analysis algorithm.

Thereby, the analysis is successful if it is successful on *every* disjunct. Note that it is not sufficient if the verification is successful for only one disjunct, as the verified properties should hold in any situation that emerges at runtime and not just in some situations.

DISJUNCTIVE TEMPORAL DEPENDENCIES Disjunctive formulas are excluded from temporal dependencies of complex actions to maintain a deterministic execution of actions. Non-deterministic choices, as they are common for the specification of business processes, are thus not considered for Dura. There are, however, examples that maintain a deterministic execution of actions but require disjunctive constraints.

```
or{ succ(a) <= init(b), fail(a) <= init(b) }
```

The given temporal dependencies are, for instance, intended to initiate `b` after `a` has been executed, regardless of whether `a` succeeded or failed.

To support such kind of constraints, the translation of complex actions needs to be adapted so that queries for `a$succ` and `a$fail` are translated by means of a disjunctive query, instead of a conjunctive query. Moreover, the temporal analysis for actions needs to be adapted to be suitable for disjunctive W_A as it has been described previously.

SYNTACTIC SUGAR FOR SIMPLE CONDITIONS The `IF` part of conditional actions and the specification of the status of actions expect a generic event query. However, in some situations it may be sufficient to specify simple conditions on variables or the times of actions instead of event queries. Therefore, it seems desirable to introduce

```
IF var T >= 50 THEN ...
```

as syntactic sugar for

```
IF and{} where { var T >= 50 } THEN ...
```

and likewise to introduce

```
succeeds on { succ(a) <= init(b) + 1min }
```

as syntactic sugar for

```
succeeds on and{} where { succ(a) <= init(b) + 1min }
```

13.4 EXPLOITING TEMPORAL ANALYSIS ON EVENTS

For the evaluation of Temporal Stream Algebra (TSA), a temporal analysis is applied to the temporal conditions of TSA expressions [BB12a]. It verifies the consistency of temporal dependencies and furthermore deduces some interesting properties on events. It determines, for instance, the temporal relevance of events, that is, a duration that indicates how long events of a certain type need to be stored by the Event Processing System (EPS) until they can be removed by garbage collection. It furthermore deduces the relative temporal distance between

two events, that is, the largest time difference between two events matching a certain query.

It seems desirable to elaborate a better integration of Dura and the temporal analysis that is applied to TSA expressions. In this way, the deduced information can be used for the compilation of Dura rules and thus facilitates some valuable extensions to Dura.

FURTHER IMPLICIT TIMES OF EVENTS Some of the information that is deduced by the temporal analysis can be added to the payload of events. For instance, the earliest time for the derivation of an event can be represented in a distinguished `stimulus-time` attribute. Moreover, the time of the actual derivation of the event can be represented in a distinguished `detection-time` attribute.

Both times provide insights to the current performance of the system, as their difference corresponds to the latency of events. Thus, by adding this information to the payload of events, it becomes accessible for the EPS itself as well as for external components that could recognize the current latency of the EPS and adapt to it appropriately.

IMPLICIT SUCCESS AND FAILURE OF ACTIONS Information on the relative distance between events can be exploited to implicitly determine the failure of an action based on the absence of its success. In this way, the supplement of complex actions can be restricted to contain either a `succeeds on` or a `fails on` part, effectively preventing the unintended specification of two queries that match in the same situation and thus entail a `succeeded` and `failed` event for the same action instance.

Listing 13.5: Query Determining the Success of an Action

```
DETECT
  adapt-ventilation$succeeded{ pl{ v$init.pl } }
ON
  and{
    event e: smoke{ ... }
    event v$init: adapt-ventilation$initiated{ ... }
  } where {
    end(v$init) < end(e), end(e) < end(v$init)+3min
  } group by { v$init }
END
```

For instance, the temporal analysis of TSA determines that the rule in Listing 13.5 derives an `succeeded` event at the latest after three minutes from the occurrence of the `initiated` event. This information can be used to generate a rule which derives a `failed` event if there is a `initiated` but no corresponding `succeeded` event within three minutes.

Note however, that the relative temporal distance between events is unbound unless appropriate temporal conditions are specified. Ac-

cordingly, as a matter of principle, the implicit definition for `succeeded` and `failed` events can only be applied if the relevant temporal distances remain finite.

OPTIONAL ELSE PART FOR CONDITIONAL ACTIONS The information obtained from the temporal analysis facilitates an optional `ELSE` part for conditional actions. Similar as before, knowledge on the temporal distance between events can be used to construct an event query that matches if the query in the `IF` part does not match which can be used as a trigger for the action in the `ELSE` part.

```
IF «query»
THEN action a: ...
ELSE action b: ...
```

However, an `ELSE` part can only be realized if appropriate temporal conditions on the events queried by `«query»` have been specified.

13.5 INTRODUCING A LIGHTWEIGHT TYPE SYSTEM

It seems natural to introduce a lightweight type system to Dura that resembles, for instance, the basic type declarations known from the functional language SML [MTH90]. Introducing user defined types to Dura also facilitates the support of enumerations that are capable of representing finite collections of predefined syntactical items.

```
TYPE area = long
TYPE location = { station{area}, area{area}, subarea{area} }

ENUM opm-mode = NORMAL | EXCEPTIONAL | ...
```

Even with such a rather rudimentary type system, generic long or integer values that are misused for the representation of areas and the operation mode can be avoided and instead proper types can be introduced whose correct application is statically verified.

Adding a type system to Dura is rather a matter of extending already available static type checks than of original research, as the underlying theory is well established [Pie02] and does not affect any runtime aspects of Dura. Nevertheless, Dura would certainly benefit from a lightweight type system.

13.6 GENERIC AND REASONABLE EVENT SELECTION

The conflict resolution of stateful objects specified in the `on conflict select` part can be generalized to be applicable to generic event queries. It is for example reasonable to define a `select` construct that specifies the selection criterion by referring to the attributes of the respective events.

```
and{
  event e: ...,
  event f: ...
} where { f during from-end(e, 1min) }
  select { max end(f), min id(f) }
```

Note that such a `select` statement generalizes `last` and `first` statements known from other languages whereas it avoids implicit assumptions, eg, on events occurring in the same instant as it is discussed in Section 4.5. In addition, the selection criterion can be customized to the specific requirements of programmers.

Internally, the `select` statement can be realized by translating it to appropriate event queries. Therefore, the translation can be actually derived from the translation of the `on conflict select` statement of stateful objects.

13.7 DECLARATIVE SEMANTICS FOR FULL DURA

Dura is a declarative language and has been carefully designed to facilitate a natural model theoretic semantics. To this end, the basic ideas that are used to elaborate a declarative semantics for the language XChange^{EQ} can be borrowed from [Ecko8] and applied to Dura.

First efforts towards a declarative semantics for Dura were made by Mayer [May13] in his Diploma thesis supervised by François Bry and the author of this dissertation. In, [May13] a model theoretic fixpoint based semantics is elaborated that is well suited to cover aspects related to the modification of stateful objects as it directly incorporates the law of inertia (frame problem) into the semantics by determining a minimal fixpoint. However, the presented work is focused on Dura$_C$ without recursive rules and thus requires substantial extensions to apply for Dura.

13.8 COMPLEXITY CLASSES FOR EVENT PROCESSING

Although the efficient evaluation of event queries is a fundamental aspect of EPSs there have been little theoretical considerations on the complexity of event queries.

Chandramouli, Goldstein, Barga, Riedewald, and Santos [Cha+11] elaborate a measure for the prediction of the latency of an EPS. It seems desirable to generalize their work to be applicable to evaluation scheduling strategies other than the one proposed in [Cha+11].

It seems furthermore desirable to elaborate an abstraction of EQLs similar to Datalog [CGT89], which can be seen as an abstraction of database query languages, and to elaborate complexity classes for event queries that extend existing complexity classes with a temporal dimension.

14 SUMMARY AND CONCLUSION

Inspired by three Emergency Management (EM) use cases we have elaborated the original Event Query Language (EQL) Dura to close the gap between the capabilities of current EQLs and the desire for expressive event queries that are integrated with a notion of state and the execution of versatile reactions by means of physical actuators.

To this end, Dura has been conceived as a uniform and declarative language with a clear separation of dimensions, a minimal set of well aligned language constructs, and a deep integration of events, stateful objects, and actions. In doing so, special attention has been devoted to aspects related to interactions with physical actuators which manifests in complex actions with a versatile notion of success and failure and flexible temporal dependencies in addition to the integrated access to events and non-volatile data. Moreover, these aspects are complemented by expressive event queries with support for flexible temporal conditions, grouping and negation capabilities, and support for multiple time lines.

To account for inconsistent specifications of complex actions that cause undesired behavior at runtime we have elaborated a static analysis that provably verifies crucial properties of complex actions prior to their execution. The static analysis is based on a semantics of complex actions which is carefully adapted to the inherently incomplete knowledge on physical actions.

We have complemented our theoretical findings by an operational semantics that facilitates the evaluation of Dura programs. To this end, we have identified the rather minimal sub-language Dura_C and elaborate translations that express high-level concepts, such as, stateful objects and complex actions, by the means available in Dura_C. In this way, original aspects of Dura can be easily transferred to other EQLs that are as expressive as Dura_C.

Finally, this work is rounded off by a prototypical implementation of Dura based on the Event-Mill runtime system and the elaboration of a sophisticated use case that is based on realistic EM applications that were specified as part of the joint European EMILI project.

Although the inspiration for this work has been drawn from EM related use cases, the scope of this thesis is not limited to EM and generalizes to a broad range of domains. In particular with the advent of the Internet of Things and Cyber-Physical Systems we expect various challenging applications that benefit from a deep integration of complex event detection with the capabilities of executing sophisticated composite reactions as provided by Dura.

Part VI

APPENDIX

A

FORMAL GRAMMAR

In the following, we will elaborate a formal grammar for Dura by means of the formalism that is provided by the Xtext [Xte] parser which roughly corresponds to the Extended Backus-Naur Form (EBNF) formalism [ISO96].

A.1 XTEXT GRAMMAR FORMALISM

Xtext grammars are build from production rules [Xte14]: Xtext rules are written `name : expression ;` whereby name starts with a capital letter and expressions are build from the following constructs.

1. Tokens are build from a sequence of characters escaped by single or double quotes (`'0'`, `'int'`) and character ranges using the `..` operator (`'0'..'9'`).
2. Alternatives between expressions specified by means of the `|` operator (`'and' | 'or' | 'not'`).
3. Groups determined by a sequence of expressions in addition to optional round brackets (`'a'..'z' ('0'..'9'|'a'..'z')`)
4. Unordered groups specified by means of the `&` operator (`'input' & 'output'`).
5. Rule calls referring to other rules.

Moreover, the cardinality of expressions can be adapted by means of the `?` operator (one or none), the `+` operator (one or more), and the `*` operator (zero or more).

A.2 SIMPLIFIED DURA GRAMMAR

It follows a simplified version of the Xtext grammar for Dura. Thereby, aspects related to modules and schemas as they are discussed in Section 12.1 are omitted for the sake of readability.

Moreover, technical aspect that are related to the construction of the parse tree, such as assignment operators and cross references, are omitted from the given representation.

```
Module:
  (ConstDefinition | DeclarativeRule | ReactiveRule | ActionRule)*
;


ConstDefinition:
  'CONST' ID '=' Expression
;

DeclarativeRule:
  'DETECT'
    ID '{' (Reference | (ConstructTerm Sep?)+)? '}'
  'ON'
    Query
  'END'
;

ReactiveRule:
  'ON'
    Query
  'DO'
    Action
  'END'
;

ActionRule:
  'FOR'
    ActionQuery
  'DO'
    Action
  'END'
;


Query:
  AtomicQuery
  | CompositeQuery
;

CompositeQuery:
  'and' '{' (Query Sep?)* '}' QuerySupplement*
  | 'or' '{' (Query Sep?)* '}' QuerySupplement*
  | 'not' AtomicQuery
  | 'not' '{' Query '}' QuerySupplement*
;

AtomicQuery:
  EventQuery
  | StateQuery
;

EventQuery:
  'event' ID ':' (ID|QualifiedName)
    '{' (AtomicQueryTerm | (CompositeQueryTerm Sep?)+)? '}'
    QuerySupplement*
;
```

```
StateQuery:
  'state' ID ':' (ID|QualifiedName)
    '{' (AtomicQueryTerm | (CompositeQueryTerm Sep?)+)? '}'
    QuerySupplement*
;

ActionQuery:
  'action' ID ':' ID
    '{' (AtomicQueryTerm | (CompositeQueryTerm Sep?)+)? '}'
    QuerySupplement?
;

QuerySupplement:
  'where' CompositeConditionFormula
  | 'let' '{' VariableAssignment? (Sep VariableAssignment)* '}'
  | 'group by' '{' Reference? (Sep Reference)* '}'
    ('aggregate' '{' VariableAssignment? (Sep VariableAssignment)* '}')?
;


Action:
  AtomicAction
  | CompositeAction
  | ConditionalAction
;

CompositeAction:
  'compound' '{'
    (Action Sep?)*
  '}' ActionSupplement
;

ConditionalAction:
  'IF' Query 'THEN' Action
;

AtomicAction:
  'action' ID ':' (ID|QualifiedName)
    '{' (Reference | (ConstructTerm Sep?)+)? '}'
;

ActionSupplement:
  ('where' CompositeConditionFormula)?
  ('hence' CompositeConditionFormula)?
  ('initiates on' Query)?
  ('succeeds on' Query)?
  ('fails on' Query)?
;


ConditionFormula:
  CompositeConditionFormula
  | AtomicConditionFormula
;

CompositeConditionFormula:
  ('and' | 'or' | 'not' ) '{' ConditionFormula? (Sep ConditionFormula)* '}'
```

```
;

AtomicConditionFormula:
  AddExpression (OpCompare | RelQualitativeTemporal) AddExpression
  | '{' AddExpression (Sep AddExpression)* '}' RelQuantitativeTemporal AddExpression
  | BooleanLiteral
;


Expression:
    AddExpression (OpCompare AddExpression)*
;

AddExpression:
    MultiExpression (OpAdd MultiExpression)*
;

MultiExpression:
    UnaryExpression (OpMulti UnaryExpression)*
;

UnaryExpression:
  OpUnary DuraExpression
  | DuraExpression
;

DuraExpression:
  BasicTypes '(' Expression ')'
  | '[' Expression Sep Expression ']'
  | OpAggregateValues '(' Expression ')'
  | OpRelativeTimer '(' Expression Sep Expression ')'
  | OpTimepoint '(' Expression ')'
  | OpIdentifier '(' Reference ')'
  | OpSelect '(' Expression? (Sep Expression)* ')'
  | OpRound '(' Expression Sep Expression ')'
  | AtomicExpression
;

AtomicExpression:
  Literal
  | Reference
  | Constant
  | Clock
  | '(' Expression ')'
;

Clock:
  OpClock
;


AtomicQueryTerm:
  Literal
  | Constant
  | Reference
;
```

```
CompositeQueryTerm:
  ID '{' AtomicQueryTerm '}'
  | ID '{' (CompositeQueryTerm Sep?)* '}'
;

ConstructTerm:
  ID '{' (Expression | (ConstructTerm Sep?)+)? '}'
;


Reference:
  Variable
  | QueryPath
  | StreamReference
;

Variable:
  'var' ID
;

VariableAssignment:
  'var' ID '=' Expression
;

Constant:
  'const' (ID|QualifiedName)
;

StreamReference:
  ID
  | 'event' ID
  | 'state' ID
  | 'action' ID
;


QueryPath:
  ID ProperSubPath
;

SubPath:
  ID ProperSubPath?
  | ProperSubPath
;

ProperSubPath:
  '.' ID SubPath?
;


Literal:
  LONG
  | STRING
  | FloatingPointLiteral
  | DurationLiteral
  | BooleanLiteral
;
```

```
BooleanLiteral:
  ('true' | 'false')
;

DurationLiteral:
  (LONG 'min') (LONG 'sec')? (LONG 'ms')?
  | (LONG 'sec') (LONG 'ms')?
  | (LONG 'ms')
;

FloatingPointLiteral:
  LONG ('E' ('+'|'-') LONG)
  | LONG '.' LONG ('E' ('+'|'-') LONG)?
;


Sep:
   ','
;

QualifiedName:
  ID ('.' ID)*
;

BasicTypes:
  'int' | 'long' | 'float' | 'double' | 'boolean' | 'string' | 'duration'
  | 'timestamp' | 'identifier'
;

RelQuantitativeTemporal:
  'within' | 'apart'
;

RelQualitativeTemporal:
  'before' | 'after' | 'contains' | 'during' | 'overlaps'
  | 'overlapped-by'  | 'meets' | 'met-by'| 'starts' | 'started-by'
  | 'finishes' | 'finished-by' | 'equals' | 'valid-at' | 'valid-during'
;

OpAggregateValues:
  'min' | 'max' | 'sum' | 'avg' | 'count'
;

OpTimepoint:
  'begin' | 'end'
;

OpIdentifier:
  'id' | 'init' | 'succ' | 'requested' | 'fail' | 'created' | 'term' | 'time'
;

OpRelativeTimer:
  'extend' | 'shorten' | 'extend-begin' | 'shorten-begin' | 'shift-forward'
  | 'shift-backward' | 'from-end' | 'from-end-backward' | 'from-begin'
  | 'from-begin-backward'
;
```

```
OpCompare:
  '=' | '!=' | '>=' | '<=' | '>' | '<' ;

OpAdd:
  '+' | '-';

OpMulti:
  '*' | '**' | '/' | '%';

OpUnary:
  "!" | "-" | "+";

OpSelect:
  'least' | 'greatest';

OpRound:
  'ceil' | 'floor';

OpClock:
  'system-time.now()' | 'sequence.next()'
;


ID     : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'-'|'0'..'9'|"'")*;
LONG   : ('0'..'9')+;
STRING : '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"') )* '"'
```

B

TRANSLATIONS

This chapter provides the complete rule set obtained by the transformation of the complex action `adapt-ventilation` and the stateful object `operation-mode`. Both transformations are thoroughly discussed in Chapter 11 but only excerpts of the following rule sets are given there.

B.1 COMPLETE TRANSLATION OF A COMPLEX ACTION

Consider the following complex action rule.

```
FOR
  action v: adapt-ventilation{ area{var A} }
DO
  compound{
    action a: open-fire-dampers{ area{var A} }
    action b: activate-ventilators{ area{var A} }
    action c: warn-of-smoke{ area{var A} }
  } where { succ(a) <= init(b), init(c)+1min <= init(b) }
    succeeds on and{
        event e: smoke{ area{var A}, amount{var C} }
      } where { var C < 0.1, init(v) < end(e), end(e) < init(v)+2min }
        group by { v }
END
```

The given complex action rule is used in Section 11.1 to illustrate the translation of complex actions. It is translated to the following set of deductive rules.

```
DETECT
  open-fire-dampers$requested{
    reception-time{ end{greatest(end(v$init))} }
    pl{ id{sequence.next()}, ref{v$init.pl.id}, area{var A} }
  }
ON
  event v$init: adapt-ventialtion$initiated{ pl{ area{var A} } }
END


DETECT
  activate-ventilators$requested{
    reception-time{ end{
      greatest(end(v$init), end(a$succ), end(c$init)+1min)
    } }
    pl{ id{sequence.next()}, ref{v$init.pl.id}, area{var A} }
  }
ON
```

```
  and{
    event v$init: adapt-ventialtion$initiated{ pl{ area{var A} } }
    event a$succ: open-fire-dampers$succeeded{}
    event c$init: warn-of-smoke$intitated{}
  } where { v$init.pl.id = a$succ.pl.ref, v$init.pl.id = c$init.pl.ref}
END


DETECT
  warn-of-smoke$requested{
    reception-time{ end{greatest(end(v$init))} }
    pl{ id{sequence.next()}, ref{v$init.pl.id}, area{var A} }
  }
ON
  event v$init: adapt-ventialtion$initiated{ pl{ area{var A} } }
END


DETECT
  adapt-ventialtion$requested{ pl{ id{var Id}, ref{var Id}, area{var A} } }
ON
  event e: certain-alarm{ area{var A} }
    let{ var Id = sequence.next() }
END


DETECT
  adapt-ventialtion$initiated{ pl{e.pl} }
ON
  event e: adapt-ventialtion$requested{}
END


DETECT
  adapt-ventialtion$succeeded{ pl{v$init.pl} }
ON
  and{
    event e: smoke{ area{var A}, conc{var C} }
    event v$init: adapt-ventialtion$initiated{}
  } where { var C < 0.1, end(v$init) < end(e), end(e) < end(v$init)+2min }
    group by { v$init }
END
```

B.2 COMPLETE TRANSLATION OF A STATEFUL OBJECT

Consider the following schema of a stateful object.

```
operation-mode{ area{long}, mode{int} }
on conflict select { max mode, min id }
```

The given stateful objects is used in Section 11.2 to illustrate the translation of complex actions. It is translated to the following set of deductive rules.

```
DETECT
  operation-mode$create$initiated{
    semantic-time{r.semantic-time}
    payload{r.payload}
  }
ON
  event r: operation-mode$create$requested{}
END


DETECT
  operation-mode$create$succeeded{
    semantic-time{r.semantic-time}
    payload{r.payload}
  }
ON
  event r: operation-mode$create$requested{}
END


DETECT
  operation-mode$created{
    semantic-time{r.semantic-time},
    payload{r.payload}
  }
ON
  event r: operation-mode$create$requested{}
END


DETECT
  operation-mode$created{
    semantic-time{ud.semantic-time}
    payload{ud.new-payload}
  }
ON
  event ud: operation-mode$updated{}
END


DETECT
  operation-mode$terminate$initiated{
    semantic-time{d.semantic-time}
    payload{d.payload}
  }
ON
  or{
    event d: operation-mode$terminate$succeeded{}
    event d: operation-mode$terminate$failed{}
  }
END


DETECT
  operation-mode$terminate$succeeded{
    request-time{r.request-time}
    payload{r.payload}
```

```
  }
ON
  and{
    event r: operation-mode$terminate$requested{},

    event cd: operation-mode$created{},
    not event td: operation-mode$terminated{},

    not event tr:  operation-mode$terminate$requested{}
    not {
      event ur:  operation-mode$update$requested{}
    } where and{
      end(r) > end(cd) + min-dist,
      ur.request-time < r.request-time
    }
  } where {
    end(cd) <= end(r) - min-dist,
    end(td) <= end(r) - min-dist,
    end(r) - min-dist <= end(tr), end(tr) <= end(r),
    end(r) - min-dist <= end(ur), end(ur) <= end(r),

    cd.payload.id = td.payload.id,
    cd.payload.id = r.payload.query,

    tr.id != r.id,
    tr.payload.query = r.payload.query,
    tr.request-time < r.request-time,

    ur.payload.query = r.payload.query,
  }
END


DETECT
  operation-mode$terminate$failed{
    request-time{tr.request-time}
    payload{tr.payload}
  }
ON
  and{
    event tr: operation-mode$terminate$requested{}
    not event ts: operation-mode$terminate$succeeded{}

    event td: operation-mode$terminated{}
  } where {
    tr.payload.id = ts.payload.id,
    tr.payload.query = td.payload.id,

    tr.request-time = ts.request-time,
    td.request-time <= tr.request-time,

    tr.observation-time <= td.request-time
  }
END


DETECT
```

```
  operation-mode$terminated{
    request-time{r.request-time},
    payload{cd.payload},
  }
ON
  and{
    event r: operation-mode$terminate$succeeded{}

    event cd: operation-mode$created{}

    not event ts: operation-mode$terminate$succeeded{}
  } where {
    end(cd) + min-dist <= end(r),
    end(r) - min-dist <= end(ts), end(ts) <= end(r),

    r.payload.query = cd.payload.id,
    r.payload.query = ts.payload.query,

    r.request-time = ts.request-time,
    ts.payload.id > r.payload.id,
  }
END


DETECT
  operation-mode$terminated{
    request-time{r.request-time},
    payload{
      id{us.payload.id},
      area{us.payload.set.area}, mode{us.payload.set.mode}
    }
  }
ON
  and{
    event r: operation-mode$terminate$succeeded{}

    not event ts: operation-mode$terminate$succeeded{}

    event us: operation-mode$update$succeeded{}
    not event other-us: operation-mode$update$succeeded{}
  } where {
    end(r) - min-dist <= end(ts), end(ts) <= end(r),
    end(r) - min-dist <= end(us), end(us) <= end(r),
    end(r) - min-dist <= end(other-us), end(other-us) <= end(r),

    r.payload.query = us.payload.query,
    r.payload.query = ts.payload.query,
    r.payload.query = other-us.payload.query,

    r.request-time = ts.request-time,
    ts.payload.id > r.payload.id,

    us.request-time < r.request-time,
    us.request-time < other-us.request-time,
    other-us.request-time < r.request-time
  }
END
```

```
DETECT
  operation-mode$terminated{
    request-time{ud.request-time}
    payload{ud.old-payload}
  }
ON
  event ud: operation-mode$updated{}
END


DETECT
  operation-mode$update$initiated{
    semantic-time{d.semantic-time}
    payload{d.payload}
  }
ON
  and{
    event d: operation-mode$update$succeeded{}
    event d: operation-mode$update$failed{}
  }
END


DETECT
  operation-mode$update$succeeded{
    request-time{r.request-time}
    payload{r.payload}
  }
ON
  and{
    event r: operation-mode$update$requested{},

    event cd: operation-mode$created{},
    not event td: operation-mode$terminated{},

    not event tr:  operation-mode$terminate$requested{}
    not {
      event ur:  operation-mode$update$requested{}
    } where or{
      and{ end(r) > end(cd) + min-dist, ur.request-time < r.request-time },
      and{
        ur.request-time = r.request-time,
        or{
          ur.pl.mode > r.pl.mode,
          and{ ur.pl.mode = r.pl.mode, ur.pl.id < r.pl.id }
        }
      }
    }
  } where {
    end(cd) <= end(r) - min-dist,
    end(td) <= end(r) - min-dist,
    end(r) - min-dist <= end(tr), end(tr) <= end(r),
    end(r) - min-dist <= end(ur), end(ur) <= end(r),

    cd.payload.id = td.payload.id,
```

```
    cd.payload.id = r.payload.query,

    tr.payload.query = r.payload.query,
    tr.request-time <= r.request-time,

    ur.id != r.id,
    ur.payload.query = r.payload.query,
  }
END


DETECT
  operation-mode$update$failed{
    request-time{r.request-time}
    payload{r.payload}
  }
ON
  and{
    event r: operation-mode$update$requested{}
    not event us: operation-mode$update$succeeded{}

    event td: operation-mode$terminated{}
  } where {
    r.payload.id = us.payload.id,
    r.payload.query = td.payload.id,

    r.request-time = us.request-time,
    td.request-time <= r.request-time,

    r.observation-time <= td.request-time
  }
END


DETECT
  operation-mode$updated{
    request-time{r.request-time},
    old-payload{cd.payload},
    new-payload{
      id{r.payload.id},
      area{r.payload.set.area}, mode{r.payload.set.mode}
    }
  }
ON
  and{
    event r: operation-mode$update$succeeded{}

    event cd: operation-mode$created{}

    not event us: operation-mode$update$succeeded{}
  } where {
    cd.payload.id = r.payload.query,
    cd.payload.id = us.payload.query,

    end(cd) + min-dist <= end(r),
    end(r) - min-dist <= end(us), end(us) <= end(r),
```

```
    us.request-time < r.request-time
  }
END

DETECT
  operation-mode$updated{
    request-time{r.request-time},
    old-payload{
      id{us.payload.id},
      area{us.payload.set.area}, mode{us.payload.set.mode}
    }
    new-payload{
      id{r.payload.id},
      area{r.payload.set.area}, mode{r.payload.set.mode}
    }
  }
ON
  and{
    event r: operation-mode$update$succeeded{}
    event us: operation-mode$update$succeeded{}
    not event other-us: operation-mode$update$succeeded{}
  } where {
    r.payload.query = us.payload.query,
    r.payload.query = other-us.payload.query,

    end(r) - min-dist <= end(us), end(us) <= end(r),
    end(r) - min-dist <= end(other-us), end(other-us) <= end(r),

    us.request-time < r.request-time,
    other-us.request-time < r.request-time,
    us.request-time < other-us.request-time
  }
END
```

Consider the following rules querying and updating the stateful object from above.

```
DETECT
  answer{ payload{ area{var A}, mode{var M} } }
ON
  and{
    event e: query{ area{var A} },
    state s: operation-mode{ area{var A}, mode{var M} }
  } where { s valid-at end(e) }
END

ON
  event e: touch{ area{var A}, mode{var M} }
DO
  action a: operation-mode$create{ area{var A}, mode{var M} }
END

ON
  and{
    event e: rm{ area{var A} },
    state s: operation-mode{ area{var A} }
  } where { s valid-at end(e) }
DO
  action a: operation-mode$terminate{ id{id(s)} }
END

ON
  and{
    event e: modify{ area{var A}, set{ mode{var M} } },
    state s: operation-mode{ area{var A} }
  } where { s valid-at end(e) }
DO
  action a: operation-mode$update{
    query{ id{id(s)} },
    set{ area{var A}, mode{var M} }
  }
END
```

The given rules are translated to the following set of deductive rules. Thereby, each rule from above corresponds to one of the rules below.

```
DETECT
  answer{ area{var A}, mode{var M} }
ON
  and{
    event e: query{ area{var A} }

    event cd: operation-mode$created{ payload{ area{var A}, mode{var M} } },
    not event td: operation-mode$terminated{}
  } where {
    cd.payload.id = td.payload.id,
```

```
    cd.request-time < end(e),
    td.request-time < end(e)
  }
END


DETECT
  operation-mode$create$requested{
    reception-time{ end{var RequestTime} }
    payload{ area{var A}, mode{var M}, id{sequence.next()} }
  }
ON
  event e: touch{ area{var A}, mode{var M} }
    let { var RequestTime = end(e) }
END


DETECT
  operation-mode$terminate$requested{
    observation-time{var QueryTime}
    request-time{var RequestTime}
    reception-time{ end{greatest(end(cd)+min-dist, var RequestTime)} }
    payload{ query{cd.payload.id}, id{sequence.next()} }
  }
ON
  and{
    event e: rm{ area{var A} }

    event cd: operation-mode$created{ payload{ area{var A} } },
    not event td: operation-mode$terminated{}
  } let {
    var QueryTime = end(e),
    var SemanticTime = var QueryTime,
    var ReceptionTime = greatest(var SemanticTime, end(cd) + mind-dist)
  } where {
    cd.payload.id = td.payload.id,

    cd.semantic-time < end(e),
    td.semantic-time < end(e),

    end(cd) <= var QueryTime,
    end(td) <= var QueryTime - min-dist
  }
END


DETECT
  operation-mode$update$requested{
    observation-time{var QueryTime}
    request-time{var RequestTime}
    reception-time{ end{greatest(end(cd)+min-dist, var RequestTime)} }
    payload{
      query{cd.payload.id},
      set{ area{var A}, mode{var M} }
      id{sequence.next()}
    }
```

```
  }
ON
  and{
    event e: modify{ area{var A}, set{ mode{var M} } }

    event cd: operation-mode$created{ payload{ area{var A} } },
    not event td: operation-mode$terminated{}
  } let {
    var QueryTime = end(e),
    var SemanticTime = var QueryTime,
    var ReceptionTime = greatest(var SemanticTime, end(cd) + mind-dist)
  } where {
      cd.payload.id = td.payload.id,

      cd.semantic-time < end(e),
      td.semantic-time < end(e),

      end(cd) <= var QueryTime,
      end(td) <= var QueryTime - min-dist
  }
END
```

BIBLIOGRAPHY

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. "The CQL Continuous Query Language: Semantic Foundations and Query Execution." In: *The VLDB Journal* 15.2 (2006), pp. 121–142 (cit. on pp. 22, 23).

[ABW88] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. "Towards a Theory of Declarative Knowledge." In: *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988, pp. 89–148 (cit. on pp. 75, 145, 149, 176, 227).

[AC05] Raman Adaikkalavan and Sharma Chakravarthy. "Formalization and Detection of Events Using Interval-Based Semantics." In: *Proceedings of the International Conference on Management of Data*. COMAD'2005. Computer Society of India, 2005, pp. 58–69 (cit. on p. 21).

[AC06] Raman Adaikkalavan and Sharma Chakravarthy. "SnoopIB: Interval-Based Event Specification and Detection for Active Databases." In: *Data and Knowledge Engineering* 1.59 (2006), pp. 139–165 (cit. on p. 21).

[ACG00] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. "SAT-Based Procedures for Temporal Reasoning." In: *Proceedings of the European Conference on Planning*. ECP'99. Springer, 2000, pp. 97–108 (cit. on p. 167).

[AD90] Rajeev Alur and David L. Dill. "Automata For Modeling Real-Time Systems." In: *Proceedings of the International Colloquium on Automata, Languages and Programming*. ICALP'90. Springer, 1990, pp. 322–335 (cit. on pp. 26, 28).

[AE04] Asaf Adi and Opher Etzion. "Amit— the situation manager." In: *The VLDB Journal* 13.2 (2004), pp. 177–203 (cit. on pp. 20, 21).

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu, eds. *Foundations of Databases: The Logical Level*. 1st. Addison-Wesley, 1995. ISBN: 978-0-201-53771-0 (cit. on pp. 40, 57, 187, 189, 194).

[All83] James F. Allen. "Maintaining Knowledge about Temporal Intervals." In: *Communications of the ACM* 26.11 (1983), pp. 832–843 (cit. on p. 61).

[Ani+10] Darko Anicic et al. "A Rule-based Language for Complex Event Processing and Reasoning." In: *Proceedings of the International Conference on Web Reasoning and Rule Systems*. RR'10. Springer, 2010, pp. 42–57 (cit. on pp. 30, 40).

[Ani+12a] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. "Real-time Complex Event Recognition and Reasoning: A Logic Programming Approach." In: *Applied Artificial Intelligence* 26.1-2 (2012): *Special Issue on Event Recognition*, pp. 6–57 (cit. on p. 30).

[Ani+12b] Darko Anicic, Sebastian Rudolph, Paul Fodor, and Nenad Stojanovic. "Stream Reasoning and Complex Event Processing in ETALIS." In: *Semantic Web Journal* 3.4 (2012): *Special Issue on Semantic Web Tools and Systems*, pp. 397–407 (cit. on p. 30).

[ANS] ANSYS Inc. *ANSYS Fluent*. URL: http://www.ansys.com/Products/Simulation+Technology/Fluid+Dynamics (visited on 06/30/2014) (cit. on p. 10).

[Art+12] Alexander Artikis, Anastasios Skarlatidis, François Portet, and Georgios Paliouras. "Logic-based Event Recognition." In: *The Knowledge Engineering Review* 27.4 (2012), pp. 469–506 (cit. on p. 30).

[ASP12] Alexander Artikis, Marek Sergot, and Georgios Paliouras. "Run-time Composite Event Recognition." In: *Proceedings of the International Conference on Distributed Event-Based Systems*. DEBS'12. ACM, 2012, pp. 69–80 (cit. on pp. 30, 83).

[Bat94] Don Batory. *The LEAPS Algorithm*. Tech. rep. University of Texas at Austin, 1994 (cit. on p. 24).

[BB12a] Simon Brodt and François Bry. *Analysing Temporal Relations: Beyond Windows, Frames and Predicates*. University of Munich, 2012 (cit. on pp. 5, 76, 77, 165, 187–189, 194, 195, 205, 206, 245, 249, 253, 271).

[BB12b] Simon Brodt and François Bry. *Temporal Stream Algebra*. University of Munich, 2012 (cit. on pp. 163, 169, 170).

[BC06] Roger S. Barga and Hillary Caituiro-Monge. "Event Correlation and Pattern Detection in CEDR." In: *Proceedings of the International Conference on Current Trends in Database Technology*. EDBT'06. Springer, 2006, pp. 919–930 (cit. on p. 21).

[BD13] James C. Boerkoel Jr. and Edmund H. Durfee. "Decoupling the Multiagent Disjunctive Temporal Problem." In: *Proceedings of the International Conference on Autonomous Agents and Multi-agent Systems*. AAMAS'13. International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 1145–1146 (cit. on p. 167).

[BE07] François Bry and Michael Eckert. "Rule-based Composite Event Queries: The Language XChange^{EQ} and its Semantics." In: *Proceedings of the International Conference on Web Reasoning and Rule Systems*. RR'07. Springer, 2007, pp. 16–30 (cit. on pp. 4, 28, 30, 39–41, 46, 52, 56, 59, 113).

[BE08a] François Bry and Michael Eckert. "On Static Determination of Temporal Relevance for Incremental Evaluation of Complex Event Queries." In: *Proceedings of the International Conference on Distributed Event-based Systems*. DEBS'08. ACM, 2008, pp. 289–300 (cit. on pp. 158, 170).

[BE08b] François Bry and Michael Eckert. "Rules for Making Sense of Events: Design Issues for High-Level Event Query and Reasoning Languages." In: *AI Meets Business Rules and Process Management, Proceedings of the AAAI Press Spring Symposium*. AAAI Press. 2008 (cit. on pp. 4, 46, 102, 113).

[Beh+06] Erik Behrends, Oliver Fritzen, Wolfgang May, and Franz Schenk. "Combining ECA Rules with Process Algebras for the Semantic Web." In: *Proceedings of the International Conference on Rules and Rule Markup Languages for the Semantic Web*. RULEML'06. IEEE, 2006, pp. 29–38 (cit. on p. 35).

[BEP06a] François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. "Querying Composite Events for Reactivity on the Web." In: *Proceedings of the International Conference on Advanced Web and Network Technologies, and Applications*. APWeb'06. Springer, 2006, pp. 38–47 (cit. on p. 35).

[BEP06b] François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. "Reactivity on the Web: Paradigms and Applications of the Language XChange." In: *Journal of Web Engineering* 5.1 (2006), pp. 3–24 (cit. on pp. 21, 35, 41, 52).

[Ber+07] Bruno Berstel, Philippe Bonnard, François Bry, Michael Eckert, and Paula-Lavinia Pătrânjan. "Reactive Rules on the Web." In: *Proceedings of the International Summer School Conference on Reasoning Web*. RW'07. Springer, 2007, pp. 183–239 (cit. on pp. 24, 36).

[Bet01] Marco Bettelini. "CFD for Tunnel Safety." In: *FLUENT User Meeting*. 2001 (cit. on p. 10).

[Bet08] Marco Bettelini. "Transient One-dimensional Simulation for Optimum Road-Tunnel Ventilation." In: *Proceedings of the World Tunnel Congress*. 2008 (cit. on p. 10).

[BFC86] Avron Barr, Edward A. Feigenbaum, and Paul R. Cohen, eds. *Handbook of Artificial Intelligence*. Addison-Wesley, 1986. ISBN: 978-0-201-11810-0 (cit. on p. 73).

[BHB10] Simon Brodt, Steffen Hausmann, and François Bry. *Reactive Rules for Emergency Management*. Technical Report. EMILI Deliverable D4.2. University of Munich, 2010. 51 pp. (cit. on pp. x, 15, 18).

[BHB11] Simon Brodt, Steffen Hausmann, and François Bry. *Implementation*. Technical Report. EMILI Deliverable D4.5. University of Munich, 2011. 57 pp. (cit. on pp. ix, 39).

[BHB12] Simon Brodt, Steffen Hausmann, and François Bry. *Refinement of the Implementation of Event Processing and ECA Rules for SITE*. Technical Report. EMILI Deliverable D4.7. University of Munich, 2012. 76 pp. (cit. on pp. ix, 39, 249, 255, 256, 259).

[BN09] Philip Bernstein and Eric Newcomer. *Principles of Transaction Processing*. 2nd. Morgan Kaufmann, 2009. ISBN: 978-1-55860-623-4 (cit. on p. 35).

[Boy09] Stuart A. Boyer. *SCADA: Supervisory Control And Data Acquisition*. 4th. International Society of Automation, 2009. ISBN: 978-1-936007-09-7 (cit. on p. 96).

[Bro+10] Simon Brodt, Steffen Hausmann, François Bry, Olga Poppe, and Michael Eckert. *A Survey on IT-Techniques for a Dynamic Emergency Management in Large Infrastructures*. Technical Report. EMILI Deliverable D4.1. University of Munich, 2010. 48 pp. (cit. on pp. x, 15, 18).

[BRS13] Marco Bettelini, Samuel Rigert, and Nikolaus Seifert. "Optimum Emergency Management through Physical Simulation— Findings from the EMILI Research Project." In: *Proceedings of the World Tunnel Congress*. CRC Press, 2013, pp. 290–297 (cit. on p. 11).

[Bry+06] François Bry, Michael Eckert, Paula-Lavinia Pătrânjan, and Inna Romanenko. "Realizing Business Processes with ECA Rules: Benefits, Challenges, Limits." In: *Proceedings of the International Workshop on Principles and Practice of Semantic Web*. Springer, 2006, pp. 48–62 (cit. on pp. 35, 111).

[Bry+07] François Bry et al. "Foundations of Rule-Based Query Answering." In: *Proceedings of the International Summer School on Reasoning Web*. Vol. 4636. Lecture Notes in Computer Science. Springer, 2007 (cit. on pp. 67, 70, 164, 174).

[Bry+08] François Bry, Bernhard Lorenz, Hans Jürgen Ohlbach, Martin Roeder, and Marc Weinberger. "The Facility Control Markup Language FCML." In: *Proceedings of the International Conference on Digital Society*. ICDS'08. IEEE, 2008, pp. 117–122 (cit. on pp. 16, 96).

[BS02] François Bry and Sebastian Schaffert. "Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification." In: *Proceedings of the International Conference on Logic Programming*. ICLP'02. Springer, 2002, pp. 255–270 (cit. on pp. 29, 41, 43, 52, 53, 266).

[BSB11] Marco Bettelini, Nikolaus Seifert, and François Bry. "Innovatives Sicherheitssystem für U-Bahn-Stationen." German. In: *Fachzeitschrift für Information Management & Consulting* (1 2011), pp. 71–78 (cit. on pp. 12, 13).

[Bul03] Peter S. Bullen. "The Power Means." In: *Handbook of Means and Their Inequalities*. Vol. 560. Mathematics and Its Applications. Springer, 2003, pp. 175–265 (cit. on p. 125).

[BW03] David Bailey and Edwin Wright. *Practical SCADA for Industry*. Newnes, 2003 (cit. on pp. 15, 16).

[CA08] Sharma Chakravarthy and Raman Adaikkalavan. "Events and Streams: Harnessing and Unleashing Their Synergy!" In: *Proceedings of the International Conference on Distributed Event-based Systems*. DEBS'08. ACM, 2008, pp. 1–12 (cit. on p. 31).

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. "What You Always Wanted to Know About Datalog (And Never Dared to Ask)." In: *IEEE Transactions on Knowledge and Data Engineering* 1.1 (1989), pp. 146–166 (cit. on pp. 40, 274).

[Cha+11] Badrish Chandramouli, Jonathan Goldstein, Roger Barga, Mirek Riedewald, and Ivo Santos. "Accurate Latency Estimation in a Distributed Event Processing System." In: *Proceedings of the International Conference on Data Engineering*. ICDE'11. IEEE, 2011, pp. 255–266 (cit. on p. 274).

[Cha+94] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. "Composite Events for Active Databases: Semantics, Contexts and Detection." In: *Proceedings of the International Conference on Very Large Data Bases*. VLDB'94. Morgan Kaufmann, 1994, pp. 606–617 (cit. on pp. 20, 21).

[Cla78] Keith L. Clark. "Negation as Failure." In: *Logic and Data Bases*. Springer, 1978, pp. 293–322 (cit. on p. 72).

[CM03] William F. Clocksin and Chritopher S. Mellish. *Programming in Prolog*. 5th. Springer, 2003. ISBN: 3-540-00678-8 (cit. on pp. 30, 40).

[CM10] Gianpaolo Cugola and Alessandro Margara. "TESLA: A Formally Defined Event Specification Language." In: *Proceedings of the International Conference on Distributed Event-Based Systems*. DEBS'10. ACM, 2010, pp. 50–61 (cit. on pp. 30, 40, 45).

[CM12a] Gianpaolo Cugola and Alessandro Margara. "Complex Event Processing with T-REX." In: *Journal of Systems and Software* 85.8 (2012), pp. 1709–1728 (cit. on p. 30).

[CM12b] Gianpaolo Cugola and Alessandro Margara. "Processing Flows of Information: From Data Stream to Complex Event Processing." In: *ACM Computing Surveys* 44.3 (2012), 15:1–15:62 (cit. on pp. 19, 32, 41).

[Cor+01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. 2nd. MIT Press, 2001. ISBN: 978-0-07-013151-4 (cit. on pp. 77, 158, 167).

[Dem+07] Alan J. Demers et al. "Cayuga: A General Purpose Event Monitoring System." In: *Proceedings of the Biennial Conference on Innovative Data Systems Research*. CIDR'07. 2007, pp. 412–422 (cit. on p. 24).

[DHL91] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. "A Transactional Model for Long-Running Activities." In: *Proceedings of the International Conference on Very Large Data Bases*. VLDB'91. Morgan Kaufmann, 1991, pp. 113–122 (cit. on p. 35).

[DK75] Randall Davis and Jonathan King. *An Overview of Production Systems*. Stanford University, 1975, p. 43 (cit. on p. 73).

[DMP91] Rina Dechter, Itay Meiri, and Judea Pearl. "Temporal Constraint Networks." In: *Artificial Intelligence* 49.1-3 (1991), pp. 61–95 (cit. on pp. 158, 166, 169, 170, 174).

[Droa] JBoss Community. *Drools Expert*. URL: `http://www.jboss.org/drools/drools-expert.html` (visited on 06/30/2014) (cit. on p. 26).

[Drob] JBoss Community. *Drools Fusion*. URL: `http://www.jboss.org/drools/drools-fusion.html` (visited on 06/30/2014) (cit. on pp. 25, 26, 33, 83).

[DS99] Axel Daneels and Wayne Salter. "What is SCADA?" In: *International Conference on Accelerator and Large Experimental Physics Control Systems*. ICALEPCS'99. 1999, pp. 339–343 (cit. on p. 96).

[Eck08] Michael Eckert. "Complex Event Processing with XChange^{EQ}: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events." Doctoral Thesis. Institute for Informatics, University of Munich, 2008 (cit. on pp. 4, 28, 30, 34, 39, 46, 56, 59, 113, 114, 158, 170, 274).

[Eck+11a] Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. "A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed." In: *Reasoning in Event-based Distributed Systems*. Vol. 347. Studies in Computational Intelligence. Springer, 2011, pp. 47–70 (cit. on pp. ix, 18).

[Eck+11b] Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann. "Two Semantics for CEP, no Double Talk: Complex Event Relational Algebra and its Application to XChange^{EQ}." In: *Reasoning in Event-based Distributed Systems*. Vol. 347. Studies in Computational Intelligence. Springer, 2011, pp. 71–98 (cit. on pp. ix, 15, 59).

[Ecla] Eclipse Foundation, Inc. *Eclipse Documentaion: Installing New Software*. URL: http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Ftasks%2Ftasks-129.htm (visited on 06/30/2014) (cit. on p. 259).

[Eclb] Eclipse Foundation, Inc. *Eclipse IDE*. URL: http://www.eclipse.org (visited on 06/30/2014) (cit. on pp. 249, 259).

[EE11] Yagil Engel and Opher Etzion. "Towards Proactive Event-driven Computing." In: *Proceedings of the International Conference on Distributed Event-based System*. DEBS'11. ACM, 2011, pp. 125–136 (cit. on p. 32).

[EEF12] Yagil Engel, Opher Etzion, and Zohar Feldman. "A Basic Model for Proactive Event-driven Computing." In: *Proceedings of the International Conference on Distributed Event-Based Systems*. DEBS'12. ACM, 2012, pp. 107–118 (cit. on p. 32).

[EMF] Eclipse Foundation, Inc. *Eclipse Modeling Framework Project*. URL: http://www.eclipse.org/modeling/emf/ (visited on 06/30/2014) (cit. on p. 253).

[EMI] EMILI Consortium. *EMILI— Emergency Management in Large Infrastructure*. URL: http://www.emili-project.eu (visited on 06/30/2014) (cit. on p. 9).

[EN10] Opher Etzion and Peter Niblett. *Event Processing in Action*. 1st. Manning Publications Co., 2010. ISBN: 978-1-935182-21-4 (cit. on p. 32).

[EPL] Eclipse Foundation, Inc. *Eclipse Public License - v 1.0*. URL: https://www.eclipse.org/legal/epl-v10.html (visited on 06/30/2014) (cit. on p. 255).

[Esp] EsperTech Inc. *Esper*. URL: http://esper.codehaus.org (visited on 06/30/2014) (cit. on pp. 24, 31, 33, 35).

[Esp10] Jose Luis Marín Español. *Specific Report for Use Case III, Power Networks*. EMILI Deliverable D3.1 Annex C. Grupo AIA, 2010. 18 pp. (cit. on p. 14).

[Esp14] EsperTech Inc. *EsperIO Reference*. 2014 (cit. on p. 33).

[FBS07] Tim Furche, François Bry, and Sebastian Schaffert. *Xcerpt 2.0 Specification of the (Core) Language Syntax*. Reverse Deliverable I4-D12. 2007, pp. 1–216 (cit. on pp. 29, 53, 54, 266).

[FM77] C. Forgy and J. McDermott. "OPS: A Domain-independent Production System Language." In: *Proceedings of the International Joint Conference on Artificial Intelligence*. IJCAI'77. Morgan Kaufmann, 1977, pp. 933–939 (cit. on p. 26).

[For81] Charles Forgy. *OPS5 User's Manual*. CMU-CS-81-135. Carnegie Mellon University, 1981 (cit. on p. 26).

[For82] Charles L. Forgy. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem." In: *Artificial Intelligence* 19.1 (1982), pp. 17–37 (cit. on p. 24).

[Fri10] Karl Fridolf. *Fire Evacuation in Underground Transportation Systems: A Review of Accidents and Empirical Research*. Tech. rep. Department of Fire Safety Engineering and Systems Safety, Lund University, 2010 (cit. on p. 12).

[FRV10] Marco Falda, Francesca Rossi, and Kristen Brent Venable. "Dynamic Consistency of Fuzzy Conditional Temporal Problems." In: *Journal of Intelligent Manufacturing* 21.1 (2010), pp. 75–88 (cit. on p. 171).

[GA02] Antony Galton and Juan Carlos Augusto. "Two Approaches to Event Definition." In: *Proceedings of the International Conference on Database and Expert Systems Applications*. DEXA'02. Springer, 2002, pp. 547–556 (cit. on p. 21).

[GAC06] Vihang Garg, Raman Adaikkalavan, and Sharma Chakravarthy. "Extensions to Stream Processing Architecture for Supporting Event Processing." In: *Proceedings of the International Conference on Database and Expert Systems Applications*. DEXA'06. Springer, 2006, pp. 945–955 (cit. on p. 31).

[GD93] Stella Gatziu and Klaus R. Dittrich. "Events in an Active Object-Oriented Database System." In: *Proceedings of the International Workshop on Rules in Database Systems*. Springer, 1993, pp. 23–39 (cit. on p. 21).

[GD94] Stella Gatziu and Klaus R. Dittrich. "Detecting Composite Events in Active Database Systems Using Petri Nets." In: *Proceedings of the International Workshop on Research Issues in Data Engineering: Active Database Systems*. IEEE, 1994, pp. 2–9 (cit. on pp. 20, 21).

[GH13] Brendan Galloway and Gerhard P. Hancke. "Introduction to Industrial Control Networks." In: *IEEE Communications Surveys & Tutorials* 15.2 (2013), pp. 860–880 (cit. on p. 16).

[GJS92a] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. "Composite Event Specification in Active Databases: Model and Implementation." In: *Proceedings of the International Conference on Very Large Data Bases*. VLDB'92. Morgan Kaufmann, 1992, pp. 327–338 (cit. on p. 21).

[GJS92b] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. "Event Specification in an Active Object-oriented Database." In: *Proceedings of the International Conference on Management of Data*. SIGMOD'92. ACM, 1992, pp. 81–90 (cit. on p. 21).

[GJS93] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. "COMPOSE: A System For Composite Specification and Detection." In: *Advanced Database Systems*. Springer, 1993, pp. 3–15 (cit. on p. 21).

[Gra05] Tim Grant. "Unifying Planning and Control Using an OODA-based Architecture." In: *Proceedings of the Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*. SAICSIT'05. South African Institute for Computer Scientists and Information Technologists, 2005, pp. 159–170 (cit. on p. 140).

[H2] *H2 Database Engine*. URL: http://www.h2database.com/ (visited on 06/30/2014) (cit. on p. 255).

[Hau11] Steffen Hausmann. "A Uniform Approach for More Reactivity in Complex Event Processing." Ph.D. Workshop Paper at *International Conference on Distributed Event Based Systems*. DEBS'11. 2011. 6 pp. (Cit. on p. ix).

[Hau+13] Steffen Hausmann, Simon Brodt, Marco Bettelini, and François Bry. "Dynamic Emergency Management." In: *Fachzeitschrift für Information Management & Consulting* (2 2013): *Urban Solutions*, pp. 36–47 (cit. on pp. ix, 10, 11).

[HB13] Steffen Hausmann and François Bry. "Towards Complex Actions for Complex Event Processing." In: *Proceedings of the International Conference on Distributed Event Based Systems*. DEBS'13. ACM, 2013, pp. 135–146 (cit. on pp. ix, 137, 169).

[HBB11] Steffen Hausmann, Simon Brodt, and François Bry. *Dura: Concepts and Examples*. Technical Report. EMILI Deliverable D4.3. University of Munich, 2011. 58 pp. (cit. on pp. ix, 39).

[HBB12] Steffen Hausmann, Simon Brodt, and François Bry. *Modularization Mechanisms for Dura*. Technical Report. EMILI Deliverable D4.6. University of Munich, 2012. 39 pp. (cit. on pp. ix, 39, 249, 250).

[HRS98] Thomas A. Henzinger, Jean-François Raskin, and Pierre-Yves Schobbens. "The Regular Real-Time Languages." In: *Proceedings of the International Colloquium on Automata, Languages and Programming*. ICALP'98. Springer, 1998, pp. 580–591 (cit. on p. 28).

[HSQ] *HyperSQL*. URL: http://hsqldb.org (visited on 06/30/2014) (cit. on p. 255).

[IAE07] International Association of Emergency Managers. *Principles of Emergency Management*. 2007 (cit. on p. 9).

[IBM] IBM Corporation. *IBM Operational Decision Manager*. URL: http://www.ibm.com/software/products/en/odm (visited on 06/30/2014) (cit. on p. 26).

[IET14] Internet Engineering Task Force. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. 2014. URL: http://tools.ietf.org/html/rfc7230 (cit. on p. 34).

[ISO96] International Organization for Standardization. *ISO/IEC 14977:1996(e) Information Technology - Syntactic Metalanguage - Extended BNF*. 1996 (cit. on p. 279).

[IST] IST GmbH. *Aseri*. URL: http://aseri.ist-net.de/aseri/ (visited on 06/30/2014) (cit. on p. 10).

[KEP00] Gerhard Knolmayer, Rainer Endl, and Marcel Pfahrer. "Modeling Processes and Workflows by Business Rules." In: *Business Process Management, Models, Techniques, and Empirical Studies*. Springer, 2000, pp. 16–29 (cit. on pp. 35, 111).

[KGR12] Ramkumar Krishnan, Jonathan Goldstein, and Alex Raizman. *A Hitchhiker's Guide to Microsoft StreamInsight Queries*. Tech. rep. Microsoft Corporation, 2012 (cit. on pp. 24, 34).

[KLG07] Martin Kersten, Erietta Liarou, and Romulo Goncalves. "A Query Language for a Data Refinery Cell." In: *Proceedings of the International Workshop on Event-Driven Architecture, Processing and Systems*. EDA-PS'07. 2007 (cit. on p. 24).

[Koz+06] Alex Kozlenkov et al. “Prova: Rule-based Java Scripting for Distributed Web Applications.” In: *Proceedings of the International Conference on Current Trends in Database Technology*. EDBT’06. Springer, 2006, pp. 899–908 (cit. on p. 30).

[Kra+04] Jan Krákora, Libor Waszniowski, Pavel Pisa, and Zdeněk Hanzálek. “Timed Automata Approach to Real Time Distributed System Verification.” In: *Proceedings of the International Workshop on Factory Communication Systems*. 2004, pp. 407–410 (cit. on p. 28).

[KS09] Jürgen Krämer and Bernhard Seeger. “Semantics and Implementation of Continuous Sliding Window Queries over Data Streams.” In: *Transactions on Database Systems (TODS)* 34.1 (2009), 4:1–4:49 (cit. on p. 24).

[KS86] Robert A. Kowalski and Marek J. Sergot. “A Logic-based Calculus of Events.” In: *New Generation Computing* 4.1 (1986), pp. 67–95 (cit. on pp. 28, 30, 87).

[KSM05] Hubert Klüpfel, Michael Schreckenberg, and Tim Meyer-König. “Models for Crowd Movement and Egress Simulation.” In: *Traffic and Granular Flow*. Springer, 2005, pp. 357–372 (cit. on p. 10).

[Lan+13] Andreas Lanz, Roberto Posenato, Carlo Combi, and Manfred Reichert. “Controllability of Time-Aware Processes at Run Time.” In: *Proceedings of the International Conference on Cooperative Information Systems*. CoopIS’13. Lecture Notes in Computer Science 8185. Springer, 2013, pp. 39–56 (cit. on p. 171).

[LGI09] Erietta Liarou, Romulo Goncalves, and Stratos Idreos. “Exploiting the Power of Relational Databases for Efficient Stream Processing.” In: *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology*. EDBT’09. ACM, 2009, pp. 323–334 (cit. on p. 24).

[LS11] David Luckham and W. Roy Schulte. *Event Processing Glossary*. Event Processing Technical Society, 2011 (cit. on pp. 19, 43, 51).

[May13] Michael Mayer. “Introducing Semantics to Dura and its Sublanguages.” Diploma Thesis. University of Munich, 2013 (cit. on p. 274).

[McG+13] Kevin McGrattan et al. *Fire Dynamics Simulator Technical Reference Guide*. Tech. rep. Institue of Standards and Technology (NIST), 2013 (cit. on p. 10).

[Mic] Microsoft Corporation. *Microsoft StreamInsight*. URL: http://www.microsoft.com/en-us/sqlserver/solutions-technologies/business-intelligence/streaming-data.aspx (visited on 06/30/2014) (cit. on p. 24).

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1982. ISBN: 978-0-387-10235-1 (cit. on p. 35).

[Mir87] Daniel P. Miranker. "TREAT: A Better Match Algorithm for AI Production Systems." In: *Proceedings of the National Conference on Artificial Intelligence*. AAAI Press'87. AAAI Press, 1987, pp. 42–47 (cit. on p. 24).

[MM05] Paul Morris and Nicola Muscettola. "Temporal Dynamic Controllability Revisited." In: *Proceedings of the National Conference on Artificial Intelligence*. AAAI Press'05. AAAI Press, 2005, pp. 1193–1198 (cit. on p. 170).

[MMT91] Michael Merritt, Francesmary Modugno, and Marc R. Tuttle. "Time-Constrained Automata." In: *Proceedings of the International Conference on Concurrency Theory*. CONCUR'91. Springer, 1991, pp. 408–423 (cit. on p. 28).

[Mon] *MonetDB*. URL: http://www.monetdb.org (visited on 06/30/2014) (cit. on p. 255).

[MS02] Rob Miller and Murray Shanahan. "Some Alternative Formulations of the Event Calculus." In: *Computational Logic: Logic Programming and Beyond*. Springer, 2002, pp. 452–490 (cit. on p. 82).

[MS97] Masoud Mansouri-Samani and Morris Sloman. "GEM: A Generalized Event Monitoring Language for Distributed Systems." In: *Distributed Systems Engineering* 4.2 (1997), pp. 96–108 (cit. on pp. 20, 21).

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990. ISBN: 0-262-63181-4 (cit. on p. 273).

[NFP] National Fire Protectino Association. *Fire Investigation Summary Düsseldorf*. (Visited on 06/30/2014) (cit. on p. 12).

[OC00] Angelo Oddi and Amedeo Cesta. "Incremental Forward Checking for the Disjunctive Temporal Problem." In: *Proceedings of the European Conference on Artificial Intelligence*. ECAI'00. IOS Press, 2000, pp. 108–112 (cit. on p. 170).

[OL08] Martin Ouimet and Kristina Lundqvist. "The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering." In: *Journal of Universal Computer Science* 14.12 (2008), pp. 2007–2033 (cit. on p. 28).

[OMG11a] Object Management Group, Inc. *Business Process Model and Notation, v2.0*. 2011. URL: http://www.omg.org/spec/BPMN/2.0/ (visited on 06/30/2014) (cit. on p. 28).

[OMG11b] Object Management Group, Inc. *Unified Modeling Language, v2.4.1*. 2011. URL: http://www.omg.org/spec/UML/2.4.1/ (visited on 06/30/2014) (cit. on p. 28).

[Oraa] Oracle Corporation. *Java*. URL: http://www.java.com (visited on 06/30/2014) (cit. on pp. 249, 252).

[Orab] Oracle Inc. *Oracle Complex Event Processing: Lightweight Modular Application Event Stream Processing in the Real World*. URL: http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/oracle-37.pdf (visited on 06/30/2014) (cit. on pp. 24, 31).

[Pas05] Adrian Paschke. *ECA-RuleML: An Approach Combining ECA Rules with Temporal Interval-based KR Event/Action Logics and Transactional Update Logics*. Technical University Munich, 2005 (cit. on p. 83).

[Pas06] Adrian Paschke. "ECA-LP / ECA-RuleML: A Homogeneous Event-Condition-Action Logic Programming Language." In: *CoRR* abs/cs/0609143 (2006) (cit. on pp. 30, 35).

[Pas+12] Adrian Paschke, Harold Boley, Zhili Zhao, Kia Teymourian, and Tara Athan. "Reaction RuleML 1.0: Standardized Semantic Reaction Rules." In: *Proceedings of the 6th International Conference on Rules on the Web: Research and Applications*. RuleML'12. Springer, 2012, pp. 100–119 (cit. on p. 30).

[Pat05] Paula-Lavinia Pătrânjan. "The Language XChange: A Declarative Approach to Reactivity on the Web." Doctoral Thesis. Institute for Informatics, University of Munich, 2005 (cit. on p. 21).

[PD99] Norman W. Paton and Oscar Díaz. "Active Database Systems." In: *ACM Computing Surveys* 31.1 (1999), pp. 63–103 (cit. on pp. 20, 21, 93).

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN: 0-262-16209-1 (cit. on p. 273).

[PK09] Adrian Paschke and Alexander Kozlenkov. "Rule-Based Event Processing and Reaction Rules." In: *Proceedings of the International Symposium on Rule Interchange and Applications*. RuleML'09. Springer, 2009, pp. 53–66 (cit. on pp. 20, 36).

[PKB07] Adrian Paschke, Alexander Kozlenkov, and Harold Boley. "A Homogenous Reaction Rule Language for Complex Event Processing." In: *International Workshop on Event-driven Architecture, Processing and Systems*. EDA-PS'07. VLDB Endowment, 2007 (cit. on pp. 30, 33, 35).

[Pop+13] Olga Poppe, Sandro Giessl, Elke A. Rundensteiner, and Franiçois Bry. "The HIT Model: Workflow-Aware Event Stream Monitoring." In: *Transactions on Large-Scale Data- and Knowledge-Centered Systems*. Vol. 8290. Lecture Notes in Computer Science. Springer, 2013, pp. 26–50 (cit. on pp. 26–28).

[Pos] *PostgreSQL*. URL: http://www.postgresql.org (visited on 06/30/2014) (cit. on p. 255).

[PR07] Paolo Pialorsi and Marco Russo. *Introducing Microsoft LINQ*. First. Microsoft Press, 2007. ISBN: 978-0-7356-2391-0 (cit. on p. 34).

[Prz88] Teodor C. Przymusinski. "On the Declarative Semantics of Deductive Databases and Logic Programs." In: *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988, pp. 193–216 (cit. on p. 227).

[PVY07] Bart Peintner, Kristen Brent Venable, and Neil Yorke-Smith. "Strong Controllability of Disjunctive Temporal Problems with Uncertainty." In: *Proceedings of the International Conference on Principles and Practice of Constraint Programming*. CP'07. Springer, 2007, pp. 856–863 (cit. on p. 170).

[RVY04] Francesca Rossi, Kristen Brent Venable, and Neil Yorke-Smith. "Controllability of Soft Temporal Constraint Problems." In: *Principles and Practice of Constraint Programming*. Vol. 3258. Lecture Notes in Computer Science. Springer, 2004, pp. 588–603 (cit. on p. 170).

[RVY06] Francesca Rossi, Kristen Brent Venable, and Neil Yorke-Smith. "Uncertainty in Soft Temporal Constraint Problems: A General Framework and Controllability Algorithms for the Fuzzy Case." In: *Journal Of Artificial Intelligence Research* 27.1 (2006), pp. 617–674 (cit. on p. 170).

[San] Sandia National Laboratories. *Jess, the Rule Engine for the Java Platform*. URL: http://www.jessrules.com (visited on 06/30/2014) (cit. on p. 26).

[SAP] SAP AG. *SAP Event Stream Processor*. URL: http://www.sap.com/pc/tech/database/software/sybase-complex-event-processing/index.html (visited on 06/30/2014) (cit. on p. 24).

[SAP13] SAP AG. *Programmers Guide: SAP Event Stream Processor 5.1*. Tech. rep. 2013 (cit. on p. 24).

[SAS08] Kay-Uwe Schmidt, Darko Anicic, and Roland Stühmer. "Event-driven Reactivity: A Survey and Requirements Analysis." In: *International Workshop on Semantic Business Process Management*. SBPM'08. CEUR, 2008 (cit. on p. 43).

[SB05] Marco Seiriö and Mikael Berndtsson. "Design and Implementation of an ECA Rule Markup Language." In: *Proceedings of the International Conference on Rules and Rule Markup Languages for the Semantic Web*. RuleML'05. Springer, 2005, pp. 98–112 (cit. on p. 21).

[SB10a] Nikolaus Seifert and Marco Bettelini. *Specific Report for Use Case II, Public Transport*. EMILI Deliverable D3.1 Annex B. ASIT Ltd., 2010. 97 pp. (cit. on pp. 17, 52).

[SB10b] Nikolaus Seifert and Marco Bettelini. *Use Cases Requirements Analysis and Specification (Main Report)*. EMILI Deliverable D3.1. ASIT Ltd., 2010. 111 pp. (cit. on pp. 11, 13, 15, 17, 41, 52, 74, 93, 95).

[SBR11a] Nikolaus Seifert, Marco Bettelini, and Samuel Rigert. *Concrete Use Case Models (Main Report)*. Deliverable D3.2. ASIT Ltd., 2011 (cit. on pp. 9, 13–15, 41, 74, 93, 95, 117, 118).

[SBR11b] Nikolaus Seifert, Marco Bettelini, and Samuel Rigert. *Emergency Management an Rules in Control Systems of Critical Infrastructures*. EMILI Deliverable D3.2 Annexe A. ASIT Ltd., 2011. 42 pp. (cit. on pp. 9, 14, 117).

[SBR11c] Nikolaus Seifert, Marco Bettelini, and Samuel Rigert. *Simulation Methodology*. EMILI Deliverable D3.2. ASIT Ltd., 2011. 33 pp. (cit. on p. 11).

[Sch04] Sebastian Schaffert. "Xcerpt: A Rule-Based Query and Transformation Language for the Web." Doctoral Thesis. Institute for Informatics, University of Munich, 2004 (cit. on p. 43).

[Sch11] Maximilian Scherr. "Desugaring Dura— Compiling a High-Level Event Processing Language." Diploma Thesis. University of Munich, 2011 (cit. on p. 213).

[SFS12] Keith Stouffer, Joe Falco, and Karen Scarfone. *Guide To Industrial Control Systems (ICS) Security*. CreateSpace, 2012. ISBN: 978-1-4701-5814-9 (cit. on pp. 15, 16).

[Sha99] Murray Shanahan. "The Event Calculus Explained." In: *Artificial Intelligence Today*. Vol. 1600. Lecture Notes in Computer Science. Springer, 1999, pp. 409–430 (cit. on pp. 82, 87).

[She85] John C. Shepherdson. "Negation as Failure. II." In: *Journal of Logic Programming* 2.3 (1985), pp. 185–202 (cit. on pp. 57, 72).

[SKoo] Kostas Stergiou and Manolis Koubarakis. "Backtracking Algorithms for Disjunctions of Temporal Constraints." In: *Artificial Intelligence* 120.1 (2000), pp. 81–117 (cit. on pp. 163, 169).

[Sto+11] Nenad Stojanovic et al. "Semantic Complex Event Reasoning— Beyond Complex Event Processing." In: *Foundations for the Web of Information and Services*. Springer, 2011, pp. 253–279 (cit. on p. 31).

[SV98] Eddie Schwalb and Lluís Vila. "Temporal Constraints: A Survey." In: *Constraints* 3.2 (1998), pp. 129–149 (cit. on pp. 163, 169, 170).

[TIB] TIBCO Software Inc. *TIBCO BusinessEvents*. URL: `http://www.tibco.de/products/event-processing/complex-event-processing/businessevents/default.jsp` (visited on 06/30/2014) (cit. on pp. 26, 31, 83).

[TPo3] Ioannis Tsamardinos and Martha E Pollack. "Efficient Solution Techniques for Disjunctive Temporal Reasoning Problems." In: *Artificial Intelligence* 151.1-2 (2003), pp. 43–89 (cit. on p. 170).

[TPR03] Ioannis Tsamardinos, Martha E. Pollack, and Sailesh Ramakrishnan. "Assessing the Probability of Legal Execution of Plans with Temporal Uncertainty." In: *Proceedings of the Workshop on Planning Under Uncertainty and Incomplete Information*. 2003 (cit. on p. 170).

[Tra] TraffGo HT GmbH. *PedGo*. URL: `http://www.traffgo-ht.com/de/pedestrians/products/pedgo/index.html` (visited on 06/30/2014) (cit. on p. 10).

[TRP12] Kia Teymourian, Malte Rohde, and Adrian Paschke. "Fusion of Background Knowledge and Streams of Events." In: *Proceedings of the International Conference on Distributed Event-Based Systems*. DEBS'12. ACM, 2012, pp. 302–313 (cit. on p. 31).

[TVP03] Ioannis Tsamardinos, Thierry Vidal, and Martha E. Pollack. "CTP: A New Constraint-Based Formalism for Conditional, Temporal Planning." In: *Constraints* 8.4 (2003), pp. 365–388 (cit. on pp. 105, 165, 167, 170).

[Ven+10] Kristen Brent Venable, Michele Volpato, Bart Peintner, and Neil Yorke-Smith. "Weak and Dynamic Controllability of Temporal Problems with Disjunctions and Uncertainty." In: *Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*. COPLAS'10. 2010 (cit. on p. 170).

[VF99] Thierry Vidal and Helene Fragier. "Handling Contingency in Temporal Constraint Networks: From Consistency to Controllabilities." In: *Journal of Experimental and Theoretical Artificial Intelligence* 11.1 (1999), pp. 23–45 (cit. on pp. 105, 141, 170).

[Vra+10] Sanja Vraneš et al. *Specific Report for Use Case I, Airport*. EMILI Deliverable D3.1 Annex A. Institute Mihailo Pupin, 2010. 92 pp. (cit. on pp. 12, 13, 52, 117).

[VY05] Kristen Brent Venable and Neil Yorke-Smith. "Disjunctive Temporal Planning with Uncertainty." In: *Proceedings of the International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 2005, pp. 1721–1722 (cit. on p. 170).

[W3C07] Martin Gudgin et al. *SOAP Version 1.2 Part 1: Messaging Framework*. 2007. URL: http://www.w3.org/TR/soap12-part1/ (cit. on p. 34).

[W3C10] Anders Berglund et al. *XML Path Language (XPath) 2.0*. 2010. URL: http://www.w3.org/TR/xpath20/ (cit. on p. 55).

[Wat85] Donald A. Waterman. *A Guide to Expert Systems*. Addison-Wesley, 1985. ISBN: 0-201-08313-2 (cit. on p. 73).

[WBG08] Karen Walzer, Tino Breddin, and Matthias Groch. "Relative Temporal Constraints in the Rete Algorithm for Complex Event Detection." In: *Proceedings of the International Conference on Distributed Event-based Systems*. DEBS'08. ACM, 2008, pp. 147–155 (cit. on pp. 24, 31).

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. "High-performance Complex Event Processing over Streams." In: *Proceedings of the International Conference on Management of Data*. SIGMOD'06. ACM, 2006, pp. 407–418 (cit. on p. 21).

[Wen09] John F. Wendt. *Computational Fluid Dynamics*. 3rd. Springer, 2009. ISBN: 978-3-540-85055-7 (cit. on p. 10).

[WGB08] Karen Walzer, Matthias Groch, and Tino Breddin. "Time to the Rescue - Supporting Temporal Reasoning in the Rete Algorithm for Complex Event Processing." In: *Proceedings of the International Conference on Database and Expert Systems Applications*. DEXA'08. Springer, 2008, pp. 635–642 (cit. on p. 24).

[Wik14] Wikipedia. *Daegu Metro Fire — Wikipedia, The Free Encyclopedia*. 2014. URL: http://en.wikipedia.org/w/index.php?title=Daegu_metro_fire&oldid=596315874 (visited on 06/30/2014) (cit. on p. 13).

[WRE11] Di Wang, Elke A. Rundensteiner, and Richard T. Ellison III. "Active Complex Event Processing over Event Streams." In: *Proceedings of the VLDB Endowment* 4.10 (2011), pp. 634–645 (cit. on p. 35).

[Xte] Eclipse Foundation, Inc. *Xtext— Language Development Made Easy*. URL: http://www.eclipse.org/Xtext/ (visited on 06/30/2014) (cit. on pp. 253, 259, 279).

[Xte14] Eclipse Foundation, Inc. *Xtext Documentation*. Tech. rep. 2014 (cit. on p. 279).

[Zlo77] Moshe M. Zloof. "Query-by-Example: A Data Base Language." In: *IBM Systems Journal* 16.4 (1977), pp. 324–343 (cit. on p. 42).

[ZU99] Detlef Zimmer and Rainer Unland. "On the Semantics of Complex Events in Active Database Management Systems." In: *Proceedings of the International Conference on Data Engineering*. ICDE'99. IEEE, 1999, pp. 392–399 (cit. on p. 20).

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of October 21, 2014 (`classicthesis` version 4.1).

DECLARATION

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.

München, 7. August 2014

Steffen Hausmann