



Hasso Plattner Institute for Software Systems Engineering
System Analysis and Modeling Group

Thesis

Modeling Collaborations in Adaptive Systems of Systems

Dissertation
zur Erlangung des akademischen Grades
"doctor rerum naturalium"
(Dr. rer. nat.)
in der Wissenschaftsdisziplin
"Systemanalyse und Modellierung"

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Sebastian Wätzoldt

Potsdam, October 11, 2016

This work is licensed under a Creative Commons License:
Attribution – Noncommercial – Share Alike 4.0 International
To view a copy of this license visit
<http://creativecommons.org/licenses/by-nc-sa/4.0/>

Published online at the
Institutional Repository of the University of Potsdam:
URN [urn:nbn:de:kobv:517-opus4-97494](http://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-97494)
<http://nbn-resolving.org/urn:nbn:de:kobv:517-opus4-97494>

Abstract

Recently, due to an increasing demand on functionality and flexibility, beforehand isolated systems have become interconnected to gain powerful adaptive System of Systems (SoS) solutions with an overall robust, flexible and emergent behavior. The adaptive SoS comprises a variety of different system types ranging from small embedded to adaptive cyber-physical systems. On the one hand, each system is independent, follows a local strategy and optimizes its behavior to reach its goals. On the other hand, systems must cooperate with each other to enrich the overall functionality to jointly perform on the SoS level reaching global goals, which cannot be satisfied by one system alone. Due to difficulties of local and global behavior optimizations conflicts may arise between systems that have to be solved by the adaptive SoS.

This thesis proposes a modeling language that facilitates the description of an adaptive SoS by considering the adaptation capabilities in form of feedback loops as first class entities. Moreover, this thesis adopts the Models@runtime approach to integrate the available knowledge in the systems as runtime models into the modeled adaptation logic. Furthermore, the modeling language focuses on the description of system interactions within the adaptive SoS to reason about individual system functionality and how it emerges via collaborations to an overall joint SoS behavior. Therefore, the modeling language approach enables the specification of local adaptive system behavior, the integration of knowledge in form of runtime models and the joint interactions via collaboration to place the available adaptive behavior in an overall layered, adaptive SoS architecture.

Beside the modeling language, this thesis proposes analysis rules to investigate the modeled adaptive SoS, which enables the detection of architectural patterns as well as design flaws and pinpoints to possible system threats. Moreover, a simulation framework is presented, which allows the direct execution of the modeled SoS architecture. Therefore, the analysis rules and the simulation framework can be used to verify the interplay between systems as well as the modeled adaptation effects within the SoS. This thesis realizes the proposed concepts of the modeling language by mapping them to a state of the art standard from the automotive domain and thus, showing their applicability to actual systems. Finally, the modeling language approach is evaluated by remodeling up to date research scenarios from different domains, which demonstrates that the modeling language concepts are powerful enough to cope with a broad range of existing research problems.

Zusammenfassung

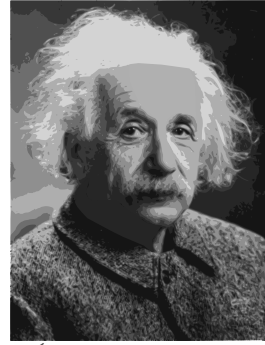
Seit einiger Zeit führen ein ansteigender Bedarf nach erweiterter Systemfunktionalität und deren flexible Verwendung zu vernetzten Systemen, die sich zu einem übergeordneten adaptiven System von Systemen (SoS) zusammenschließen. Dieser SoS Zusammenschluss zeigt ein gewünschtes, robustes und flexibles Gesamtverhalten, welches sich aus der Funktionalität der einzelnen Systeme zusammensetzt. Das SoS beinhaltet eine Vielzahl von verschiedenen Systemarten, die sich von eingebetteten bis hin zu Cyber-Physical Systems erstrecken. Einerseits optimiert jedes einzelne System sein Verhalten bezüglich lokaler Ziele. Andererseits müssen die Systeme miteinander interagieren, um neue, zusammengesetzte Funktionalitäten bereitzustellen und damit vorgegebene SoS Ziele zu erreichen, welche durch ein einzelnes System nicht erfüllt werden können. Die Schwierigkeit besteht nun darin, Konflikte zwischen lokalen und globalen Verhaltensstrategien zwischen Systemen innerhalb des SoS zu beseitigen.

Diese Doktorarbeit stellt eine Modellierungssprache vor, welche für die Beschreibung von adaptiven SoS geeignet ist. Dabei berücksichtigt die Modellierungssprache die Adaptionslogik des SoS in Form von periodischen Adaptationsschleifen als primäres Sprachkonstrukt. Außerdem übernimmt diese Arbeit den Models@runtime Ansatz, um verfügbares Systemwissen als Laufzeitmodelle in die Adaptationslogik des Systems zu integrieren. Weiterhin liegt der Fokus der Modellierungssprache auf der Beschreibung von Systeminteraktionen innerhalb des SoS. Dies ermöglicht Schlussfolgerungen von individuellem Systemverhalten sowie deren Aggregation zu kollaborativem Verhalten im Kontext von Systeminteraktionen im SoS. Damit unterstützt die entwickelte Modellierungssprache die Beschreibung von lokalem adaptivem Verhalten, die Integration von Wissen über die Modellierung von Laufzeitmodellen und Systeminteraktionen in Form von kollaborativem Verhalten. Alle drei Aspekte werden in die adaptive SoS Architektur integriert.

Neben der entwickelten Modellierungssprache führt diese Doktorarbeit Analyseregeln zur Untersuchung des modellierten SoS ein. Diese Regeln ermöglichen die Erkennung von Architekturmustern und möglichen Schwächen im Systementwurf. Zusätzlich wird eine Simulationsumgebung für die Modellierungssprache präsentiert, welche die direkte Ausführung von einer modellierten SoS Architektur erlaubt. Die Analyseregeln und die Simulationsumgebung dienen demnach sowohl der Verifizierung von Systeminteraktionen als auch der spezifizierten Adaptationslogik innerhalb des SoS. Die vorliegende Arbeit implementiert die vorgestellten Konzepte der Modellierungssprache durch deren Abbildung auf einen aktuellen Standard im Automobilbereich und zeigt damit die Anwendbarkeit der Sprache auf gegenwärtige Systeme. Zum Schluss findet eine Evaluierung der Modellierungssprache statt, wobei aktuelle Forschungsszenarien aus unterschiedlichen Bereichen erneut mit der vorgestellten Sprache modelliert werden. Dies zeigt, dass die Modellierungskonzepte geeignet sind, um weite Bereiche existierender Forschungsprobleme zu bewältigen.

Acknowledgement

This thesis would not have been possible without my great family, friends and colleagues. First and foremost, I am very grateful for the endless support of my parents Marco and Evelin Wätzoldt, who enabled my studies and never doubted about my scientific career. Furthermore, I like to express my gratitude to colleagues from the System Analysis and Modeling group for fruitful teamwork and countless discussions about my research. Especially, I am very thankful to Thomas Beyhl for keeping me back to real life, whenever I had a new weird idea that would far extend the focus of my research topic, for providing tool support what enables the implementation of the simulation environment described in this thesis, and for being a great office colleague with many patience listening my ideas in combination with constructive feedback. I thank Stefan Neumann for the fruitful discussions about embedded-systems engineering, real-time analysis and the good teamwork in the laboratory. I would like to thank the professors and colleagues from the HPI research school for their sincere and constructive feedback. For their assistance in organizational and bureaucratic matters, I thank Kerstin Miers and Sabine Wagner. I also thank the students, who refined the tool support, namely Paul Geppert and Michael Fabian. For great proofreading, I am very thankful to Inga Melching, who rigorously read this very long thesis. A deep, great thanks is for my friend Tonio-Erik Schultze, who supports me in all circumstances of all the highs and lows during my life. Last but not least, I am deeply grateful to my partner, Friederike Melching, who is my quiescent point in this hectic world and, even under difficult circumstances, was able to motivate me to finish this thesis.



Albert Einstein ©

"The only source of knowledge is experience."
Albert Einstein¹

¹Picture from <https://openclipart.org/detail/213786/albert-einstein>

Contents

1. Introduction	1
1.1. Research Challenges and Goals	2
1.2. Contributions	7
1.3. Structure	10
2. Preliminaries	13
2.1. System Types	13
2.1.1. Self-Adaptive System	13
2.1.2. Cyber-Physical System	16
2.1.3. Networked Cyber-Physical System	17
2.1.4. System of Systems	18
2.1.5. Internet of Things	18
2.1.6. Adaptive Systems of Systems	19
2.2. Model-Driven Engineering	20
2.2.1. Model	21
2.2.2. Metamodel	22
2.2.3. Runtime Model	23
2.2.4. Model Management	25
2.2.5. Model Manipulation	26
2.3. Eurema Modeling Language	29
2.4. Collaborations in SoS	32
2.5. Running example	35
3. Modeling Language Requirements	39
3.1. Characteristics	39
3.2. Requirements	45
3.3. State of the Art	52
4. Overview	57
4.1. Deurema Modeling Language	59
4.1.1. Modeling the Adaptation Logic	60
4.1.2. Knowledge as Runtime Models	62
4.1.3. Modeling Collaborations	63
4.1.4. Modeling the Adaptive SoS Architecture	65
4.2. Deurema Analysis	66
4.3. Deurema Simulation	67
4.4. Deurema Realization	68

5. Deurema Modeling Language	71
5.1. Deurema Core Concepts	71
5.2. Deurema Runtime Models	75
5.2.1. Runtime Model Categorization	77
5.2.2. Runtime Model Integration	83
5.2.3. Runtime Model Example	85
5.2.4. Runtime Model Metamodel	87
5.2.5. Runtime Model Summary	88
5.3. Deurema Module Templates	89
5.3.1. Template Variables and Runtime Model Views	92
5.3.2. Feedback Loop Module Template	93
5.3.3. Software Module Template	103
5.3.4. Application Module Template	106
5.3.5. Behavior Module Template	115
5.4. Deurema Adaptive System Architecture	123
5.5. Deurema Collaboration	127
5.5.1. Collaboration Structure	129
5.5.2. Collaboration Knowledge	131
5.5.3. Collaboration Choreography	132
5.5.4. Collaboration Role Interfaces	140
5.5.5. Collaboration Role Mapping	141
5.5.6. Collaboration Deployment	142
5.6. Deurema Reflection, Reconfiguration and Adaptation	145
5.6.1. Runtime Reconfiguration	150
5.6.2. Runtime Adaptation	153
5.6.3. Meta-Adaptation	155
5.7. Deurema Modeling Language Discussion	158
5.7.1. Summary of Deurema concepts	158
5.7.2. Design Decisions	163
5.7.3. Coverage of Requirements	165
6. Analysis	167
6.1. Basic Metrics	169
6.1.1. Causality	169
6.1.2. Knowledge	177
6.1.3. Adaptation Purpose	179
6.2. Complex Metrics	181
6.2.1. Combining Causality and Adaptation Purpose	181
6.2.2. Combining Knowledge and Adaptation Purpose	183
6.2.3. Combining Knowledge and Causality	184
6.2.4. Complex Analysis Rule Combination	186
6.3. Architectural Patterns and Design Smells	186
6.4. Discussion	190

7. Simulation	193
7.1. Execution State Models	195
7.1.1. Modules and Interactions	195
7.1.2. Module Template Elements	199
7.2. Interpreter	201
7.2.1. Execution Semantic Module	201
7.2.2. Execution Semantic Deurema Elements	202
7.2.3. Execution Semantic Behavior Model	203
7.2.4. Execution Semantic Interaction	205
7.3. Simulator	205
7.4. Simulation Run Example	208
7.5. Runtime Analysis	210
7.6. Discussion	213
8. Realization	215
8.1. Scope	216
8.2. AUTOSAR	217
8.3. Systems and Modules	220
8.4. Software Module Template	221
8.5. Application Module Template	223
8.6. Feedback Loop Module Template	226
8.7. Behavior Module Template	226
8.8. Discussion	227
8.8.1. Deurema Modeling Process	227
8.8.2. Software Tools	231
9. Application	233
9.1. Case Studies	233
9.1.1. Traffic Monitoring System	234
9.1.2. Smart Home	238
9.2. Ensemble-Based Component Systems	241
10. Related Work	245
10.1. General Purpose Modeling Languages	245
10.2. Domain Specific Languages and Approaches	250
10.3. Frameworks and Patterns	259
10.4. Experience from the Research Group	264
10.5. Discussion	268
11. Conclusion	269
11.1. Discussing Goals and Contribution	269
11.2. Modeling Language Requirements and Deurema	270
11.3. Future Work	272

Bibliography	275
Author's References	275
Other References	276
Appendix A. Deurema Metamodel	291
Appendix B. Interaction Message	301
Appendix C. Analysis Rules	305
C.1. Annotation Types	307
C.2. Analysis Rules	309
Appendix D. Simulation Rules	341
D.1. Simulation Rules	341
D.2. Simulation Metrics	362
List of Figures	367
List of Tables	375
List of Abbreviations	377

1. Introduction

Embedded software-intensive systems can be found in many application domains such as mobile devices, vehicles, avionics, buildings, or production systems [45, 59, 69, 71, 124, 187]. Recently, due to an increasing demand on functionality and flexibility, such beforehand isolated systems have become interconnected to gain powerful System of Systems (SoS) solutions with an overall robust, flexible and emergent behavior. On the one hand, SoS are envisioned to realize modern demands on intelligent systems such as smart cities that optimize electrical power production in so-called *smart grids* [112] or dynamically reorganize the traffic flow avoiding traffic jams [69, 71, 112]. Another important domain are *smart homes* [45, 57] that are designed to adapt their pool of functionality to available resources such as sensors and actuators, different users, and situations to increase comfortability, security, and to save costs. Additionally, SoS became important for managing catastrophic scenarios such as search and rescue or firefighting. There, unreliable sensor networks, mobile devices and different software systems must be spontaneously interconnected and coordinated to handle the catastrophic situation appropriately. On the other hand, the inherent distributed nature of SoS, the complexity of contained systems as well as their interaction raises challenges for the development, understanding and execution of such systems [58, 81, 111, 127].

In the context of this thesis, there are two important research areas tackling the complexity of SoS. First, the Model-Driven Engineering (MDE) approach provides standards and techniques, where models are used as first class entities during the system development and lifetime describing key aspects of the (software) system. In general, models raise the level of abstraction, which enables the handling of complexity [75]. Furthermore, analysis techniques can be applied on models to verify the overall system against its requirements and finally, model transformation approaches as for example code generation support the implementation of complex systems. This thesis considers the MDE research direction that is related to the use of runtime models, where development models are kept alive during system execution, describing key artifacts of interest, and manipulating the system behavior at runtime [38]. Such runtime models are used to specify the available knowledge within the system, where the provided system functionality operates on this knowledge base.

The second research direction tackling the complexity of SoS is in the context of (self-) adaptive systems. Such systems are able to react on changing environmental conditions, requirements or user interaction by adapting its runtime behavior accordingly [156]. Consequently, adaptive systems can be designed to show an overall robust system behavior that is able to cope with software and hardware errors as well as uncertain situations. Unfortunately, new demands concerning the modeling of such adaptive system behavior and the interplay between different adaptive systems inside a SoS arise [49, 162].

A SoS contains several independent and diverse systems, which further show adaptive capabilities coping with changing environmental conditions. Additionally, the isolated system solutions have to cooperate with each other on the SoS level to reach global goals that cannot be fulfilled by one system alone. Thereby, each system has to keep local goals in mind, which may generate conflicts between local and global system goal optimizations. The cooperation between systems in combination of the local adaptive capabilities lead to an

overall emergent *adaptive SoS* behavior, where the SoS dynamically establishes and reorganizes collaboration links between systems. This thesis focus on modeling such adaptive SoS, which comprises the local adaptation capabilities of the independent systems, the representation of the available knowledge using the runtime model approach, and the explicit specification of system interactions by means of collaborations between independent and possibly distributed systems. Modeling the collaboration aspects within the adaptive SoS as first class entities further enables the analysis of the overall emergent behavior as well as helps understanding the impact of system interactions for the local system functionality. The concrete challenges of adaptive SoS modeling and the corresponding goals for this thesis are discussed in the next section. Afterwards, the contributions of this thesis are outlined.

1.1. Research Challenges and Goals

In this section, the main research challenges and goals for this thesis are discussed as they are depicted in the overview in Figure 1.1. The *National Academy of Science and Engineering* emphasizes the evolution of self-contained embedded systems to Cyber-Physical Systems (CPS) into the Internet of Things (IoT) [69]. As a consequence, single system solutions become interconnected and finally, merge their pool of functionality into an overall System of Systems (SoS) [130]. A key characteristic for embedded systems is the interaction with the environment that can be a human or another system. Usually, the interaction is enabled by physical actors that can directly manipulate parts of the system surroundings. Beside the influence of the environment via actors, embedded systems are often equipped with sensors to retrieve status information from the environment. Because *"the knowledge of the system [and its environment] is uncertain and incomplete"* [54], embedded systems must permanently react on changing environmental conditions, e. g., user interaction, or changes in its own system state, e. g., hardware status or battery level. Therefore, embedded systems are usually designed in form of a periodical control loop following the sense, compute, act paradigm [54, 70, 125], which enables an adaptive and robust system behavior. Those control loops are designed close to the hardware capabilities of the embedded system, where the software part mainly focuses on the realization of the control loop functionality. Because of the system evolution towards CPS, the software part gains more intention realizing the domain logic of the system, which on the one hand, extends the pool of functionalities of the system and on the other hand, exits the focus from the hardware towards software intensive system parts. Kephart et al. [110] transfer the control loop idea to software intensive systems in the autonomic computing domain. A so-called feedback loop senses the underlying software system and optimizes it according to given system requirements. Thereby, Kephart et al. [110] propose a reference architecture for a feedback loop that consists of four subsequently performed adaptation activities named *Monitor, Analyze, Plan, and Execute (MAPE)*. The MAPE feedback loop approach follows the same idea of the sense, compute, act paradigm from the control engineering domain, but focuses rather on the software capabilities of the system than on the plant. Therefore, the monitoring step senses the underlying software system to retrieve the current software system state. The analysis activity checks whether an adaptation is necessary or not and triggers the corresponding planning activity appropriately. Consequently, analysis and planning are the compute part of the feedback loop. Finally, the execute activity corresponds to the act step in the control engineering paradigm and forces the planned adaptation steps to the running software system. Consequently, the autonomic system becomes reactive at the software level showing similar adaptive and robustness properties as envisioned

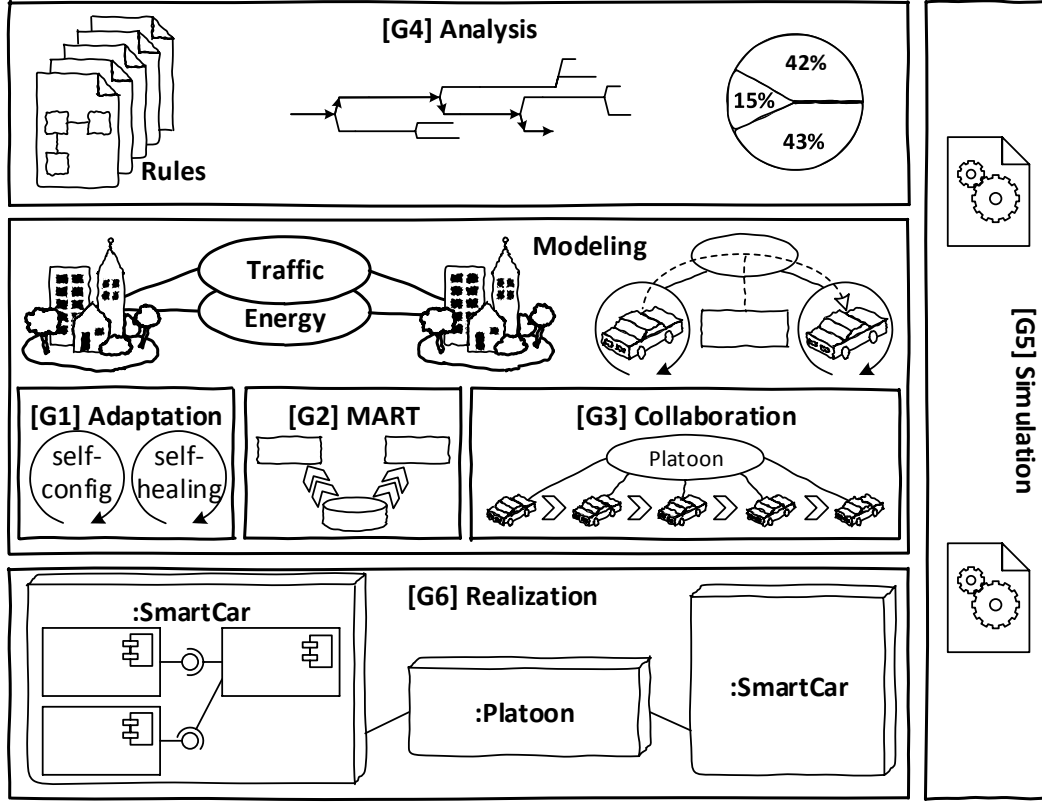


Figure 1.1: Overview of goals

by the control loop paradigm. Moreover, Holland stated *"it is feasible to understand any System of Systems as an artificial complex adaptive system"* [101]. First, because a SoS consists of several interconnected and diverse system types ranging from small embedded systems over cyber-physical systems to software intensive systems, it inherits such system characteristics as adaptiveness, collaborativity and flexibility. Second, a SoS is able to combine single capabilities by system interaction to offer new services or functionalities that cannot be realized by a single system alone [39, 81, 130, 140]. Third, the complexity of the overall SoS raises challenges in describing, analyzing and understanding the emerged system behavior. The focus of this thesis is the modeling of the adaptive, interactive, emerged system behavior according to the statement of Kilicay-Ergin et al., who say that *"the challenge is to identify the right collection of systems that will collaborate to satisfy the client requirements"* [111]. For emphasizing the emergent adaptive behavior aspect from collaborating systems within the SoS, the term *adaptive SoS* is used for the rest of this thesis. Thus, the adaptation capabilities of an adaptive SoS, which emerges from the contained systems as well as their collaborations, must be explicitly considered, which leads to the following goal of this thesis.

Thesis goal G1: *One goal of this thesis is to consider the heterogeneous characteristics of an adaptive System of Systems and to model the adaptation capabilities as first class entities.*

Beside the first goal of understanding SoS characteristics and modeling the adaptive system architecture, this thesis focuses on the data (knowledge) inside an adaptive SoS. Combining the ideas of explicitly modeling the adaptation logic in form of feedback loops and model-driven techniques that uses models as primary artifacts lead to the Models@runtime (MART)

approach [38]. Runtime models are an abstract representation of key aspects from the running software system defining the notion of knowledge in the reference MAPE-K feedback loop architecture from Kephart et al. [110]. Furthermore, runtime models are maintained during system execution and represent key points of interest of the running system. The information in the runtime model changes according to the observed runtime phenomena of the system, where a change in the system state or environment triggers a corresponding update in the runtime model representation. Beside the update of system observations to its representation in the runtime model, the MART approach enables a manipulation of the system by directly working on the representation in the runtime models. Therefore, a change in the runtime model leads to a corresponding change in the running system. This two way synchronization of system observation to runtime models and vice versa is called *causal connection* [38]. Due to the causal connection, all changes in the runtime model will affect the running system itself, which allows an adaptive control of the software system on a much higher level of abstraction directly working with the corresponding runtime models [32, 38]. Furthermore, because adaptive SoS are complex and inherent distributed [81, 130] new challenges concerning the knowledge management arise. For example, if systems inside the adaptive SoS interact with each other to reach common goals, they must communicate with each other or share information for coordination purposes to agree on a common knowledge base for further task processing. In general, systems are not willing to provide their complete data to the outside and restrict the amount to the minimum that is needed to perform the common tasks. This leads to partial available knowledge and incomplete views within systems. Furthermore, the knowledge may become outdated over time if systems miss the updating of it via the joint interactions. Thus, runtime models may exist in different versions distributed in systems throughout the adaptive SoS. As a consequence, the adaptive SoS has to deal with partial available knowledge as well as outdated or versioned runtime models [86], which leads to the following thesis goal.

Thesis goal G2: *One goal of this thesis is to consider the role of knowledge, which is captured in runtime models, and to integrate it in a modeling approach for an adaptive System of Systems.*

Having an understanding about adaptive SoS characteristics and the used knowledge raises a new research question for describing such kind of systems. Stankovic et al. emphasize that building adaptive SoS *"requires a deep understanding of how to model and analyze large-scale systems' behaviors, which necessitate large-scale coordination and cooperation"* [162]. The explicit specification of feedback loops, which describe the adaptation logic, raises the level of abstraction and decouples self-adaptive capabilities from the domain logic [49, 110, 175]. There are different kinds of self-* systems¹ ranging from autonomous self-organizing to hierarchical self-adaptive systems [58]. Besides single feedback loops controlling the adaptation of the core system, also multiple feedback loops realizing different adaptation concerns and distributed, collaborating feedback loops must be considered to achieve the desired self-adaptation for the whole emergent adaptive SoS behavior. Additionally, in most cases, each system has some individual self-* capabilities [156] and is aware of its environment to react on changing demands properly [58]. However, the overall interconnected system is open in the sense that new systems (e.g., mobile devices) may join or leave over time by providing or removing additional functionalities. Furthermore, individual systems cooperate with other systems to

¹Salehie et al. [156] subsume different adaptive capabilities of systems, as for example self-awareness, self-optimizing, self-healing, or self-configuring, to so-called *self-** properties, which further leads to the corresponding *self-** system realizing this property.

reach overall system goals [45]. On the one hand, because of the dynamics within the system and an upfront unknown number of system participants, there is a broad range of possible coordination schemes between systems ranging from central coordination schemes to fully self-organizing solutions. On the other hand, there are high demands on reliability, availability, and robustness of systems, which leads to an overall resilient, adaptive SoS. Consequently, the inherent distributed and flexible nature of adaptive SoS raises new demands of modeling, understanding and analyzing system interactions [19, 162, 183]. The following thesis goal focuses on the modeling aspect of system interactions as follows:

Thesis goal G3: *One goal of this thesis is the development of a modeling language that suits the systematic specification of system interactions by means of collaborations as first class entities in adaptive Systems of Systems.*

Unfortunately, there is currently a lack of a clear understanding of effect propagations via manipulating runtime models by adaptation activities inside one feedback loop. Even worse, the impact of the effect propagation for the whole adaptive SoS becomes unclear due to system interactions [9, 162]. This holds especially for multiple collaborating feedback loops that share parts of the local knowledge and therefore influence each other by the arising indirect coupling over the shared knowledge base. As a consequence, effect propagation between distributed and collaborating feedback loops must be considered. A nonambiguous modeling language enables the explicit specification of feedback loops together with their interactions. This specification can be used for further analysis of the overall system for understanding or predicting emergent behavior and verification of given requirements [19]. This verification of system behavior can be performed during the development (e. g., by applying static analysis techniques) or during the lifetime of the adaptive SoS (e. g., by monitoring of system invariants or runtime analysis). On basis of the beforehand motivated goals of the explicit modeling of the adaptation logic (G1), the available knowledge (G2) and the system collaborations (G3), this thesis focuses on the analysis of the modeled adaptive SoS as emphasized by the following goal:

Thesis goal G4: *One goal of this thesis is the analysis of the specified adaptation logic, system collaborations and knowledge distribution inside an adaptive System of Systems to understand the interplay and the coupling over the shared knowledge base between adaptive systems.*

Beside the analysis, simulations of the modeled adaptive SoS behavior during the development on different levels are important to verify whether the modeled system behaves as expected or violates given requirements. For example, looking at system collaborations, simulation can help understanding a single interaction protocol between systems. Later, the emergent effects of multiple protocols between several system interactions may be investigated. Another example is the simulation of one feedback loop on top of a single system that realizes a specific self-* behavior. If multiple adaptation concerns are modeled by distinct feedback loops, each concern can be separately investigated during different simulation runs. Afterwards, emergent or contradicting adaptation effects can be investigated by combining the functionality of the feedback loops to larger sets during the simulation. Even more, adaptive capabilities can be grouped and simulated within one system. Finally, more and more systems can be integrated as well as simulated towards the simulation of the complete adaptive SoS behavior. Therefore, simulation contributes to the verification of expected adaptive SoS behavior during the development, but needs a clear semantic of the underlying models for its execution.

Thesis goal G5: *One goal of this thesis is the simulation of the modeled adaptive System of Systems architecture on different levels to enable the investigation of collaborative and emerged system behavior.*

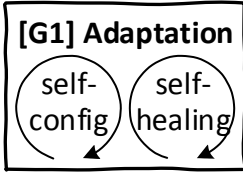
According to [181], there is lack of complete knowledge, uncertainty, and a missing overall systematic engineering approach for distributed self-adaptive software. Therefore, the tool support for the specification of the adaptive systems and its execution as well as simulation are very important [181]. Beside the analysis and simulation of the modeled system, which contributes to the verification of the behavior, the modeling language concepts for specifying an adaptive SoS should be realized in a concrete domain showing that the language concepts can be implemented and are applicable for actual systems. Furthermore, deriving an implementation of the modeled adaptive SoS enables its execution as well as simulation in the corresponding domain. Depending on the domain, modeling language concepts may be differently realized by following the dominant development paradigm of this domain. This thesis has the following goal according to the applicability of the modeling language focusing on its realization:

Thesis goal G6: *One goal of this thesis is the realization of modeling language concepts in one specific domain to investigate the emergent behavior of the adaptive System of Systems in a concrete implementation and to show that the modeling language concepts can cope with existing development approaches.*

In summary, existing approaches enable the modeling of adaptive behavior by means of feedback loops by adopting well-understood concepts from the control engineering domain and transfer those to software intensive systems [110]. In the context of this thesis, the ideas of the feedback loop modeling for adaptive systems are used to develop a modeling language that suits the specification of adaptive SoS by considering the feedback loop modeling concept as first class entity (G1). Furthermore, the available knowledge within adaptive systems is a crucial point. The MART approach proposes the promising runtime model concept [38], where system phenomena are represented, manipulated, and maintained on a higher level of abstraction during the lifetime of the system. The goal of this thesis is to integrate the runtime model concept in a modeling approach together with the adaptation logic of systems (G2). Furthermore, as outlined above, the modeling approach must also consider new arising challenges concerning partial knowledge and outdated information on the SoS level. Thus, the modeling language must have a clear semantic for the runtime models itself, its access by the adaptation logic as well as between different systems inside the SoS. Even worse, an adaptive SoS is inherent distributed with emergent behavior [39, 81, 130, 140]. Therefore, contained systems must collaborate with each other to reach SoS goals by simultaneously optimize their own behavior according to local goals. This interplay of local and global adaptive behavior as well as system collaborations must be systematically specified for the overall adaptive SoS by means of nonambiguous modeling language concepts (G3). Thereby, the coupling of the knowledge over collaboration must be considered to predict the emergent adaptation effects. Thus, this thesis focuses on a seamless approach, whose modeling language supports the explicit design of the adaptation logic (G1), the integration of runtime models (G2), and the specification of system collaborations within the adaptive SoS (G3). The modeled system behavior must be analyzable (G4) towards an understanding of adaptation effects and the prediction to the overall SoS behavior [111, 162]. Furthermore, simulation support (G5) closes the gap of verifying the behavior against given requirements of the modeled adaptive SoS by executing feedback loops and system interactions. Thus, the analysis of the modeled adaptive SoS (G4) and the simulation of the SoS behavior (G5) contributes to the verification of required system characteristics. Finally, the modeling language concepts should be realizable (G6) in a concrete domain that supports the implementation of the modeled behavior.

1.2. Contributions

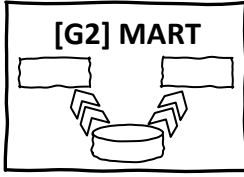
According to the beforehand motivated research challenges and goals, there are the following contributions of this thesis. An adaptive SoS consists of several diverse system types and thus, inherits different characteristics from these contained systems. One contribution of this thesis is to provide a common understanding of SoS characteristics by deriving those from the state of the art literature. Furthermore, those characteristics are used to derive requirements for a modeling language that support the specification of the interplay between different system types, such as embedded systems or CPS, by means of collaborations as well as the modeling of the complete adaptive SoS architecture.



Therefore, the main contribution of this thesis is the Deurema² modeling language. First, Deurema considers the adaptive behavior by means of feedback loops as first class entities. Second, multiple feedback loops can be modeled that contribute to an overall adaptive, layered SoS architecture. Thereby, this thesis focuses on the adaptation capabilities of the system and therefore does not tackle the engineering or modeling of the domain logic of adaptive systems. However, embedded systems and CPS need an appropriate representation of the domain logic for enabling real-time static reconfiguration and dynamic adaptation that considers nonfunctional requirements of these systems such as time. Therefore, one contribution of this thesis is the support of different, domain specific development paradigms and providing appropriate modeling concepts that enable the integration of domain aspects in the adaptive capabilities of the SoS. Additional to the feedback loops, Deurema supports two state of the art development paradigms. First, the component-based modeling is the dominant development approach for modern embedded, robotic and automotive systems. Those system types are contained in an adaptive SoS and thus, relevant for modeling the adaptation logic. Therefore, Deurema natively supports this wide range of embedded as well as cyber-physical systems and integrates the component-based approach with adaptive behavior specifications. Second, inspired by state of the art modeling techniques from the MDE, one contribution of this thesis is the first class support of rule-based behavior modeling via graph transformations. The MDE considers models as primary development artifacts [75, 90], where usually a huge number of different models are created during the development of the contained systems within the adaptive SoS. Those models describe different aspects such as the structure or behavior of the system. They are used to derive an implementation from the modeled system aspects, e. g., using code generation techniques, towards a realization of the envisioned system. Furthermore, the development models can be kept alive during the lifetime of the adaptive SoS following the MART approach [38] as outlined above. As a consequence, models play an important role during the development and lifetime of the adaptive SoS, which is a main motivation why they gain a specific attention in the context of this thesis. However, almost all models can be represented as graphs. Thus, the direct support of the powerful graph transformation concept within the Deurema modeling language enables the manipulation and integration of models into the modeling of the adaptation logic. Both paradigms, the component-based development and graph transformation, complement each other, where the former considers the development of a wide range of embedded as well as cyber-physical system types and

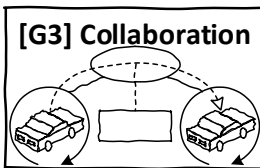
²The Distributed Eurema with Collaborations (Deurema) modeling language extends a previous modeling language called Eurema, where this modeling language name is an abbreviation for Executable Runtime Megamodels (Eurema).

the latter focuses on the general handling of models caused by the dominant MDE approach in software engineering. Thus, the integration of both paradigms into a modeling language for adaptive behavior modeling is a contribution of this thesis and supports a wide range of adaptive SoS architectures. Summarizing the contributions of this thesis with respect to the goal G1, this thesis provides the Deurema modeling language that explicitly considers the adaptive behavior of a system by means of feedback loops together with the local behavior description, which can be modeled following the component-based development paradigm, and model manipulations using a rule-based graph transformation approach from the MDE.



The starting point for describing the adaptation logic of adaptive SoS is the MAPE-K feedback loop approach from Kephart et al. [110]. This thesis considers the knowledge in this approach as first class modeling entities in form of runtime models following the MART approach [38] as emphasized by the goal G2. Because runtime models encapsulate the complete available knowledge, they must have a clear semantic that on the one hand, facilitates their integration into the

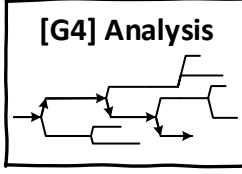
local adaptation logic and on the other hand, enables an unambiguous handling such as its access and distribution throughout the adaptive SoS via system collaborations. This thesis contributes with a runtime model categorization that refines the purpose of runtime models and gives insights into the contained information. Based on the common notion of runtime model purposes, this thesis proposes an integration concept for runtime models into the adaptation logic within the Deurema modeling language. Thereby, this thesis does not restrict the content of the runtime model information, but rather supports the encapsulation of arbitrary, domain specific information and proposes a model management concept for the uniform handling of runtime models for all systems inside the adaptive SoS. Furthermore, the model management concept facilitates partial knowledge exchange between system interactions and the modeling of visibility restrictions as motivated above. Finally, this thesis shows how runtime models can be seamlessly integrated into the beforehand mentioned component-based and rule-based graph transformation development paradigms, which leads to a consistent handling of the knowledge across potential multiple domains in the overall adaptive SoS.



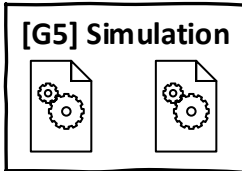
With respect to system interactions as highlighted by the goal G3, one contribution of this thesis is the systematical specification of collaborations among independent, distributed feedback loops as they are contained in the independent systems within the adaptive SoS. Within Deurema, collaborations are considered as first class entities. Furthermore, Deurema provides concepts for modeling the interplay (protocol) between systems during the collaboration. On the one

hand, Deurema fosters the separation of concerns by describing local adaptation behavior and collaboration related behavior separately. On the other hand, it provides concepts for integrating the collaboration behavior into the local feedback loops of the adaptive systems. As a consequence, the integration of the runtime model concept and the two domain specific modeling paradigms into the collaboration aspects are considered as well. Because different domains and problems necessitate several modeling guidelines or best practices, it is not in the focus of this thesis to propose a special development process or architectural design for a specific problem or domain. The presented modeling concepts suit the specification of collaborative feedback loops and its dependencies. The use of these concepts depends on the concrete application or system developer. Therefore, this thesis focuses on the Deurema modeling language approach that facilitates the specification of the adaptation logic (G1),

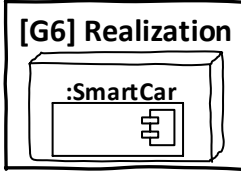
the seamless integration of runtime models (G2), and the modeling of system interactions by means of collaborations (G3) to describe the overall emergent adaptive behavior of a layered SoS architecture.



On basis of the Deurema modeling language concepts, one contribution of this thesis is the support for the analysis of the adaptive and collaborative SoS behavior considering the goal G4. First, the explicit modeling of feedback loops and their collaborations can be used to describe the local adaptation capabilities as well as the interplay of systems in the overall adaptive SoS architecture. Therefore, typical adaptation structures such as hierarchical or distributed control can be identified and their effects analyzed. Second, due to the complexity of the emergent adaptive SoS behavior, the overall adaptation effects may be hard to predict. For the verification of the adaptive SoS behavior, this thesis proposes a set of analysis rules to investigate the causality between adaptation effects, the usage as well as distribution of knowledge, and the local adaptation purpose of feedback loops. On basis of this three basic metrics, this thesis combines analysis rules, such as the causality of adaptation effects together with the access of available runtime models, to investigate more complex relationships inside the modeled SoS architecture, which contributes to a common understanding of the emerged SoS behavior. By investigating system interaction, the coupling of feedback loops becomes visible and the effects of knowledge propagation can be analyzed. At the highest abstraction layer, this thesis identifies a set of architectural patterns, such as hierarchical or layered control, as well as architectural flaws, which identify possible system threats. All presented analysis rules in this thesis can be statically (at development time) and dynamically (at runtime) applied on the adaptive SoS architecture, which is modeled with the Deurema language. Therefore, the analysis rules of this thesis contribute to a common understanding of adaptive SoS behavior (G4) and propose a general approach, which can be extended and applied for specific investigations in a corresponding domain.



Another possibility of verifying the modeled SoS architecture is simulation. In general, simulation techniques can be applied on different level of abstraction. For example, the testing of one self-* capability may involve the execution of a single feedback loop, where the verification of an interaction protocol may comprise several systems and multiple simulation runs to test different variants of the protocol. This thesis describes the Deurema execution framework, which can directly execute Deurema models. Thereby, the developer can choose, which parts of the overall adaptive SoS should be simulated ranging from a single feedback loop, over the execution of a more complex system interaction towards the execution of the complete adaptive SoS architecture. Because of a missing overall formal theory of distributed control, it is not a focus of this thesis to present formal proofs of the modeled overall collaborative system behavior. However, the simulation capabilities of the Deurema execution framework help to understand the interplay between distributed feedback loops and the developer can verify if the adaptive behavior works as expected. Furthermore, the beforehand mentioned analysis rules for Deurema models can be applied during simulation, which explicitly pinpoint to the influencing parts of the executed adaptation behavior and the knowledge propagation throughout the adaptive SoS. Additionally, the simulation of the collaborative, adaptive behavior supports timing and probabilistic effects as they are inherently needed for embedded systems and CPS.



Beside the analysis and the simulation of the modeled SoS architecture, realizing the modeling language concepts for a concrete execution platform contributes to a possible implementation of the SoS requested by the goal G6. Therefore, this thesis discusses the realization of the Deurema modeling language using a state of the art standard from the automotive domain called *Automotive Open System Architecture (AUTOSAR)*. Thereby, a mapping of Deurema concepts to the AUTOSAR framework is presented, which is prototypically implemented with the help of software tools from the automotive industry. Although the presented realization of the Deurema concepts in this thesis has the focus on the AUTOSAR standard, the general mapping can be transferred to other domains of interest or execution frameworks as well. However, the AUTOSAR framework is the current de facto standard of the automotive industry, which shows the applicability of the Deurema modeling language to a broad range of actual systems.

Finally, one contribution of this thesis is the application of the Deurema modeling language to existing approaches. Therefore, state of the art case studies and frameworks from literature are modeled respectively compared with Deurema. On the one hand, this thesis has not the focus to judge about existing adaptive architecture or frameworks nor providing metrics to evaluate existing approaches. On the other hand, the evaluation shows that the concepts of the Deurema modeling language are powerful enough covering current approaches. Thereby, typical patterns as introduced by the analysis rules occur in the evaluated case studies, which strengthened the argumentation of using the proposed analysis rules to highlight special architectural constellations or interaction patterns.

1.3. Structure

The thesis is structured as follows and the related own former work is cited along for each chapter.

Chapter 2: This chapter introduces important terms, frameworks and different system types that are necessary to understand the context of this thesis. Moreover, a running example for this thesis is introduced that is used for explaining the Deurema modeling language concepts. This chapter is partially based on definitions and terms as already introduced in the own former work in [1, 3, 11].

Chapter 3: After having a common understanding about the context of this thesis, this chapter motivates the need for a new modeling language for adaptive SoS, which at first necessitates the understanding of adaptive SoS characteristics. Furthermore, requirements for the Deurema modeling language are derived from the outlined SoS characteristics and the state of the art literature in modeling adaptive SoS is discussed. This chapter enhances the previous own work in [11, 14].

Chapter 4: With regards to the discussed modeling gaps in Chapter 3, this chapter gives an overview about the Deurema modeling language concepts. Thereby, this chapter goes through the main ideas of modeling the local adaptation logic, integrating the knowledge, and specifying system collaborations in Deurema. Furthermore, it pinpoints to the ideas of analyzing, simulating and realizing a modeled adaptive SoS architecture. Based on this chapter, the main ideas are introduced in detail in the subsequently chapters. This chapter is based on the former own work in [9, 10].

Chapter 5: The Deurema modeling language is presented in this chapter covering the goal of the direct modeling of the adaptation logic (G1). Furthermore, this chapter presents

the runtime model categorization, which enables the integration of runtime models into the adaptation logic (G2) and is partially based on former work described in [9]. Afterwards, the Deurema collaboration concept is introduced in detail, which can be used to explicitly model system interactions (G3), which is also partially described in [10]. Furthermore, this chapter considers different domains by means of integrating the component-based and rule-based paradigms into Deurema. According to the integration of different domains, this thesis benefits from the research that is not directly connected to the modeling of adaptive SoS, but rather tackles different domains such as embedded systems and CPS as described in the own former research in [2, 7, 8, 12, 13, 14]. Finally, reconfiguration and adaptation concepts of Deurema are introduced in this chapter.

Chapter 6: This chapter discusses the analysis capabilities of the Deurema modeling language, which contributes to the thesis goal of verifying the modeled, adaptive SoS architecture (G4). Therefore, several concrete analysis rules are introduced that cover dependencies between feedback loops, knowledge distribution, the causality of adaptation effects, and collaborations. An early version of such analysis rules on basis of runtime models is described in [9]. Afterwards, this chapter describes rule combinations to identify patterns and architectural design flaws within the adaptive SoS.

Chapter 7: This chapter introduces the Deurema simulation framework, which contributes to the thesis goal G5. First, a model interpreter defines the concrete semantic for each Deurema model element, which can be aggregated to simulate the adaptive SoS architecture. This thesis benefits from the former experience of model execution described in [6].

Chapter 8: This chapter comprises a mapping from the Deurema modeling language concepts to the AUTOSAR standard in the automotive domain as requested by goal G6. Furthermore, a prototypical implementation for a concrete example is discussed, where the used industry software tools are outlined. The mapping description is based on experience discussed in own former research as outlined in [2, 7, 8, 14].

Chapter 9: For the application of the Deurema modeling language, this chapter models state of the art scenarios and discusses related modeling approaches. The research scenarios are comprehensively discussed in [11], whereas one is partially modeled in the former work in [10]. However, this chapter discusses the overall applicability of the Deurema modeling language and pinpoints to found architectural patterns in the modeled examples, which aims for the goal G6.

Chapter 10: This chapter compares and shows differences of the Deurema modeling language with other modeling languages and frameworks. Additionally, it outlines the history of own former work, which leads to the Deurema modeling language. Finally, it discusses how the derived requirements from Chapter 3 are realized by Deurema and related approaches.

Chapter 11: This chapter summarizes the thesis, recaps the introduced Deurema concepts and discusses their conformance to the derived thesis goals in Chapter 1. Furthermore, this chapter outlines possible future work.

2. Preliminaries

This chapter introduces terms and definitions that are necessary and in the focus of this thesis. Thereby, it retrieves viewpoints from current literature about different system types to provide a common understanding for adaptive SoS in Section 2.1. Furthermore, important terms from the Model-Driven Engineering (MDE) domain are introduced in Section 2.2. Subsequently, the predecessor modeling language Eurema of the Deurema approach is discussed in Section 2.3 to highlight the research background of this thesis. With this background in mind, a common understanding of system collaboration is derived from literature in Section 2.4. Finally, the running example for this thesis is described in Section 2.5.

2.1. System Types

This section subsumes different system types. First, self-adaptive systems are introduced that provide flexible, dynamic behavior by providing so-called self-* properties. Second, systems that have to integrate physical elements together with software aspects are discussed. Therefore, the term Cyber-Physical System is outlined. Afterwards, the system size is increased, which leads to networks of cyber-physical systems and further to so-called Systems of Systems. This section ends with a discussion about the different introduced system types and pinpoints to the term *adaptive Systems of Systems*, which is in the focus of this thesis.

2.1.1. Self-Adaptive System

Definitions for (self-)adaptive systems vary a lot in the literature depending on the research viewpoint and the application domain. In general, there are many challenges and research questions for self-adaptive systems as discussed in [58, 127]. Furthermore, a broad discussion about different viewpoints concerning self-adaptive software is given by Salehie et al. [156]. They identify different domains, as for example autonomic computing [110], adaptive programming, software evolution, or software-intensive systems that influence the definition and viewpoint for a self-adaptive system. The common basis for all viewpoints is twofold. First, the life cycle of a self-adaptive system does not end after the development phase of the software or initial deployment on the target platform, but rather is continued during system execution [156]. Therefore, the system is able to react on changes in the environment, failures, or new requirements at runtime. Second, self-adaptive software provides so-called *self-* properties* of the system as for example *self-configuration*, *self-optimization*, *self-healing*, or *self-protection* [110]. The key requirement for the self-* capabilities is that the system is aware of its own state (self-awareness [129, 173]), requirements/goals (requirements-awareness [158]), or context (context-awareness [173]). As a consequence, this thesis refers to those *-aware property in the definition for a self-adaptive system as follows:

Definition Self-Adaptive System: *A Self-Adaptive System (SAS) is a software system that is aware of its own state, requirements, or context at runtime. Furthermore, the system uses this runtime information for adjusting its behavior or structure according to the systems' purposes.*

According to the discussed goals in Chapter 1, this thesis focuses on self-adaptive systems that control their dynamic behavior with the help of a feedback mechanism. This feedback control is widely used in embedded systems by designing the control algorithm in form of a feedback loop. In general, there are two ways specifying the adaptive capabilities of a software system, which are the *internal* and *external approach* [156]. The internal approach directly implements the adaptation logic within the domain logic of a software system, whereas the external approach splits both into separated parts that are called *adaptation engine* and *adaptable software* as shown on the left in Figure 2.1. Furthermore, from the autonomic computing domain, Kephart et al. [110] propose a reference architecture with an *autonomic manager* (adaptation engine) that is decoupled from the managed system (adaptable software). In this approach, the managed system runs the application logic, which can be influenced by the autonomic manager that executed the adaptation logic. As a consequence, the autonomic manager introduces the beforehand mentioned self-* capabilities to the system. As depicted on the right in Figure 2.1, Kephart et al. propose four dedicated adaptation activities, namely Monitor, Analyze, Plan, and Execute (MAPE) that are consecutively executed. The MAPE activities share a common knowledge base (MAPE-K) that for example consists of a set of different runtime models. The MAPE activities form a feedback loop that is usually periodically executed. First, the monitor activity retrieves information from the running system that is extracted via (software) sensors and updated in the knowledge base. Second, the analyze activity checks on basis of the updated knowledge whether an adaptation of the system is needed to fulfill given requirements and goals. The result of the analysis is also annotated in the common knowledge base. Third, depending on the analysis result, the planning activity derives proper adaptation strategies for the system that are executed afterwards. Therefore, the execute activity uses so-called effectors to force the adaptation changes back to the running system.

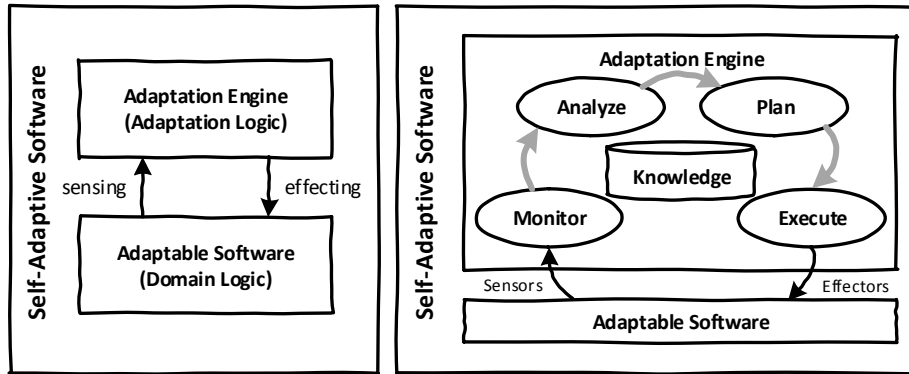


Figure 2.1: On the left: external adaptation approach [156, 175];
On the right: MAPE-K feedback loop [110]

Orthogonal to the external adaptation approach and the proposed reference architecture, adaptive behavior can be realized using different approaches. The state of the art literature distinguishes between system *reconfiguration* and *adaptation* [117, 120, 134, 156, 169, 180], whereas both techniques can be applied to change the system behavior via *parameter* or *structural* changes [134].

First, a system reconfigures its behavior by choosing predefined configurations or reconfiguring operations for changing its system behavior or structure. The key point to identify a

reconfiguration is that the developer has to specify the possible configuration space before the system adaptation happens. Applying reconfiguration techniques has the advantage that the possible configuration space is known in advance. Therefore, applying a reconfiguration suits resource restricted systems such as embedded or cyber-physical systems. Furthermore, it enables the changing of the system structure or behavior under timing constraints as necessary for hard real-time systems. Because of the predefined reconfiguration space, applying reconfiguration techniques is more statically than using adaptation approaches. The specification of available reconfiguration capabilities is possible at development time as usually applied for embedded systems using the AUTOSAR standard [14, 31, 82, 93, 170] or even at runtime [57]. Furthermore, possible reconfigurations are typically specified using variability models [57], predefined operations [169], or graph transformation techniques [82, 176], where the desired modeled configurations benefit well the given goals.

Second, system adaptation enables the changing of the systems' behavior or structure without defining all possible combinations of the configuration space in advance. Therefore, the same models can be applied for system adaptation as used for system reconfiguration. The difference is that the sequence of operations is not specified (or not known) before the adaptation happens. The system is aware of its adaptation capabilities and dynamically decides which operations are applied to reach the desired adaptation effect. Thereby, the system usually has a utility function that estimates the behavior of the system, whereas the adaptation operations can be used to optimize the behavior step by step towards the given goals. Consequently, an advantage of using system adaptation techniques is the reduced effort to specify possible adaptation steps rather than defining all combinations as done for software reconfiguration as discussed above. Thus, the system is more flexible by moving through the possible configuration space to reach its goals. A drawback of system adaptation is the raising complexity of predicting adaptation effects, because the sequence of adaptation operations is not known in advance. Therefore, the application of system adaptation raises problems in resource restricted and timing constrained systems.

However, the distinction between software reconfiguration and adaptation is not always clear in the literature and highly depends on the application domain. Broad discussions about these two terms and related techniques are given, among others, in [117, 134, 156]. For this thesis, a system that uses one or both of the approaches discussed above is considered as adaptive system.

Another dimension is the target of an adaptation that can be a parameter or the structure of the software system [134]. Because the adaptation effects are similar, and real system implementation often uses a mixture of both approaches, a clear distinction is hard to define. Therefore, this thesis considers both, parameter and structural adaptation.

As emphasized by Oreizy et al. [143] not only the adaptive system has to change its behavior to reach current needs, but also the adaptation logic itself must evolve over time to reflect changing requirements. The ability of adapting the adaptation engine is known as *meta-adaptation* [92, 99]. Examples for the successful application of meta-adaptation are given by Gui et al. [92], who introduce a framework that reuses adaptive components to change the adaptation logic, and by Piechnick et al. [149] in the context of smart environments.

Another research direction that uses variability models is Software Product Line Engineering (SPLE). In contrast to the reconfiguration of the SAS behavior, SPLE focuses on the customization of the software with respect to the underlying product [150]. Therefore, variability models in SPLE are planned during the development and describe different variants of the product, whereas the decision of a chosen configuration is explicitly done during the

development from the product management [135, 150]. Thus, in traditional SPLE the software configuration is defined during development, leads to an appropriate software deployment on the corresponding product, and cannot be reconfigured at runtime [94]. However, combining the ideas of runtime reconfiguration and software product lines leads to the *dynamic* SPLE approach, which shifts the decision of an appropriate software configuration to the runtime of the system (product). Consequently, for the dynamic SPLE approach, the same techniques for runtime adaptation as in SAS can be applied [94].

In summary, self-adaptation is important for many system types to cope with behavioral changes during system execution. It is a key feature for fulfilling high demands on flexibility, elasticity, dependability, and robustness at runtime [58]. Therefore, adaptive characteristics are needed for large systems, which are discussed below, and lead to different requirements for a modeling language that copes with collaboration in adaptive SoS. Furthermore, there are different levels of self-* properties [156] as shown in Figure 2.2 indicating a hierarchy of adaptive capabilities of the corresponding SAS. At the bottom, primitive properties enable an awareness of the system concerning its own state, requirements, or context. Major properties, at the middle layer in Figure 2.2, realize complex adaptive functionalities such as self-optimizing or self-healing on basis of the available primitive *-awareness properties. Finally, the general adaptive layer at the top copes with global adaptation strategies for the whole system. A comprehensive discussion about the taxonomy of self-* properties is given in [156].

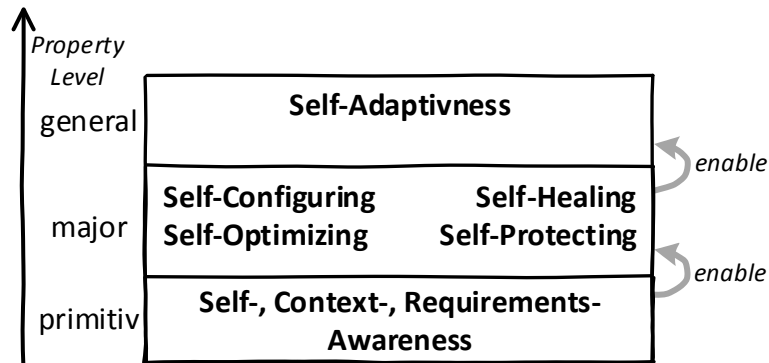


Figure 2.2: Hierarchy of self-* properties [156]

2.1.2. Cyber-Physical System

Cyber-Physical Systems (CPS) evolve from embedded systems that *"combine physical processes with computing"* [123, 126]. Whereas embedded systems are mostly closed, self-contained, and do not *"expose the computing capability to the outside"* [123], an increasing number of devices and demands of combined functionalities lead to a more and more interconnection of embedded systems to so-called Networked Embedded Systems (NES) [69]. As a consequence, such isolated control systems and afterwards interconnected embedded systems become open to their environment and build different variants of cyber-physical systems that integrate the cyber (software) and physical part [59]. Due to CPS are a federation of interconnected embedded systems, CPS inherit the sensing and interaction capabilities with the environment from the characteristics of embedded systems as discussed in [98]. Furthermore, different

research fields have to be considered that influence the design and solution space for CPS as for example distributed control and distributed computing [185].

Although there are several definitions for cyber-physical systems in the literature, e.g., from Broy et al. [45], acadtech the National Academy of Science [69], or Choi et al. [59], all definitions emphasize the tight coupling of physical processes and software computation that is characteristic for a cyber-physical system. Therefore, this thesis uses the following short, but precise definition from Lee et al. [126].

Definition *Cyber-Physical System:* *A Cyber-Physical System (CPS) is a system that integrates computation with physical processes.*

It has to be noted that *integration* implies a tight coupling between the cyber and physical parts with the effect that physical components in the system, e.g., an actuator or sensor may influence the software part and vice versa. Furthermore, because of the historical roots of CPS from the embedded domain and control engineering, CPS usually use feedback loops to cope with uncertainties in the environment or the hardware (e.g., sensor noise). Therefore, CPS often have several adaptive capabilities as discussed for SAS in Section 2.1.1. A comprehensive discussion about CPS and related system types is given by Lee [126] and Kim et al. [112].

2.1.3. Networked Cyber-Physical System

The openness of CPS lead to a varying number of subsystems that interactively join and leave the overall system. As a consequence, clear system borders cannot be defined. This causes further challenges concerning the availability of a certain functionality, the reliability of the system, fault tolerance, security and safety issues. In the following, this thesis emphasizes these additional challenges of an open system that is composed of multiple networked subsystems by the more specific term Networked Cyber-Physical System (NCPS). Definitions from the literature, such as Kim et al. [113] or Choi et al. [59], emphasize the distributed nature of a NCPS and the resulting need for coordination schemes between independent subsystems. This thesis combines both viewpoints from [113] and [59] to the following definition, which is used for the rest of this thesis.

Definition *Networked Cyber-Physical System:* *A Networked Cyber-Physical System (NCPS) consists of distributed components (subsystems) that have diverse capabilities. A NCPS requires a coordination scheme for its distributed subsystems that must balance between autonomy and cooperation.*

Furthermore, Kim et al. [113] state that the autonomous subsystems must be considered as unreliable and loosely synchronized. This thesis follows this argumentation due to the openness characteristic of the NCPS, where subsystems may arbitrary leave or join the NCPS over time. Additionally, NCPS must deal with the same problems of the tight coupling between physical components and software parts as described for CPS in Section 2.1.2. Moreover, often each autonomous subsystem within a NCPS must adapt its behavior according to its peer subsystems, while considering the interplay between its own behavior, the other subsystems' behavior, and the overall system-level behavior as defined by the collaborations. Ruling such NCPS is challenging due to the complexity, dynamics, and emergence and it is often not feasible to develop once and forever an autonomous subsystem that then lives within a NCPS in the long term without any need to adapt to changes. Therefore, it is highly attractive that NCPS are self-adaptive at the level of the individual autonomous subsystems and at the overall system-level to cope with the emergent behavior, to adapt and absorb open, dynamic, and deviating NCPS architectures, and to adapt to open and dynamic contexts. However, due to the composition of subsystems that autonomously adapt to their individual contexts into a

NCPS, also unwanted co-adaptation effects may emerge from interference of the individual feedback loops placed in the autonomous subsystems [133].

2.1.4. System of Systems

Following the trend of integrating more and more independent subsystems will reach the point, where system borders become unclear. Isolated system solutions become integrated into federations of distributed systems. This thesis refers to such kind of systems by using the term System of Systems (SoS) [172]. Similar observations concerning the emergence of such systems and the importance of their collaborating subsystems have been made for many related research areas as for example Ultra-Large-Scale Systems (ULSS) [142], and for particular technological domains as software-intensive systems [184], next generation of embedded real-time systems [42, 141, 159, 165], vehicular systems [21], and service-based systems [68].

This thesis defines the term SoS following the ideas in the research agenda from [172] and from Gezgin et al. [81] as follows.

Definition *System of Systems:* *A System of Systems (SoS) consist of other (software) systems, where each system is developed, operated, evolved, and governed independently from the other systems. Systems in a SoS are networked to achieve common goals.*

According to the definition, the overall behavior for the SoS emerges from the capabilities of each contained system solution and the interaction between contained systems. On the one hand, each system cannot achieve the overall SoS goals by its own, which is the initial trigger to cooperate with each other and emerge to a bigger system. On the other hand, systems still follow local optimization strategies according to their local subgoals. Furthermore, a clear distinction between NCPS and SoS cannot be given. Depending on different domains the terms NCPS, SoS, and ULSS are used synonymously in the literature. However, a key characteristic of a SoS is the independent management and the high heterogeneity of contained systems in contrast to CPS and NCPS, where system parts become integrated into one system solution. Consequently, there is no static integration of systems in a SoS and the available pool of functionality emerges from the independent capabilities of the contained systems.

Krygiel [121] distinguishes a special subset of SoS that are characterized by a completely decentralized control scheme, named Federation of Systems (FoS). There, a dedicated central control entity is missing and the systems in the overall FoS must cooperate and collaborate with each other to reach global goals [140].

2.1.5. Internet of Things

The Internet of Things (IoT) paradigm is the enabling technology to link physical devices and systems [69]. More and more devices have access to the Internet and thus become interconnected. As stated by the European Research Cluster in the Internet of Things "*IoT refers to objects ("things") and the virtual representations of these objects on the Internet.*" [104]. Furthermore, "*the variability and heterogeneity of devices to be considered in this domain challenges the possibilities of interoperate the devices*" [74]. Therefore, on the one hand, the IoT enables the emergence of devices, services, and systems to the above described SoS [130]. On the other hand, the huge number of different devices introduces a high heterogeneity of available functions and services as well as different technologies, network protocols, and modeling techniques. A survey about application domains and a broad discussion about the evolution of the IoT are given in [20, 104].

2.1.6. Adaptive Systems of Systems

Depending on the research domain, different viewpoints on the system types exist in the literature. This thesis does not claim unifying those different viewpoints, but rather gives an overview about different kinds of systems by citing the state of the art literature for each system type. Therefore, characteristics that appear in CPS can also be found in NCPS as well as in SoS. Because of the different terminology between NCPS, ULSS, and SoS, this thesis refers to the term SoS as synonym for all those system types. For this thesis, it is important that a SoS integrates different CPS and therefore has to deal with the complete spectrum of large software intensive systems as well as the tight coupling to physical elements of the real world.

Furthermore, with respect to the goals, this thesis focuses on the collaboration aspect of system interactions within the overall SoS. These system collaborations are the cause of the emergent SoS behavior. Additionally, systems within the SoS realize individual adaptive capabilities that optimizes its own local behavior as motivated above. Due to the joint system interactions, such local adaptive capabilities leave the border of one system, emerges into the overall SoS behavior, and evolves during the lifetime of the SoS coping with changing requirements. Thus, there are two main aspects of this thesis focusing on SoS. On the one hand, it considers the individual, adaptive local behavior of systems as first class entity, whereas these systems are contained in the SoS. On the other hand, this thesis considers collaborations as first class entities linking independent systems over joint interactions, which contributes to the emergent SoS behavior. To emphasize both aspects, the term adaptive System of Systems is used for the rest of this thesis, which is defined as follows.

Definition *Adaptive System of Systems:* *An adaptive System of Systems is a SoS, where the individual adaptive capabilities of contained (software) systems leave the local system borders and emerge via system collaborations into adaptive SoS capabilities. The adaptive SoS has to balance between global and local goal optimization of available systems by dynamically arrange, establish, and remove collaborations between those.*

Inspired by [69], Figure 2.3 shows the evolution of different system types with respect to their size and connectivity of contained subsystems, which summarizes the system types discussed in this section and finally, leads to the adaptive SoS system type as defined above. At first, embedded systems already introduce important characteristics, such as the use of control loops, to guarantee a stable set on functionality even in uncertain environments. Furthermore, they are coupled with the physical context via sensors and actuators, which enables a direct influence of the system surroundings. Concerning the size, embedded systems are considered as self-contained, closed systems realizing a dedicated piece of functionality such as the triggering of an airbag in a car. The next bigger system type in Figure 2.3 describes networked embedded systems, which are essentially embedded systems that become interconnected via buses (e. g., in a car) or the Internet (e. g., mobile phones). Cyber-Physical Systems are the next step on the evolution scale, where the software (cyber) part becomes much more powerful. Furthermore, a CPS often runs multiple feedback loops and controls several subsystems. Thus, the diversity of contained systems challenges the design and development of CPS. Examples for CPS are modern cars that contain more than one hundred interconnected, independent control units with million lines of code [46]. Finally, SoS (and all other synonymously used system types) are a federation of several independent systems. Thereby, the openness of SoS and the need of collaboration among the containing systems are important characteristics. The IoT can be seen as enabling technology [104] that establishes links between all kind of system types and devices supporting the interconnection and emergence of such systems.

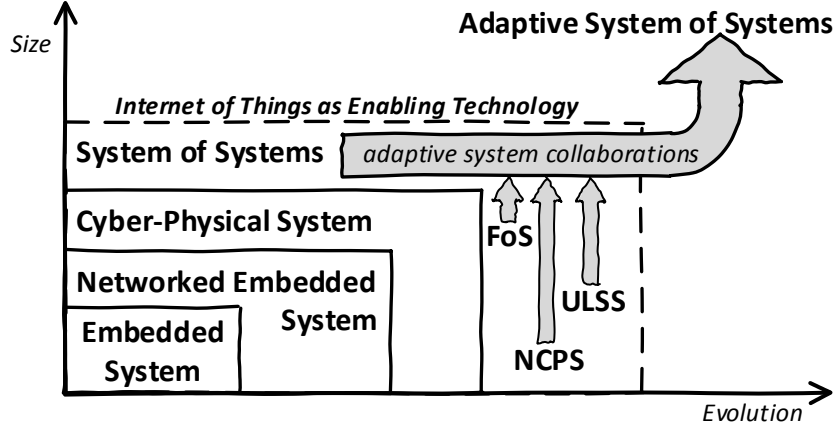


Figure 2.3: System type evolution

Considering the adaptive behavior in combination with the collaboration among systems within the SoS lead to the adaptive SoS as biggest and heterogeneous system shown at the top in Figure 2.3.

2.2. Model-Driven Engineering

While following the definitions and discussion of different system types in the former Section 2.1, an increasing complexity of nowadays systems can be observed. One approach for tackling the complexity of large software systems is the usage of Model-Driven Engineering (MDE) techniques. The goals of the MDE are twofold. First, the MDE abstracts from “complexities of the underlying implementation platform” [75] by handling models as first-class entities during the software development. Second, as stated by France et al. MDE “hide[s] the complexities of runtime phenomena” [75] by using runtime models that describe the context of the software system or the system itself during execution. Furthermore, runtime models can be used to describe system evolution or adaptation of the running system. In former own research, different MDE techniques such as model transformation [2, 3, 7, 14] or model verification techniques [6, 8] are applied. Additionally, with respect to the goals of this thesis as motivated in Chapter 1, the Deurema modeling language is developed that uses different MDE techniques to enable system analysis and simulation capabilities. In the following, with this motivation in mind, important terms that are related to MDE are clarified.

First, the term model-driven engineering itself is discussed following the definition from France et al. [75].

Definition Model-Driven Engineering: *In Model-Driven Engineering (MDE), models are the primary development artifacts. MDE tackles complexity by describing complex software systems via models at multiple levels of abstraction and from different viewpoints. Furthermore, MDE supports techniques for model simulation, verification, and transformation.*

Beside France et al. also other researches, such as Schmidt [161], discuss that MDE is a set of different techniques, technologies, and frameworks. The common view is that models are the primary development artifact. Even source code is seen as a model, representing the system behavior as stated in [75].

Related to the MDE, the Model-Driven Architecture (MDA), as defined by the Object Management Group (OMG), takes the idea of using models as first-class entities and extends it by providing standards as well as a whole development process. The MDA includes standards for model representation, exchange, modeling languages, transformation, and execution of models [90]. Furthermore, the OMG defines several layers within a conceptual framework with different model types. For example, a Platform Independent Model (PIM) is used to describe high level aspects of the system related to the current problem domain. Additionally, according to the MDA approach, a Platform Specific Model (PSM), which contains more specific information about the used realization technology or framework, should be derived from the PIM using model transformation techniques [44, 90]. Appropriate modeling languages, e.g., the Unified Modeling Language (UML) [87] or domain specific languages, are needed for all these model types on different abstraction layers. The transformation between model types and the definition of modeling languages is enabled by metamodels as further discussed in the next sections. Comprehensive discussions about MDA principles can be found in [44, 90, 115]. In the following, the key concepts from the MDE, which are used in the context of this thesis, are subsequently introduced. At first, the model term is defined, because it is the basic and primary concept within the MDE. Afterwards, metamodels are introduced that can be used to define the concepts of a model or a modeling language. Subsequently, the special research direction of using runtime models is discussed, which is the basic concept of knowledge representation within this thesis. Finally, two MDE techniques are introduced, whereas the first tackles the management of models during development and runtime, also known as megamodel approach, and the second technique considers graph transformation for the manipulation of models.

2.2.1. Model

According to the definition of the MDE, models play a key role for system and software development. There are varying definitions of a model depending on its purpose of usage. This thesis follows the definition from own former work in [1] that describe three main characteristics of a model.

Definition Model: *A model is characterized by the following three elements: an (factual or envisioned) original the model refers to, a purpose that defines what the model should be used for, and an abstraction function that maps only purposeful and relevant characteristics of the original to the model.*

This definition goes hand in hand with the model definition in the context of the MDA given by the OMG in [90, p. 5] and enriches the model definition of [98], which focuses on the abstraction only. Due to the definition, a model is always an incomplete description of the original or running system. Because of the special purpose of a model, different model types are used that fit best to the corresponding representation of the original. As for example, the UML includes different model types for describing the structure of a system, e.g., by class diagrams or component diagrams, the behavior, e.g., by automata or activity diagrams, as well as the interaction between system parts, e.g., by sequence diagrams. Additionally, different viewpoints on the same part of the system are typically modeled in distinct models that have different model types. Consequently, in a typical setting, the system developer has to deal with a large set of models using MDE techniques [97].

The example in Figure 2.4 shows on the left a physical car, where the software component structure of this car is represented as model on the right. The model depicts two software components `Driving` and `WheelController` that are connected via an unnamed interface. The

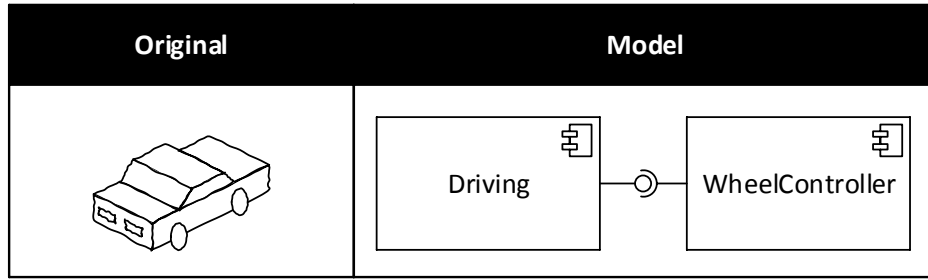


Figure 2.4: Original and its model

model in the example fulfills all three requirements of the definition above. First, it refers to the physical car on the left.¹ Second, the purpose is the (partial) description of the software architecture of the car in form of a component diagram. Third, the model omits unnecessary information as for example the size and the color of the original car.

2.2.2. Metamodel

As emphasized above, metamodels play an important role enabling MDE techniques such as model transformation, simulation and verification. For the definition of a metamodel, the thesis follows the argumentation in [36] and the definition of the OMG in [90].

Definition *Metamodel*: *A metamodel defines the modeling language concepts of a model.*

The word "*defines*" in the definition means that a "*metamodel is a formal specification*" [36] of a model. Therefore, a metamodel provides concepts (defines the abstract syntax) to create models that conform to the metamodel. The relation between model and metamodel is similar to the relation of a program and a programming language. The programming language defines concepts, usually by a grammar, which define the building blocks that can be used to create conform programs according to the programming language. A comprehensive discussion about models and metamodels is given by Bézevin et al. [36].

As an example, on the left in Figure 2.5 an excerpt of a metamodel is depicted that defines the two concepts *Component* and *Port* by an UML class diagram. A component can contain an arbitrary number of ports indicated by the *ports* containment reference and the corresponding multiplicity $0..*$. Furthermore, components can request other ports, which is modeled by the *requests* reference. The model on the upper right in Figure 2.5 shows a concrete instance situation in abstract syntax, which uses the metamodel concepts defined on the left. The model contains the same two components from Figure 2.4 describing the component architecture of a car. Therefore, the model in abstract syntax comprises two *Component* instances and one *Port* instance, which are connected by the corresponding references. The model on the lower right in Figure 2.5 shows the same instance situation, depicted in concrete syntax using the UML component diagram notation, as the model in abstract syntax above. Thus, both models on the right conform to the metamodel or, with other words, the metamodel defines the available concepts of the models.

¹Of course, the picture on the left in Figure 2.4 is not the original car, but rather a picture of a car and thus, a model according to the definition as well.

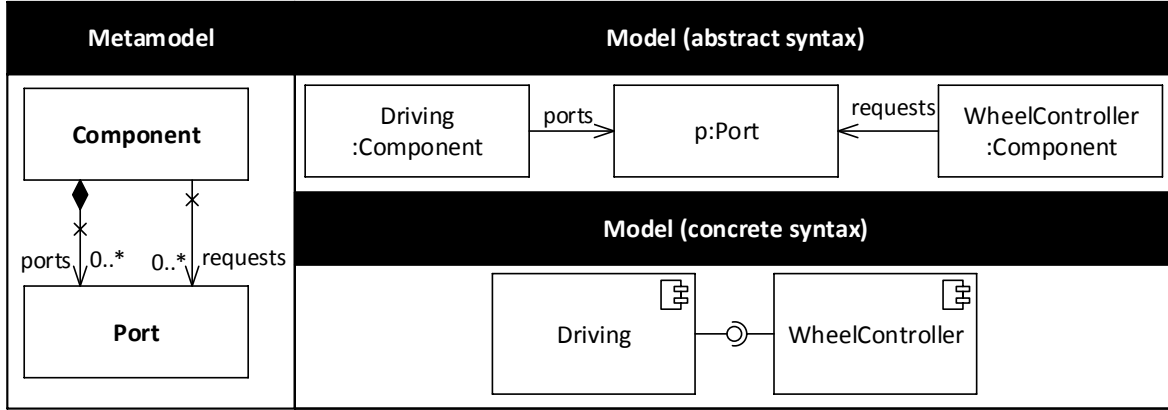


Figure 2.5: Metamodel with corresponding model

2.2.3. Runtime Model

The MDE research approach Models@runtime (MART) proposes that models, which are created during the development of a system, can be kept alive during system execution as runtime models [38]. Therefore, development models can be reused and are still valid after the deployment of the system. In contrast to development-time models, runtime models provide *“views of some aspect of an executing system and are thus abstractions of runtime phenomena”* [75]. The idea behind runtime models is to benefit from available MDE techniques and experience in the same way as it is done during software development. Furthermore, usually the same models describing parts of the system and created during the development phase can be used during system execution. This enables abstract system representations at runtime, which may support dynamic adaptation of the system [75].

Currently, there are different opinions in literature about what exactly a runtime model is and what not. In the following, a definition for runtime models is given on basis of the former own work in [1, 9] that is used for the rest of this thesis. Afterwards, differences between other opinions from literature and related research approaches are discussed.

Definition Runtime Model: (1) A runtime model is a model that describes information of interest of the system or system context. (2) The runtime model is maintained by the system and accessible during the system execution. (3) The runtime model has at least an indirect causal connection to the system.

Different properties can be subsumed for a runtime model according to this definition. From point (1) and because a runtime model is a model as defined in [128] and discussed in the former Section 2.2.1, a runtime model is always a partial description concerning the information of interest. Furthermore, runtime models capture information in the boundaries of the system and system context, which is contrary to the definition in [75] that focuses on the system only. The adaptable software together with the adaptation engine realize the overall adaptable software system (cf. adaptive system discussion in Section 2.1.1). Runtime models capture information from both parts that are referred to the general term system for the rest of the thesis. As a consequence, runtime models are conceptually not limited to be maintained in the adaptation engine. According to the point (2) in the definition, the runtime model is syntactically defined, e. g., by a metamodel. This enables the maintenance and handling of the model by the system at runtime. Moreover, runtime models are neither limited to a modeling language, e. g., UML, the Business Process Model and Notation (BPMN), or the

Knowledge Acquisition in Automated Specification (KAOS) language nor to a specific model type, e.g., state machines, class diagrams, or goal models as long as it can be accessed at runtime (2). Finally, the point (3) of the runtime model definition implies that the running system is interconnected with the runtime model, which is named *causal connection*. As described in [129], a causal connected runtime model is linked with the running system in the sense that changes in the runtime model will cause changes in the corresponding represented system entity and vice versa. This thesis extends the strict causal connection definition of [32, 38] and own former work in [1] in two ways. First, there is no strict direct causal connection to the system required. As a consequence, indirect causal connection dependencies are also considered. Indirect causal connections appear due to system collaborations within the adaptive SoS. For example, if one system shares a runtime model with another system via a joint interaction, the latter system can influence the former system by manipulating the shared runtime model representation. Due to the causal connection, changes in the runtime model will affect the represented system entities of the runtime model, which is discussed later by introducing the runtime model handling of the Deurema modeling language. Second, there can be an (indirect) causal connection to the adaptable software system or to other systems and their corresponding runtime models in the SoS. Runtime models that refer to other systems exist for example in hierarchical self-adaptive systems, which employ meta-adaptation.

A runtime model can have different degrees of complexity. Therefore, a boolean value can be a runtime model, if for example the system is aware of the fact that the value cannot only be true or false but rather represents the status of an enabled (disabled) router in a network. Manipulating the boolean value leads to the effect that the router is switched on/off. This additional semantic information potentially enables the system performing more intelligent, as it is done for example in adaptive systems.

In general, runtime models are able to represent system information during system execution [38]. In the context of this thesis, runtime models are additionally important for the data exchange and knowledge representation within the collaboration of a system. Furthermore, such runtime models can be dynamically manipulated on a much higher level of abstraction to achieve interoperability, e.g., by automatically transforming different formats, force runtime model manipulations to the running system, or enabling runtime analysis and verification.

Finally, it has to be noticed that executable models are a related research direction to runtime models. Both approaches emphasize the runtime representation of key aspects from the running system. Runtime models require an additional causal connection as defined and discussed above. However, a clear distinctive feature between both approaches cannot always be given depending on the research community and concrete application domain. For example, Xiong et al. [107] propose an approach named *knowledge-based executable models* for the quantitative evaluation of SoS architectures and claims that it overcomes the limitations of conventional architectural evaluation methods. The executable models can be used for simulating the interaction and connections between components in the overall system in order to assess the components ability to meet the specified capability requirements. However, in this approach, the executable models of the SoS architecture neither represent the running system nor have a causal connection. Also Kilicay et al. [111] argue for a shift towards executable models and the need for simulation tools. They suggest executing the modeled SoS in order to analyze system state and emergent behavior by the use of *Artificial Life* tools. The *Artificial Life* framework can generate executable SoS models and has the ability to analyze/simulate the influence of architectural changes on the overall system behavior. Both, Xiong et al. [107]

and Kilicay et al. [111] do not use the executable models as system representative during runtime but rather for simulation of different system configurations.

2.2.4. Model Management

Due to models are the main artifacts in the MDE, new challenges concerning the model management arise [97]. In the MDE, models are derived from existing models via model transformation. Several views may exist describing (overlapping) parts of a model or different models are created in the different stages during software development, whereas models in later stages are created on basis of models in former stages. As a consequence, there are a lot of dependencies between models that are important during software development as well as system execution, e. g., considering runtime models. The explicit capturing and managing of such dependencies is done in megamodels. A survey about megamodel approaches and a comprehensive discussion is given by Hebig et al. [97]. They give an overview about definitions from the two founders of the term megamodel, which are Bézevin [22, 37]) and Favre [72], and provide a unified definition of the term together with a core metamodel for megamodels. This thesis takes the definition from Hebig et al. [97], because the authors already subsume different viewpoints and the state of the art literature.

Definition Megamodel: *A megamodel is a model that contains models and relations between them.*

First of all, this thesis uses megamodels as model management technique for runtime models and collaborations between systems. Thus, megamodels can be used to maintain relationships between runtime models such as different views or track influencing requirements to the corresponding system reflection model. Due to the distributed nature of adaptive SoS, different runtime model versions may be located in different parts of the SoS that can be described in megamodels, too. Additionally, megamodels can be used to determine dependencies between runtime models of collaborating systems. Concerning the collaboration of systems, megamodels provide techniques describing impact relations due to runtime model manipulations within the adaptation process of systems. Those impact relations may emerge over system borders due to the interaction of systems.

Figure 2.6 conceptually depicts a megamodel that contains different models and the relationships between those models. First, the megamodel contains the component model from the examples above and two additional model views. Each view is derived from the component model that shows only one component, which is further refined by additional internal behavior. Second, the megamodel contains the relationships between the three models that are the two derive relations. Third, the megamodel has a notion of the relationships, which allows reasoning about dependencies between models. If for example both derive relations are operationalized, e. g., due to a graph transformation, and the source component model changes during the development, the megamodel can automatically update the two views via the known relationships. Thus, a megamodel can be seen as model storage that facilitates model management techniques such as maintenance, versioning of models, or model access control [97, 178].

More recently, the research approach of runtime megamodels combines the idea of runtime models with the megamodel approach. Thus, the megamodel is kept alive after the deployment of the corresponding software system. Therefore, the megamodel maintains all contained models and their relationships during system execution, which is similar to an online model storage. Furthermore, it can track changes as well as the access to the underlying knowledge base, which is characterized by the model manipulations of the runtime models during system

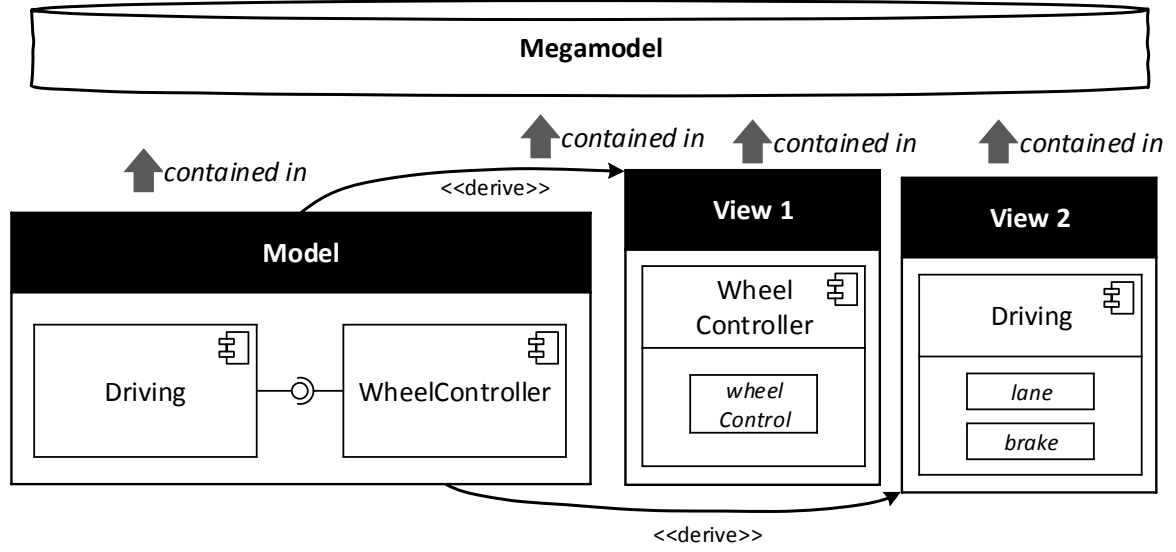


Figure 2.6: Megamodel containing models and relationships

execution. Combining the runtime model and megamodel approach leads to the following definition, which is used for the rest of this thesis.

Definition *Runtime Megamodel:* *A runtime megamodel is a megamodel that is kept alive at runtime after the software system deployment.*

As a consequence of this definition, an adaptive system can use a runtime megamodel to benefit from the explicit maintenance of models and their relationships beyond the initial deployment of the software. Thus, the knowledge base is available at runtime, which enables the adaptation logic to reason about it. Furthermore, if the megamodel contains runtime models, each model manipulation by the adaptation logic will cause a corresponding change in the underlying system due to the causal connection of the runtime model to the software system. Additional, all changes can be tracked and explicitly maintained by the runtime megamodel. This enables the reasoning about the evolving changes as well as their optimization as often done in meta-adaptation approaches. The Eurema approach [175], which is the predecessor modeling language of Deurema, proposes the runtime megamodel concept in the context of self-adaptive systems and uses it to explicitly maintain the specified adaptation logic at runtime as discussed below.

With respect to adaptive SoS, a runtime megamodel can be used to maintain the distribution of knowledge, which is caused by system interactions. Furthermore, the runtime megamodel is able to taking care about visibility restrictions to the underlying knowledge base, partial or versioned views, and can explicitly capture dependencies between collaborating systems by maintaining the relationships of shared runtime models.

2.2.5. Model Manipulation

Beside the model management, the MDE proposes different approaches for the manipulation of models. For example, following the MDA development process motivated above, PIM are created first that describe the design as possible solution of the software system under development. Afterwards, PSM are derived from the PIM that enrich the design models by tacking additional information about the used realization technology into account. This derivation

can be manually applied by the software developer or by means of model transformation techniques provided by the MDE. The latter has the advantage, that the PIM can be derived automatically once a model transformation is defined. However, such a model manipulation by deriving a target model (the PSM in the example above) out of a source model (the PIM) needs a formal basis to automate the model transformation. Due to the fact that almost all models in the MDE can be represented as graph structures, graph transformation is a powerful approach for the formal definition of such model manipulations. Thereby, graph transformation is a technique in the MDE to create a target graph, which is for example the target PSM in the example above, from a given base graph, which can be for example the PIM. Such a graph transformation can enrich the target graph by adding information in form of nodes and edges, which can be interpreted as adding information in the model. Furthermore, it can change the graph structure, e.g., by reconnecting nodes, which can be for example a redesign of the model. Finally, the graph transformation may remove nodes or edges from the graph, which corresponds to a deletion of information of the corresponding model.

As outlined in [154], there are semantically different definitions for graph transformation, whereas the following is important in the context of this thesis. A graph transformation system consists of a set of graph transformation rules, where each rule is defined by a left-hand-side (LHS) and right-hand-side (RHS). The LHS of a rule specifies a graph pattern, which is a subgraph that must be found as isomorphic image in the base graph (called *match*) as shown in the schematic sketch at the top in Figure 2.7. If a match is found, the RHS describes the application effect of the graph transformation rule. Transforming the LHS into the RHS of a rule is called *production* [154]. All nodes and edges that are part of the LHS and not in the RHS are deleted. Furthermore, all elements that appear only in the RHS are created. Applying the RHS on the found match in the base graph leads to a new modified target graph, which is called *derivation* or *transformation* step. In general, there are two semantically different algebraic approaches for graph transformation rules. The *double-pushout* approach (DPO) forbids the deletion of nodes, if it has edges in the base graph that are not part of the match (dangling-edge condition). In the semantic sketch in Figure 2.7, the node 2 could not be deleted because the edge *e* is not part of the match but exists in the base graph. The *single-pushout* approach (SPO) neglects the dangling-edge condition and additionally removes all edges from and to the deleted node in the base graph. In the context of this thesis, the SPO approach is used for all occurrences of graph transformation rules. A comprehensive discussion about graph transformation approaches can be found in [154]. Own former work in [6, 12, 13] uses graph transformation for the definition of the execution semantic of models, which allows the direct execution of the modeled system by means of a model interpreter. Furthermore, graph transformation facilitates the mapping of models from one modeling language to models in another modeling language as shown in own former work for an industrial case study in the automotive domain in [2].

At the bottom, Figure 2.7 shows a concrete example graph transformation rule separated into RHS and LHS. The LHS contains two objects (nodes) that are a **SmartCar** named **bmw** and an **Electronic Control Unit (ECU)** named **controller**. Both objects are linked via a reference (edge in the graph). For the sake of simplicity, the type of the edge can be unambiguously identified and thus, is omitted in the example in Figure 2.7. However, all graph transformation rules used in this thesis are applied on typed graphs, which implies typed (and optional named) references in the rules as well. The RHS of the graph transformation rule in the example depicts the same **SmartCar** object (denoted by the same name and type) and a new **Failure**

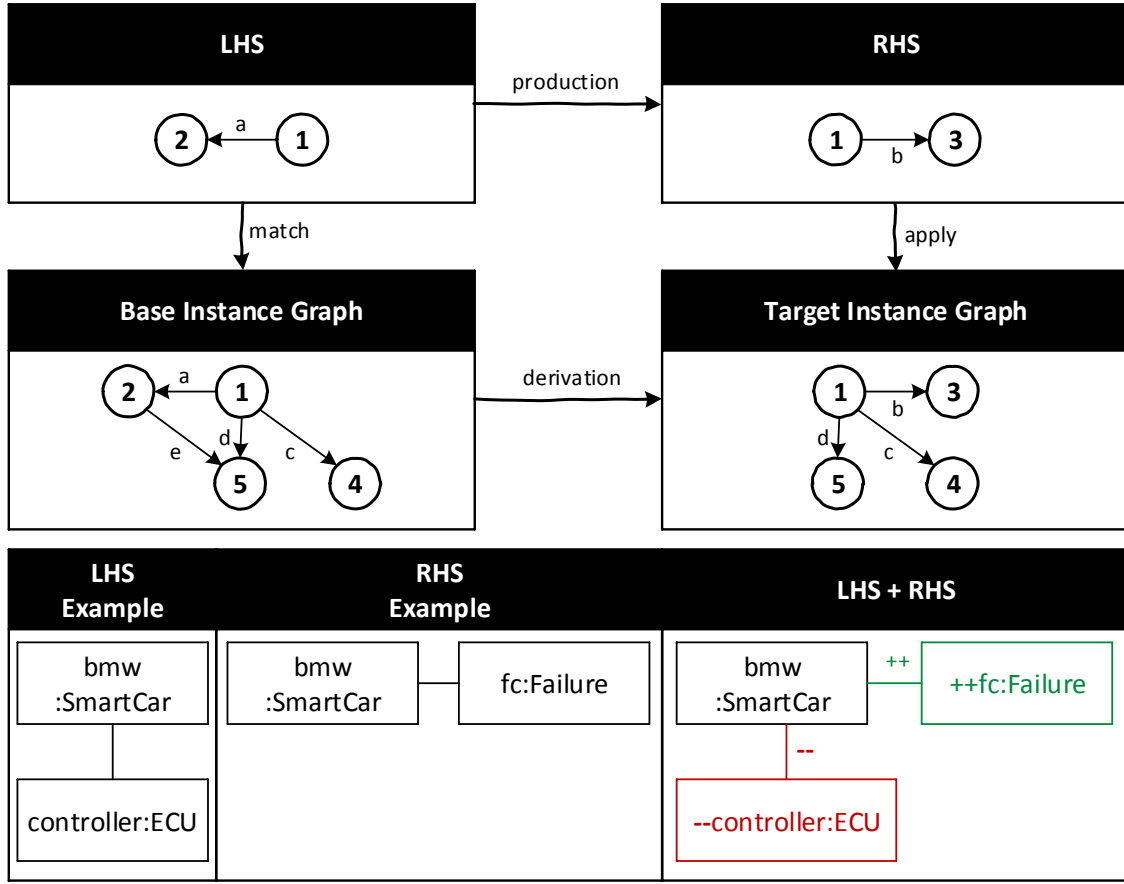


Figure 2.7: Graph transformation rule

object. Consequently, if a match for the LHS is found in the base graph, the application of the RHS will delete the found ECU object and create the new Failure object. Additionally to the distinct views of the RHS and LHS of the example graph transformation rule, a combined view including the LHS and RHS is depicted on the right in Figure 2.7 and used as notation for the rest of this thesis. Thereby, the deletion of an object or reference is indicated by a red colored border and an additional "--" sign. Furthermore, the creation of an object or reference is denoted by a green colored border and an additional "++" sign. Thus, the combined view models exact the same information as the two distinct views on the left in Figure 2.7. A comprehensive discussion about terms, definitions, and applications of graph transformation can be found in [3].

In the context of this thesis, all model manipulations are based on graph transformation rules with the advantage of a clear semantic of the underlying formalism. Moreover, this thesis manifoldly applies the graph transformation technique for different use cases. At first, the declarative pattern specification of graph transformation rules and the matching of this pattern by searching the LHS of the rule in the base graph corresponds to a model query, which is applied on the underlying knowledge base, e. g., the runtime models contained in the megamodel. Thereby, the match retrieves the specified part of the model as defined by the rule pattern. This, can be used for static model analysis, where the rule pattern define those structures that are searched in the knowledge base afterwards. Even more, due to the runtime megamodel approach, the analysis rules can be kept alive in the megamodel as well and used

for runtime analysis on all models that are contained in the megamodel. A model analysis using graph transformation rules focus on such rules, where the LHS equals the RHS. As a consequence, the LHS is searched in the model and corresponding matches are retrieved, but there are no side effects on the model base by means of graph manipulations. For an analysis, the retrieved matches must be evaluated by the software developer, which might improve the system design during the development. Alternatively, the retrieved matches can be analyzed by the adaptive system itself during its execution to reason about the current system state and trigger necessary adaptation steps.

In contrast, if the LHS and RHS of the graph transformation are not equal, applying the rule might cause a model manipulation by means of creating, reconnecting, or deleting nodes as well as edges in the underlying graph representation of the model. This thesis uses such model manipulations for two further cases. The first case uses the rules for model transformation. Thereby, model views are derived from models contained in the megamodel, which is necessary for system collaborations inside the adaptive SoS. For example, if two systems share information by exchanging models, it is reasonable that only an important subset of the complete model is shared. One possibility of specifying this subset is the definition of the desired amount of information by means of a rule pattern. The megamodel can use the formal specification of this rule pattern and can automatically derive the desired view from the knowledge base, which is shared in the system collaboration afterwards. As second use case, graph transformation rules facilitate the simulation of the adaptive SoS behavior. For example, if multiple rules define the access as well as the manipulation effect of an underlying model, a successive execution of these rules belongs to an evolving model structure over time. This effect is used to implement a model simulation based on formally defined graph transformation rules.

2.3. Eurema Modeling Language

The starting point for the research described in this thesis is the Executable Runtime Megamodels (Eurema) modeling language. Eurema is a first step towards the explicit specification of the adaptation logic following the external MAPE-K approach from Kephart et al. [110]. Therefore, the modeling language focuses on feedback loops that consist of adaptation activities and their use of knowledge in form of runtime models. Furthermore, Eurema models are kept alive during the execution of the SAS, whereas the megamodel approach as model management technique is combined with the runtime model approach. As a consequence, Eurema is a modeling language that considers runtime megamodels to reason about the adaptation logic during the lifetime of the SAS. Thereby, Eurema focuses on the systematical modeling of the adaptation engine of a single SAS, which decouples the adaptation logic from the domain logic following the external adaptation approach as proposed in [156]. Eurema introduces two diagram types that are Feedback Loop Diagrams (FLD) for the modeling of the adaptation behavior and Layer Diagrams (LD) to capture the concrete instance situation of an adaptation engine. In the following, both diagram types are introduced because their concepts are refined in the presented modeling language of this thesis.

Within a FLD, the Eurema modeling language distinguishes between adaptation activities and their use of runtime models. An activity is modeled as hexagon block arrow that is labeled with its name and stereotyped by its purpose of the activity as depicted in Figure 2.8. The purpose of the activity is related to the adaptation step proposed by the MAPE-K reference architecture [110] that are *Monitor*, *Analyze*, *Plan*, and *Execute*. Activities are one basic concept and treated as black boxes in Eurema. Consequently, the developer is responsible

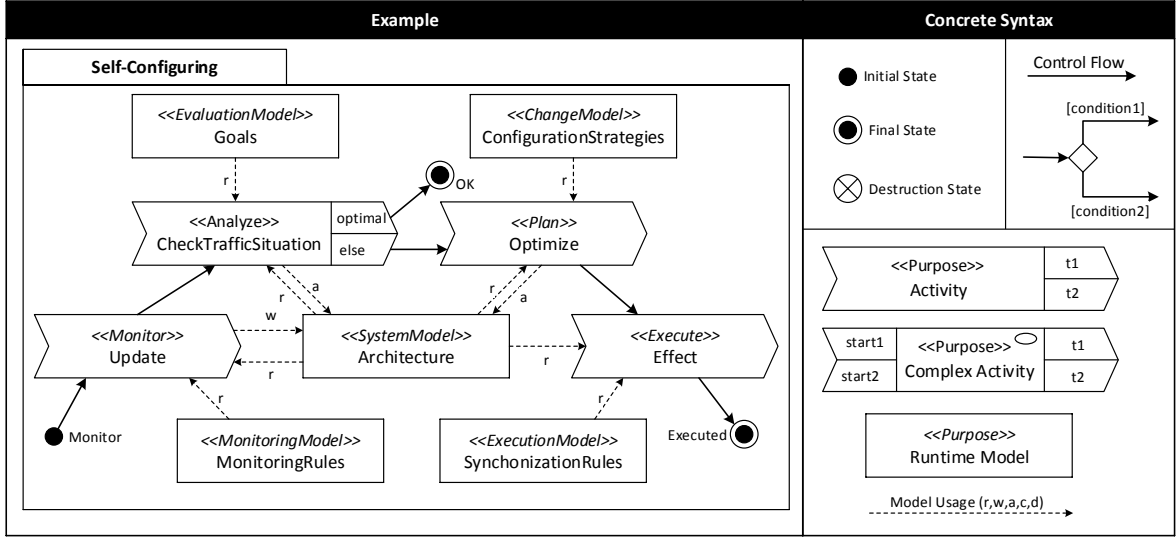


Figure 2.8: Eurema Feedback Loop Diagram (FLD) for Self-Configuring

for the implementation of activities that cannot be modeled within the Eurema language. However, the execution of an activity may result in different return states, which can be modeled by outgoing exit compartments of the activity. Beside normal activities, Eurema supports the modeling of complex activities, which are additionally labeled with an ellipse placed right next to the purpose (cf. concrete syntax on the right in Figure 2.8). Complex activities encapsulate another FLD by providing their signature. The entry compartments correspond to initial nodes of the encapsulated FLD, whereas the exit compartments refer to the final nodes. During execution of a complex activity, the corresponding FLD is unfolded and invoked. Finally, the control flow between activities can be described, which includes the possible specification of conditional branches via decision nodes.

A runtime model is modeled as rectangle that is labeled with its name and stereotyped by its purpose, whereas the latter was introduced by Vogel et al. [175, 178]. Runtime models represent the available data of the adaptation engine and can be used by activities via model operations. Eurema supports the *reading*, *writing*, *annotating*, *creating*, and *destroying* of runtime models.

The FLD example in Figure 2.8 has four activities that describe a Self-Configuring feedback loop. First, an update activity reads the two runtime models Monitoring Rules and Architecture and synchronizes system observations with the Architecture model indicated by the writing model operation. Afterwards, an analysis activity checks if the updated architectural system model is optimal according to given goals. If the architecture is optimal, the execution of the feedback loop finishes (modeled by the optimal exit compartment). Otherwise, a planning activity calculates necessary adaptation steps considering different Configuration Strategies and annotates those at the architecture model. Finally, the Effect activity forces the planned adaptation steps to the running software system using the provided Synchronization Rules runtime model. The execution of the FLD stops at the final node, labeled with the Executed state.

The modeled Self-Configuring feedback loop in Figure 2.8 can be seen as a template. The deployment of the template is modeled in Eurema Layer Diagrams (LD) as shown in Figure 2.9. LD provide a snapshot of the adaptive system by modeling an instance situation of the layered

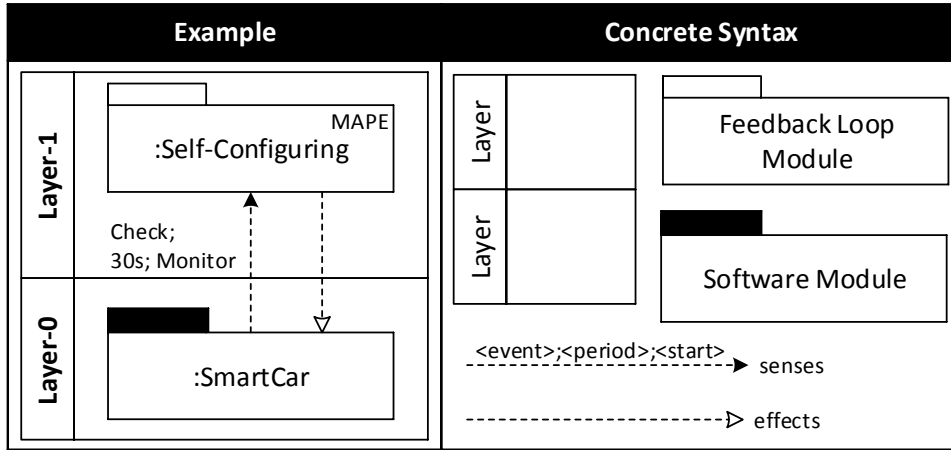


Figure 2.9: Eurema Layer Diagram (LD) for Self-Configuring

architecture of the adaptation engine. There, Eurema supports two types of modules, namely software modules and feedback loop modules, that can be deployed on different layers. Software modules encapsulate the domain logic and are treated as black boxes in Eurema, whereas feedback loop modules are white boxes that instantiate one of the beforehand modeled FLD templates. Furthermore, the LD specifies the triggering and effecting between modules. The concrete syntax of an Eurema trigger follows the `<event>;<period>;<start>` notation. The first parameter corresponds to the name of the event, whereas the second parameter denotes the minimal time (period) of two consecutive runs of the module. Finally, the start parameter identifies the initial node as starting point of the corresponding FLD of the module.

The example LD in Figure 2.9 has two module instances. The `SmartCar` represents the adaptable domain logic, which is modeled as black box software module. Additionally, there is one instance of the beforehand introduced `Self-Configuring` feedback loop. The corresponding feedback loop module is labeled with its internal adaptation steps that are in this case a full MAPE cycle (cf. Figure 2.8). Beside the deployment, a LD defines the trigger dependencies between modules. In the example, every time the software module sends a `Check` event, the feedback loop is triggered starting with the `Monitor` initial node. Furthermore, a period at the trigger defines the minimal time between two consecutive executions of the feedback loop. Consequently, the feedback loop in this example runs every time a `Check` event occurs, but at maximum every thirty seconds starting at the monitor initial node.

The resulting Eurema models are kept alive at runtime in a megamodel and can be directly executed by the Eurema interpreter. The interpreter waits until the modeled system event occurs and executes the corresponding feedback loop description (the FLD) by following the modeled control flow between the adaptation activities. Thereby, the internals of adaptation activities are not known by the interpreter and refer to a domain specific implementation. A comprehensive introduction of the Eurema modeling language can be found in [175].

Eurema focuses on self-adaptive software with non-distributed feedback loops [175]. Therefore, Eurema cannot be used for the specification of the adaptive behavior between distributed system interactions within an adaptive SoS. Furthermore, the Eurema interpreter is not able to execute multiple, parallel running feedback loops. However, the Eurema approach is a first step of considering feedback loops in self-adaptive software as first class entities, which operate on a set of runtime models as shown in the `Self-Configuring` feedback loop example in Figure 2.8.

Moreover, the Eurema models are kept alive by the corresponding execution environment, which leads to the runtime megamodel approach as outlined above. In the context of this thesis, the Eurema concepts are extended with respect to the characteristics of adaptive SoS. This includes, among others, collaboration aspects between distributed adaptive systems and the parallel execution of feedback loops, which contributes to the emergent behavior of the adaptive SoS. Furthermore, different system types from multiple domains within the overall SoS must be considered.

2.4. Collaborations in SoS

An important aspect in this thesis is the specification of system collaborations inside an adaptive SoS. Therefore, the term *collaboration* is at first clarified by retrieving definitions from the literature. Afterwards, existing approaches that facilitate the modeling of collaborations are discussed. Unfortunately, there are several synonymously used terms for collaboration such as cooperation or interaction. Parunak et al. [147] define a taxonomy of terms that are related to collaboration in multi-agent systems on bases of a literature review in [148].

This thesis uses the terms from Parunak et al. and transfers them in the context of adaptive SoS. An overview of the terms is depicted in Figure 2.10, where different layers form a hierarchy of terms. The basic terms of this taxonomy are *correlation* and *coordination*, which are depicted at the bottom of the figure. These basic terms define the notion of the behavioral joint interaction between systems. On the basis of these two terms, the difference between *cooperation* and *contention* can be defined, which is related to the intention of system interactions. Finally, at the highest layer in Figure 2.10, the usefulness of system collaborations can be defined by the terms *coherence* and *congruence*. In the following, all terms are introduced in detail.

Correlation

The correlation of systems inside a SoS is defined as the degree of behavioral joint information. If the behavior of a system, which can be observed by the visible performed actions, statistically depends on actions from other systems, both systems are correlated. The correlation metric can be empirically determined on the visible behavior. Thereby, the internals of the system must not be known as emphasized by [148]. Thus, the correlation is a number in form of a statistical fact that describes the degree of influence between systems on basis of joint information (observable actions).

Coordination

The coordination metric adds to the correlation a causal process [146], which requires communication between systems. Where correlation inside a SoS may appear between independent, randomly observable actions, a causal interaction needs an information flow between the systems, e. g., for negotiation. Furthermore, the fact that communication is necessary implies an appropriate infrastructure and different communication protocols [160]. There are a multitude of different protocols in literature depending on the underlying interaction problem that has to be solved. This thesis does not focus on the presentation of a specific interaction protocol, but rather provides modeling concepts to specify arbitrary interaction protocols between systems. The communication can be centralized or decentralized as well as direct or indirect, where an overview with example communication protocols from current literature is given in Table 2.1. A direct communication form could be the exchange of messages among systems in a client-server scenario [84, 109], whereas an indirect communication example

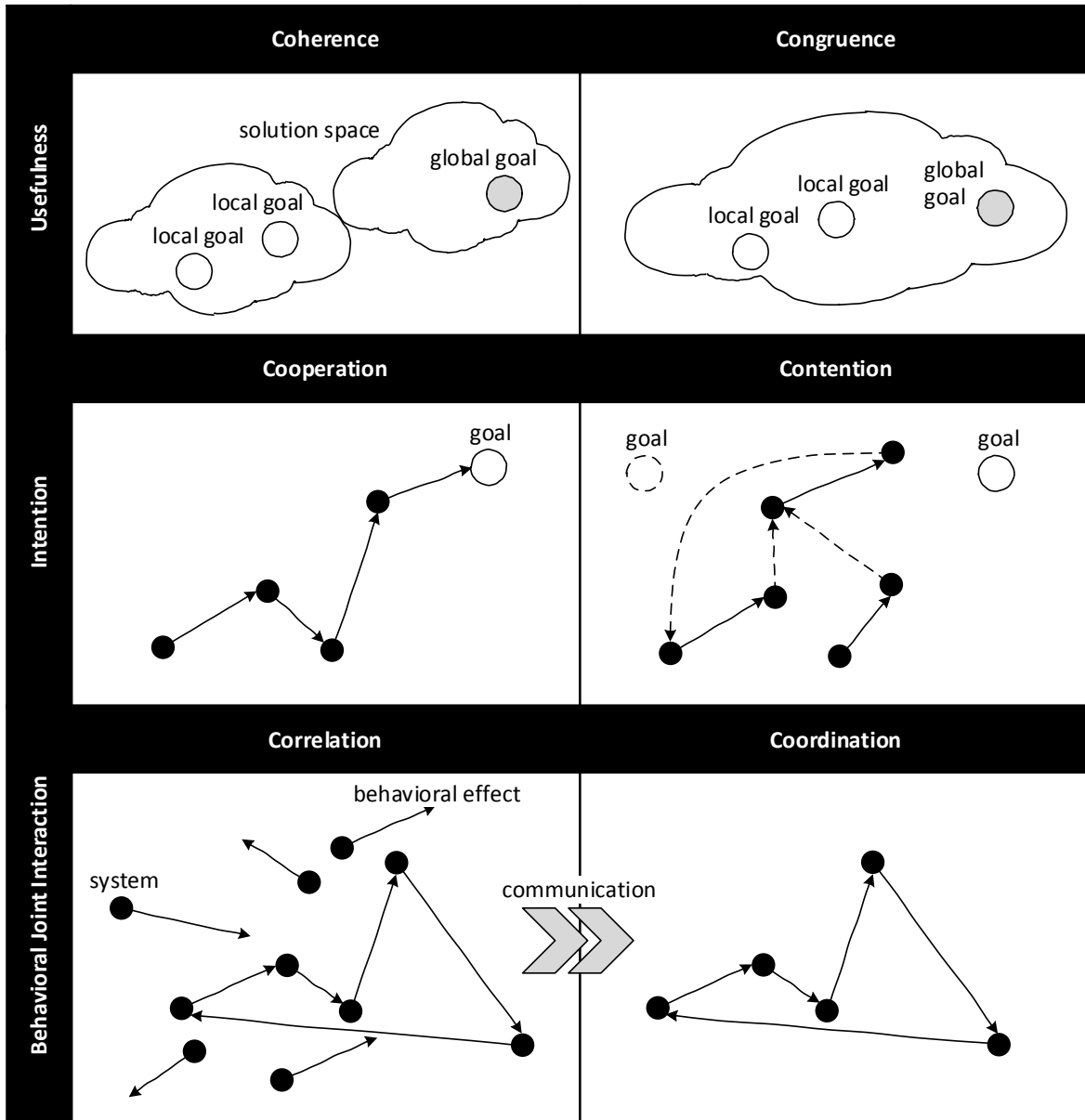


Figure 2.10: Collaboration terms

is the use of a blackboard approach as realized in [108, 190]. Centralized communication implies a dedicated system or group, which coordinates the message flow. An example for a decentralized, indirect communication is the stigmergy approach as introduced in [145, 148, 167], which is inspired by biological systems as for example the communication in ant colonies. In contrast, a decentralized direct message communication is used in peer-to-peer protocols such as introduced in [136]. Therefore, Figure 2.10 shows at the bottom how an empirical observation defining the correlation between systems turns into coordination, if communication is involved. The black dots in the figure are an illustration for systems, whereas the observable behavior is sketched as outgoing directed arrow. A dependency between systems is visualized by the appearance of another system (black dot), which starts its behavior after the observed behavioral action of the former system (the back dot appears at the end of the directed arrow).

Table 2.1: Categories of communication

	Centralized Communication	Decentralized Communication
Direct Information Flow	(1) Client-Server [84, 109]	(2) Peer-to-Peer [136]
Indirect Information Flow	3) Blackboard [108, 190]	(4) Stigmergy [145, 148, 167], Competition [114, 132]

Intention of Collaborations

If systems collaborate, e. g., by coordinating their interactions, the intention of each system defines whether the systems cooperate or work against each other. Therefore, cooperation requires joint interactions towards a common goal as shown on the left in the middle layer in Figure 2.10. In contrast, contention appears if the goal (intention) of at least one system disturbs the goal of another system as indicated by the dashed arrows and goals on the right in the figure. Therefore, the adaptive SoS should optimize the overall emergent behavior towards an increasing cooperation between independent systems by simultaneously decreasing the contention of joint interactions.

Usefulness of Collaborations

The terms coherence and congruence describe the usefulness of correlations, cooperations, and the intention of systems with respect of reaching local and global goals [148]. As motivated in Section 2.1.4, an adaptive SoS must balance between reaching its global goals and fulfilling local requirements of the contained systems. Therefore, the term coherence defines the degree of non conflicting local goals among systems. This is visualized on the left in the upper layer in Figure 2.10 by means of an abstract solution space. If local system goals can be realized within a common solution space, systems are coherent to each other [147]. Furthermore, if individual, local system goals satisfy or contribute to global SoS goals, the collaboration can be characterized as congruent as shown on the highest layer on the right in Figure 2.10.

Summarizing the collaboration terms from the literature, the correlation of system interactions defines a statistical fact with respect to observable system behavior. Due to additional communication, the correlation metric turns into a coordination scheme, which further implies a communication infrastructure and interaction protocol between systems. If systems coordinate their behavioral interactions, they can cooperate or compete each other, which is related to the intention of the joint collaboration. Thereby, the overall behavior emerges from the system collaborations within the adaptive SoS. To evaluate the emergent behavior, collaborations are defined as coherent if the interaction behavior is useful to reach local system goals contained in the overall adaptive SoS. Finally, the emergent behavior can be characterized as congruent, if system collaboration do not only contribute to local goals but rather facilitate reaching global SoS goals.

Modeling Collaborations

The collaboration concept is important for several system types. Consequently, there are different modeling approaches to describe system interactions, which comprises the specification of abstract collaboration roles, the interaction protocol and the communication mechanism. From the software engineering domain, the UML enables the specification of collaborations between software entities [87]. Other UML diagrams such as sequence diagrams or activity diagrams can be used to define the interaction protocol between these entities.

The Service-oriented architecture Modeling Language (SoaML) is a special UML profile that facilitates the collaboration modeling for service-oriented software architectures [88]. The *rigSoaML* modeling approach described in [30] extends the collaboration modeling for service-oriented systems by means of a verification concepts of joint system interactions. In the context of embedded systems, the Mechatronic UML (mUML) approach [100] facilitates the modeling, verification and simulation of collaboration behavior. None of these approaches above facilitate the explicit modeling of collaborations in adaptive SoS as done in this thesis. However, they already introduce preliminary ideas such as the explicit role modeling for joint system interactions or they emphasize the need of a separate modeling of the interaction behavior by means of a protocol and the local system behavior. Therefore, a running example is introduced in the following section, which describes the manifold collaborations within an adaptive SoS. Subsequently, requirements for a modeling language that facilitates the explicit collaboration description in adaptive SoS are derived from literature in the next chapter.

2.5. Running example

The running example of this thesis is inspired by the *European Telecommunications Standards Institute*, which proposes several use case scenarios and application standards for intelligent transportation systems for vehicles [71]. As depicted in Figure 2.11, different scenarios are combined to an adaptive SoS in form of smart cities that have different goals for optimizing the traffic situation within the city itself. Therefore, the smart city has to interact with geographically located neighbor cities, e. g., to predict incoming car commuters. Additionally, it must observe the status of its own traffic situation and interact with local traffic control systems as for example an intelligent traffic light system. Consequently, within the overall adaptive SoS, there are diverse systems of varying size that follow different purposes.

The biggest system type in this example is a smart city. In Figure 2.11, there are the two smart cities Berlin and Potsdam depicted and how they interact with each other by exchanging knowledge about the current traffic situation. Within each city, there are traffic members such as cars, buses, and pedestrians but also influencing traffic systems such as traffic lights, road works, and traffic jams. It is easy to consider a huge number of traffic members (over one million cars in Berlin) or influencing systems (over two thousand traffic lights in Berlin) in big cities that arise high demands on managing or predicting the traffic situation. Furthermore, as outlined by [71], there are several independent traffic scenarios in a smart city, where each scenario needs some coordination and follows a local strategy to fulfill a certain goal. For example, cars can build small platoons for saving fuel, autonomous driving, or following the current traffic flow through the city. Beside these advantages, platooning has high demands on a real-time communication between cars and positioning accuracy [41, 71]. Another example could be cameras placed along the roads to detect traffic jams [179]. Furthermore, scenarios are conceivable, where such cameras interact with the traffic lights or the traffic speed control system in the city to optimize the overall traffic flow. The requirements on such scenarios range from non critical information, such as warnings about road works, slow vehicles in front on the highway, or free parking lots, to critical interactions, as for example platooning or collision avoidance [71].

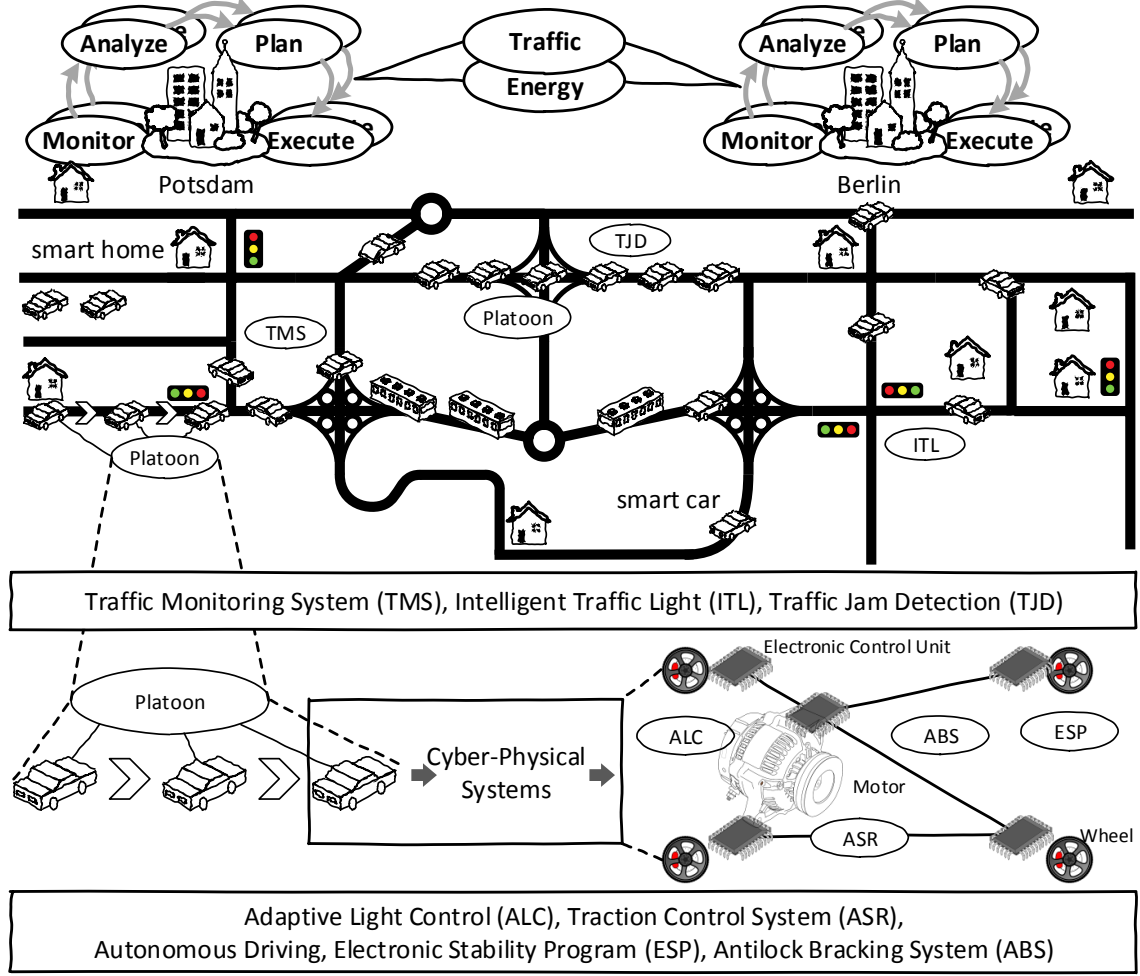


Figure 2.11: Smart city running example

Beside smart cities, this example considers smart cars that provide different adaptive functionalities to the driver as for example autonomous driving within a platoon, an Adaptive Light Control (ALC) to avoid blinding cars on the opposite direction, or other driving assistance functionalities such as an Antilock Braking System (ABS), an Electronic Stability Program (ESP), and a Traction Control System (ASR). Additionally to smart cars, smart homes are conceivable. On the one hand, smart homes locally optimize their pool of functionality according to the user, who lives in the home. For example, a smart home can close the windows if it starts raining or may start an intruder detection functionality if the user leaves the house. On the other hand, smart homes contribute to the emergent behavior of the adaptive SoS by collaborating with other contained systems. For example, the smart home can communicate with the users car to turn off or on heating if the user leaves respectively arrives the house. Furthermore, it can report the current energy consumption to the smart city so that the overall energy production of the adaptive SoS can be optimized.

However, the smart city together with the contained systems such as smart cars and smart homes show the characteristics that are in the scope of this thesis. First, the overall city is adaptive according to the current traffic situation that enables the specification of multiple,

interacting feedback loops on different levels, e. g., between smart cities, smart cars or smart homes. Second, there are a various number of system types with different complexity such as embedded systems and CPS. Third, information on different levels of abstraction must be exchanged that fits to the idea of using runtime models as knowledge representation within the adaptive SoS. Fourth, systems must interact with each other to fulfill overall goals, where one system alone is not able to reach a global goal, which suits the use of specifying the collaborations between systems. Fifth and finally, the smart city is a large scale example, which can easily divided into a subset of smaller examples of independent, interacting systems that targets the need of analyzing and simulating the emergent interaction behavior towards the understanding of the overall SoS behavior.

3. Modeling Language Requirements

After the definition of different MDE techniques that are important for this thesis and the common understanding of different system types, this chapter derives characteristics for an adaptive SoS in Section 3.1. Thereby, the characteristics are subsumed from the state of the art literature and experience from former own work in [11]. On basis of these characteristics, a set of requirements for the Deurema modeling language is derived in the subsequent Section 3.2. Those requirements are considered for the design of the different Deurema concepts and are important for the evaluation of the modeling language. Finally, the state of the art for modeling adaptive SoS is discussed in Section 3.3.

3.1. Characteristics

As outlined in Section 2.1, an adaptive SoS is an ensemble of systems that comprises a broad spectrum of system types including embedded systems, NES, CPS, and NCPS. Therefore, the overall SoS inherits the characteristics from its contained systems that are derived in this section.

The main characteristics are already caused by CPS that are stated as a federation of *"open, ubiquitous systems of coordinated computing and physical elements, which interactively adapt to their context, dynamically and automatically reconfigure themselves, and cooperate with other CPS"* [45]. Therefore, there are five main characteristics of an adaptive SoS that are the *openness* (C-1), evolving *dynamic structures* (C-2), system *collaborations* (C-3), system *interdependence* (C-4), and *incomplete knowledge* (C-5). On basis of these five main characteristics, there are further refined SoS properties as follows:

C-1 *Openness*: The SoS is considered as open, because it is a federation of systems, where an arbitrary number of unknown, potentially heterogeneous system parts may leave or join the overall SoS during system operation at arbitrary points in time [24, 39, 45, 140, 172]. As a consequence, system borders evolve over time [107].

C-1.1 *Diversity*: The diversity property of a SoS refers to a varying set of different participating system types that enable needed functionality to achieve SoS goals [64]. Furthermore, Moschoglou et al. [140] emphasize the *heterogeneity* of a SoS with respect to different system types that join and leave the overall SoS during lifetime, which is used synonymously for the diversity characteristic. Additionally, Boardman et al. [39] define a diversity property that refers to the *managerial independence* of the system, which is covered by the C-4.3 characteristic below.

C-1.2 *Distribution*: Within the SoS, the integrated, autonomous systems are inherently distributed, potentially over larger geographic spaces [81, 130, 140]. The distribution characteristic causes further challenges of controlling and enabling communication capabilities between spatial separated systems.

C-1.3 *Scalability*: The integration of independent, diverse system solutions and for the most cases the self-coordinating, decentralized collaboration scheme between

those systems leads to a large-scale federation of systems [107]. Thus, SoS must be considered as scalable by managing a potential huge amount of containing systems.

C–1.4 *Flexibility*: The flexibility characteristic of the SoS enables the reaction of the system to changes in the environment and system structure [81, 107]. Boardman et al. [39] call this property *belonging*, which goes hand in hand with an *elastic* overall system architecture. Furthermore, the SoS is considered as *agile*, if it is flexible and able to react on changes rapidly [172].

Running example: The openness and its derived characteristics can be identified within the smart city running example of this thesis as introduced in Section 2.5. If vehicles drive from one city to another, the smart city must be open (C–1) to recognize the changing traffic situation of incoming and leaving vehicles. Furthermore, different vehicle types such as cars, buses, motor bikes, or bicycles, but also other participants of the smart city such as smart houses, traffic lights, and the road infrastructure enforce the diverse (C–1.1) property of the overall adaptive SoS. The distribution (C–1.2) characteristic is inherent by the spatial distribution of vehicles, smart houses and traffic lights within a smart city. The potentially huge number of vehicles and other participants refers to the scalability (C–1.3) characteristic, whereas the dynamic handling of changing traffic situation requires the mentioned flexibility (C–1.4) of the SoS.

C–2 *Dynamic Structure*: The dynamic aspect of a SoS is caused by the openness and describes the ability of the system to change its internal structure (self-modification) and cooperation scheme according to changes in the environment and current needs [45]. Due to the varying pool of functionality, the dynamics of a SoS raises the challenge of choosing the correct set of components and capabilities to achieve current goals [64].

C–2.1 *Adaptive Behavior*: The SoS is adaptive such as each autonomous system can adapt its structure as well as behavior to the particular needs and constraints in the current SoS configuration [64]. Furthermore, an adaptive SoS exploits the full potential of autonomous, beforehand isolated systems by building interconnections and collaborations [107, 172]. Boardman et al. [39] refer to the adaptive characteristic with the term *connectivity*. Additionally, the SoS handles the emergent behavior by dynamically adapt and absorb deviations in the system structure as emphasized by [81, 140].

C–2.2 *Resilience*: On the one hand, the SoS is robust in the sense that it reduces the likelihood to experience failure due to the complexity, the dynamic configuration, and potential emergent behavior of the SoS (resilience). On the other hand, the SoS is less vulnerable to catastrophic and single point of failures [172].

Running example: The dynamics (C–2) in the smart city arises by the several adaptive (C–2.1) capabilities of included systems. For example, a car has different adaptive control functions as ALC, ASR, ESP, and ABS. Therefore, each vehicle behaves according local situations, which leads to an overall dynamic, aggregated SoS behavior. Furthermore, the failure of a vehicle, e. g., due to an accident, or traffic light, e. g., due to a power supply failure, does not break the overall operation of the smart city. Thus, the SoS is resilient (C–2.2) to partial system failures, which further causes adaptations (C–2.1) of the remaining functionality or introduces additional behavior as for example the rescue of injured humans in the case of a car accident.

C–3 Collaboration: Due to the definition of a SoS in Section 2.1.4, the independent systems within the SoS must be collaborative to achieve common goals [45, 111]. The collaboration characteristic is included in the term *connectivity* from Boardman et al. [39]. Furthermore, collaborations in SoS are a key aspect for Jamshidi [106] and Moschoglou et al. [140], who refer to the term *interoperation* instead. From the collaboration term discussion in Section 2.4, this characteristic is a consequence from the cooperation intention as well as joint interaction of contained systems inside the SoS.

C–3.1 Emergence: Due to the collaborative nature, *emergent behavior* is the result of the overall SoS behavior by aggregating the local behaviors of the contained autonomous systems and their interactions [39, 81]. Emergent behavior is considered as nonlinear [64], which means that the overall SoS behavior cannot be derived by analyzing single system behavior capabilities. Thus, emergent behavior is more than a simple aggregation of single behavioral aspects, which must include collaborations and new arising capabilities due to joint interactions. Moschoglou et al. [140] use the term *cooperation* synonymously, which includes collaboration and coordination of systems. According to the collaboration definition of this thesis, Parunak et al. [148] refer to the overall usefulness of collaborative behavior, which should be *coherent* (non conflicting) within the SoS.

C–3.2 Competition: Beside the emergent, collaborative behavior, a SoS is competitive, because each system follows and optimizes its behavior according to local goals [81]. In the worst case, this leads to contradicting behavior on the SoS level [181]. Consequently, there are the two dimensions in an adaptive SoS of cooperation and contention (cf. definition in Section 2.4). The SoS is responsible of finding appropriate trade-offs between local and global goal optimizations, which follows the argumentation of Parunak et al. [147, 148] looking at the overall *congruence* in the SoS behavior.

Running example: The collaboration (C–3) characteristic arises due to the manifold pool on possible collaborations on different layers in the smart city example. On the highest abstraction layer, smart cities collaborate with each other to exchange knowledge about the current traffic situation or energy prosumption.¹ Furthermore, within a smart city, different collaborations as for example platooning, intelligent traffic light control, or collaborative crossing of junctions lead to an emergence (C–3.1) of behavior. Thereby, each system tries to optimize local goals. For example, the driver of a car maybe wants to reach his destination point as quickly as possible, whereas another driver tries to save fuel. This might introduce competitive (C–3.2) behavior, if both drivers are in the same platoon or cross the same junction. Thus, the corresponding collaboration or responsible smart city has to solve such arising conflicts.

C–4 Interdependence: Caused by the collaborative (C–3) characteristic, Correa et al. [64] emphasize that there is a range between independent and totally dependent systems inside a SoS, which leads to the combined interdependence characteristic. On the one hand, as stated in [81], a SoS is independent in the sense that all systems are independently designed, developed and managed for their own purposes. Furthermore, systems operate independently at runtime except during the collaborations [45]. On

¹The term *prosumption* is used in the domain of smart grids and combines the two aspects of production and consumption of energy.

the other hand, systems may lose some freedom during operation, if central control authorities coordinate and orchestrate the interaction of systems during collaborations. The interdependence characteristic determines both dimensions and indicates the varying combination of dependent and independent behavior within the adaptive SoS.

C–4.1 *Local Evolution:* The *development* of a SoS is characterized by uncoordinated, independent local evolution steps of the autonomous systems, which may influence each other, rather than a global development plan that is followed. These evolution steps can happen offline or at runtime depending on changing purposes, demands or requirements [81, 140].

C–4.2 *Autonomous Systems:* The SoS is *operational independent* [81], which means that the autonomous systems are operated independently from each other and that no global coordination scheme, concerning the operation of the autonomous systems, can be assumed [140]. Especially for the FoS type [106], “*there is no central control entity and decision making is done in collaboration and cooperation*” [140]. The operational independence might decrease due to collaborations of systems, but is necessary to reach global SoS goals and may simultaneously increase the achievement of local system goals [81].

C–4.3 *Incoherent Development:* The *managerial independence* [81] of a SoS implies that also the management and design of the autonomous systems are not centralized and thus, the management/design decisions during the development for different autonomous systems may be conflicting. In general, an independent and incoherent development of systems, which are later contained in the SoS, should be assumed.

C–4.4 *Concurrent Changes:* The application effects of a SoS are characterized by concurrent changes in the environment. Concurrency effects may spread over the system and influence a bunch of local systems due to established collaboration connections.

Running example: The interdependence (C–4) characteristic can be reasoned by the independence of smart cities, vehicles, smart homes, and traffic lights and the dependence due to collaborations between those entities. Thereby, vehicles may exist in different product lines or have different variants of the deployed software (e. g., due to software updates), which is caused by evolutionary (C–4.1) development over time. Also the aging of hardware in cars and other systems as well as hardware failures lead to an evolution of available capabilities of the overall adaptive SoS. Furthermore, vehicle drivers can be seen as operational independent (C–4.2) as long as the driver does not join collaborations such as platooning. Beside vehicles, traffic lights operate independently from the current traffic situation, if no intelligent capabilities, such as monitoring the waiting vehicles, are available. The managerial independence (C–4.3) is enforced by the fact that vehicle manufactures make different design decisions during the development as companies that develop traffic lights. Even within one domain as for example the development of cars, manufactures have to develop different products to sell their cars on the market. Finally, concurrency (C–4.4) effects can be easily observed in the running example, e. g., independent driving of cars through the city changes the car position over time and thus, the state of the environment concurrently.

C–5 *Incomplete Knowledge:* The incomplete characteristic has the focus on the *knowledge* within the SoS. Each independent system has always only incomplete views of the overall

SoS and its environment. These views may be extended during the collaboration with other systems or by sensing the local environment [81]. Furthermore, due to concurrency effects and decentralized coordination, the SoS has to deal with inconsistencies by aggregation and distribution of the knowledge from and to systems. In general, a complete and comprehensive view on the overall SoS state with an aggregated, unified view of the knowledge cannot be given.

C-5.1 *Limited Information:* Each system maintains local (internal) models for anticipating or predicting its own architecture and behavior [64]. Because of privacy aspects, systems are not in general willing to provide all internal data to the outside. The provision and discovering of data is necessary for joint collaboration, which leads to partial access and visibility restrictions to foreign knowledge [140]. Therefore, the limited information characteristic of a SoS considers the restricted access to knowledge and partial knowledge exchange.

C-5.2 *Fluent Communication:* Correa et al. [64] emphasize that the flow of exchanged information varies through the whole lifetime of the SoS. Due to changing collaborations, needs, or available systems, the information flow must be always adapted to achieve given goals. Thus, the SoS has a fluent knowledge transfer, which includes changing communication partners and mechanisms.

Running example: In the smart city, knowledge must be transferred between different system types. Within a platoon, vehicles must coordinate with each other and share information about the platoon status, own state, and observed environmental aspects of interest (e. g., obstacles). However, each vehicle provides only enough data to successfully operate in the platoon. Another example is the exchange of traffic information between smart cities. First each smart city has a local, restricted view on its own traffic situation, which is a subset of the overall traffic situation on the complete road network and thus, incomplete (C-5). Second, each smart city provides, on the one hand, only traffic data that is necessary to guarantee proper traffic flow management as it is defined by the corresponding collaboration. On the other hand, smart cities are not allowed to violate governmental laws by collecting or publishing private data. Therefore, each smart city has only limited (C-5.1) access to the data in the overall adaptive SoS. Third and finally, the knowledge about the traffic situation changes all the time, which causes an unlimited data flow (C-5.2) by collecting and transferring the knowledge. Changes in the available communication channels or technologies further influence the way of how the knowledge is transmitted.

Concerning the goals of this thesis, the characteristics above influence the design as well as collaboration concepts of the Deurema modeling language. The openness characteristic (C-1) implies that the overall SoS is composed of a varying number of systems. As a consequence, clear SoS borders cannot be defined because systems interactively join and leave the overall SoS at arbitrary points in time [45, 107]. Furthermore, the overall behavior of the SoS emerges from the available systems, whereas the size and number of systems are usually neither known upfront nor limited during the SoS lifetime. Therefore, a SoS must be considered as a federation of a large number of integrated systems that goes hand in hand with the diversity (C-1.1) and distribution (C-1.2) characteristics. This distribution of the system can be twofold. As first aspect, the SoS can be logically distributed in different parts that are executed on one physical node. Additionally, because of the potential large size of the SoS, the geographic distribution of the overall system is very likely [172]. Beside the distribution

aspect, the SoS offers high potentials concerning scalability (C-1.3) and flexibility (C-1.4) characteristics. The resulting elasticity and agility of the overall SoS is needed to cope with the permanent changing of the inner system architecture as well as with the interaction with an unknown affecting environment [172]. This causes further challenges concerning the availability of a certain functionality, the reliability of the system, fault tolerance, security and safety issues. With respect to collaboration, the Deurema modeling language must be aware that new collaboration capabilities may potentially arise or disappear at every point in time.

Caused by the openness, the dynamic characteristic (C-2) of a SoS arises from the integration of beforehand isolated system solutions such as CPS. Edward A. Lee emphasizes that a *“Cyber-Physical System is an integration of computation with physical processes [...] usually with feedback loops”* [126]. Those feedback loops enable a dynamic adaptation (C-2.1) of the CPS according to changes in the environment or the system itself. The adaptation ability of each CPS or other system types emerges in the overall adaptive SoS. Furthermore, the adaptation capabilities (C-2.1) of the SoS in combination with the flexibility capabilities (C-1.4) lead to the resilient (C-2.2) characteristic of the SoS in the sense that the likelihood of catastrophic and single point of failures is drastically reduced [172]. From the modeling perspective, the modeling of feedback loops is well known for physical aspects as widely done for embedded systems. Modeling the physical system is not enough for CPS, NCPS and SoS. Here, the cyber part has to be considered, too. Especially, feedback loops controlling the software part of the autonomous systems and their interaction with other systems have to be covered as well [58, 111, 127].

The importance of modeling the interaction aspects between systems is strengthened by the collaborative characteristic (C-3) of an adaptive SoS, which can be derived from the openness (C-1) and dynamics (C-2) capabilities. Usually, each autonomous system inside an adaptive SoS must adapt its behavior according to its peer systems, while considering the interplay between its own behavior, the other systems' behavior, and the overall system-level behavior. Therefore, on the one hand, the overall system behavior emerges (C-3.1) from the contained systems in the sense that different system parts work together to reach global goals that cannot be achieved by a single system alone. On the other hand, each system behaves according to local optimization strategies that might influence, or even worse, contradict other systems [181]. This phenomenon is described by the competition characteristic (C-3.2), where the SoS is responsible of finding appropriate trade-offs between local and global requirements [81, 148]. Moreover, due to the composition of systems that autonomously adapt (C-2.1) to their individual contexts into the SoS, also unwanted co-adaptation effects may emerge from interference of the individual feedback loops [133]. A modeling language for collaboration should comprise such effects and should be able to visualize, analyze as well as simulate them.

Ruling such adaptive SoS is challenging due to the complexity (C-1), dynamics (C-2), and emergence (C-3). The fourth characteristic C-4 copes with the interdependence of a SoS. At first, the autonomous systems evolve over time (C-4.1) caused by hardware and software aging. This holds for the evolutionary development cycles as well as for the evolution of the system during system execution, e. g., due to maintenance activities as outlined in [58, 127]. Such local evolutions steps in independent systems leads to uncoordinated evolution in the overall adaptive SoS. Another aspect is the operational independence (C-4.2) [81] of the systems within the adaptive SoS. In general, a global coordination scheme controlling the operation of all systems cannot be assumed. Therefore, each system must be seen as operational independent from the other systems following local strategies. The operational

independence goes hand in hand with the managerial independence (C-4.3) [81] of the adaptive SoS. In general, no central instance exists that is able to coordinate the interaction between all systems and determines all local management, design, or development decisions. Thus, the adaptive SoS is characterized by a decentralized coordination scheme. Additionally, the concurrent characteristic (C-4.4) inherently arise due to the independent behavioral effects of the contained systems of the adaptive SoS.

The last, major characteristic focuses on the partial knowledge aspect and denotes the view over the available knowledge in the adaptive SoS as incomplete (C-5). Each system has always a local view, which is only a part of the overall SoS and its own environment, as basis for its own decision making. The local knowledge can be updated by sensing the environment or extended via exchanging data in collaborations. However, the adaptive SoS should be aware of different local views in the systems, outdated knowledge, limited access (C-5.1) to data, or inconsistencies by updating data from different sources. In general, it is not possible to provide one unified, global knowledge base that describes the whole SoS and its environment. However, due to sensing activities and collaborations, there is a continuous knowledge flow (C-5.2) throughout the adaptive SoS.

In summary, considering the discussed characteristics, it is highly attractive that SoS are self-adaptive at the level of the individual autonomous systems and at the overall SoS level to cope with the emergent behavior, to adapt and absorb open, dynamic, and deviating SoS architectures, and to adapt to open and dynamic contexts, while considering the shared ownership of knowledge inside the systems. On basis of the discussed characteristics, model language requirements are derived in the next section.

3.2. Requirements

According to the goals of this thesis and the discussed characteristics of adaptive SoS, the engineering of self-adaptive SoS must explicitly cover the coordination among the individual systems. Because the systems themselves usually have adaptive capabilities, which further can be realized in form of feedback loops [49, 58, 126], the focus of this thesis is on SoS that are composed of systems, which contain such feedback loops. Furthermore, feedback loops follow the MAPE-K blueprint from [110] as introduced in Section 2.1.1. This imposes several requirements for designing (modeling) and realizing the coordination. The following requirements are derived from the characteristics discussed in the former section, from Weyns et al. [183], who enumerate several research questions for decentralized and distributed control loops, from the Eurema modeling language, which is the predecessor modeling language to Deurema, and from own former work in [10, 11]. Requirements that are directly taken from the Eurema modeling language [175] are marked with a '*'. Furthermore, differences to existing requirements in [175] are highlighted in the detailed requirement discussion below.

The modeling language requirements are semantically grouped in six different parts that are *feedback loops and coordination* modeling (R-01 to R-08), *runtime models* for knowledge representation (R-09 to R-12), *communication* mechanisms (R-13 to R-14), *adaptation* concepts (R-15 to R-17), *development* (R-18 to R-20), and *analysis* (R-21 to R-25).

Feedback Loops and Coordination Requirements

R-01* Explicit Feedback Loops: Following the external adaptation approach [156] decouples the adaptation logic from the domain logic into the two parts adaptation engine and adaptable software (cf. Section 2.1.1). Therefore, the design and analysis of the

adaptive behavior becomes a crucial point by modeling the software system. Thus, the explicit description of feedback loops is required [49]. Furthermore, existing approaches separate the feedback loop in dedicated activities as described by Kephart et al. [110], which should be supported by a modeling language, too.

R-02* Intra-Loop Coordination: The existence of different activities inside a feedback loop causes the need of specifying a causal order between activities. The modeling language should support the design of intra-loop coordination by means of explicitly modeling of the control flow between activities [179].

R-03* Inter-Loop Coordination: On the one hand, a SoS contains several systems. This further causes new demands on modeling the coordination between multiple, decentralized feedback loops [183]. On the other hand, even within one system, different concerns, such as performance optimization and failure handling, are usually modeled in distinct feedback loops [174, 182]. However, multiple feedback loops must be meaningful coordinated to reach the expected adaptation effects. Modeling dependencies between feedback loops is known as inter-loop coordination [179], which should be supported by the modeling language. The existence of several feedback loops causes further requirements such as triggering, distribution, and collaboration.

R-04* Triggering: Modeling the inter-loop coordination allows the specification of a causal order between feedback loops. Furthermore, temporal aspects should be considered that describe when a feedback loop is executed. This can be modeled with trigger conditions, which include causal and temporal aspects.

R-05* Distribution: As emphasized by the distribution characteristic (C-1.2), adaptive SoS are potentially distributed over large geographic spaces. Furthermore, the inter-loop coordination requirement (R-03) directly raises the question towards the distribution of feedback loops and their deployment to systems. Therefore, a modeling language should be aware that interacting systems and the corresponding feedback loops can be distributed over large distances or may run on different execution nodes. This affects, among others, the communication mechanisms between systems, causes timing delays during knowledge exchange, or arises security issues by crossing system boundaries over collaborations. Consequently, a modeling language should consider such distribution effects and should provide modeling concepts for that.

R-06 Delegation: The availability of multiple adaptation capabilities and their distribution over several systems allows the delegation of adaptation activities. Therefore, a modeling language should support the invocation of available, possible distributed adaptation activities (services) or the transfer of resource intensive computing tasks.

R-07 Roles: The coordination of distributed feedback loops should cover the specification of the different participants. It seems reasonable to specify the coordinating participants by means of *roles* to foster separation of concerns with respect to behavior of participants relevant and irrelevant to the coordination. For example, the invocation of a remote service as requested in R-06 should explicitly specify the provider and the consumer of the service. This includes the specification of the local behavior of MAPE activities relevant for the coordination to accomplish the self-adaptation.

R-08 Protocol: The interplay between roles should be specified in a protocol to achieve the aim of the coordination. Likewise, some processing or filtering of exchanged knowledge according to the current role is conceivable before sending or after receiving the knowledge. Therefore, the modeling language should support the specification of the employed protocol coordinating the behavior among distributed MAPE activities, that is, the sequence of interactions among them such as push-pull, request-reply, or negotiation. This corresponds to the sequence of exchanged messages or knowledge among MAPE activities. Additionally, operational (C-4.2) and managerial (C-4.3) independence of systems requires well-defined collaboration contracts and interfaces that define in which form the required knowledge is exchanged.

Running example: The smart cities and the contained systems such as vehicles in the running example of this thesis in Figure 2.11 separate the adaptation logic from the domain logic following the MAPE-K approach, which imposes the explicit modeling of feedback loops (R-01). Furthermore, there are different adaptation activities that have to be coordinated, which is described by the intra-loop dependencies (R-02) within the corresponding system. Because the running example considers several adaptive subsystems within one vehicle, multiple feedback loops have to be coordinated (R-03), too. The spatial distribution as well as coordination of the adaptation logic in different vehicles and cities directly cause the need of considering trigger conditions (R-04) as well as distribution aspects (R-05). For example, a car may trigger a dedicated service to provide its current position in the road network, whereas the nearest service center is used according to the spatial distribution. Additionally, modern cars allow connecting the mobile phone with the car to enable a hand-free speaking during driving. In this case, the delegation of tasks (R-06), as for example voice processing or mobile connection handling, from the resource restricted mobile phone to the more powerful car is thinkable. Thereby, different protocols (R-08) can be deployed that further define dedicated roles (R-07), such as hierarchical client-server communication.

Runtime Models Requirements

R-09* Runtime Models: Feedback loops are characterized by MAPE activities sharing a common *knowledge* base [110] and thus, the available knowledge influences the collaboration and adaptation capabilities of the system. The modeling language should support runtime representations of knowledge. Models at runtime [38] (MART, cf. Section 2.2.3) are able to represent system information during system execution. This technique enables keeping alive development models at runtime and uses these kinds of models to represent the executed software system. Therefore, a modeling language should support the specification of adaptation activities together with the available knowledge in the system.

R-10 Knowledge Management: Beside the availability of runtime information during system execution, the semantic and representation of knowledge must be clear. Due to collaborations between systems inside a SoS, data formats must be transformed to link different systems and to achieve interoperability. For example, runtime models can be dynamically manipulated on a much higher level of abstraction to achieve interoperability, e.g., by automatically transforming them to different formats, forcing runtime model manipulations to the running system, or enabling runtime analysis and verification. This forces a common understanding of knowledge artifacts in the system (similar to ontologies) and enables different participants to define what information

types are shared. A modeling language should be aware of different knowledge artifacts, must explicitly determine different knowledge types, and should offer a clear concept of managing different information in the adaptive SoS.

R-11 Partial Knowledge: The inherent complex and distributed nature of adaptive SoS leads to multiple local views in the systems and different versions of data over time [59, 86]. Therefore, the modeling language should consider partial knowledge and conflict resolution mechanisms for different versions or even outdated data.

R-12 Knowledge Exchange: Beside the knowledge management (R-10), the modeling language should support the specification of which, how much, and in which way knowledge is exchanged. Therefore, when specifying a coordination protocol (R-08), it has to be addressed which knowledge is shared or exchanged among distributed MAPE activities. This includes characteristics of the knowledge such as whether it is locally or globally accessible by MAPE activities or whether it is partitioned and replicated in the system. Moreover, it should be made explicit how the exchanged knowledge is processed by the activities and in which way it is relevant for the collaboration roles (R-07), which might further affect local adaptation behavior. Additionally, the modeling of knowledge exchange should include considerations about the two dimensions that are completeness and time. The completeness dimension tackles the specification of the necessary amount of data that has to be shared within the collaboration. Because of the distributed nature of a SoS, a complete transfer of the local knowledge can be very inefficient or may raise security issues. Therefore, the modeling should support full and partial knowledge transfer including different local filters (views) or optional knowledge exchange. Concerning the timing dimension, the modeling language should be aware of the timeline of the knowledge. It might be helpful to exchange information that is collected over a time period, of a specific time frame or knowledge that is not older than a given timestamp. The explicit modeling of knowledge with a clear semantic and its local as well as collaborative usage enables further impact analysis in the sense that changes in the knowledge base by one participant may influence the behavior of an interacting system over a corresponding collaboration.

Running example: The explicit representation of available knowledge at runtime (R-09) is crucial for the adaptation logic and collaborations in the smart city. For example, the complexity of a physical vehicle can be reduced by determining key points of interest such as the current position, vehicle kind, or aimed target position and their representation in appropriate models. If that information is kept alive during the lifetime of the overall adaptive SoS, it can be used for online analysis such as the identification of traffic jams. An analysis is only possible if, on the one hand, the smart city has a clear notion of the semantic (R-10) of the runtime information. And on the other hand, if the smart city is able to cope with a continuous flow of partial knowledge (R-11), which is provided in different points in time from vehicles or other sensors. Furthermore, noise in sensor data, concurrent access to knowledge, and contradicting or outdated information have to be resolved on different levels in the adaptive SoS. The requirement of knowledge exchange (R-12) is essential for the collaborations in the system. If for example two cars want to successfully drive in a platoon, they have to coordinate each other by exchanging messages and data.

Communication Requirements

R-13 Communication: Besides specifying the roles and protocols of an interaction between systems, the underlying communication mechanism has to be defined. Especially, if the communication influences the coupling of the roles, when executing the protocol. Examples for communication paradigms are direct message exchange, either synchronously or asynchronously, or indirect blackboard communication. The modeling language should support different communication paradigms and hide implementation details for realizing the communication infrastructure.

R-14 Synchronization: Additional to different communication mechanisms, the modeling language should provide concepts for synchronizing collaborative interactions at dedicated points in time. Different synchronization methods are known from parallel programming such as barriers, mutexes, futures, or semaphores. The modeling language should provide a meaningful subset that can be used during collaboration. In general, the use of synchronization mechanisms may lead to time-based deadlocks and thus, may block the collaboration. The modeling language should enable an analysis of modeled deadlock behavior.

Running example: Due to unreliable networks, a mobile client in the smart city should communicate (R-13) asynchronously by directly exchanging messages to the server. Other examples are ping pong messages or heartbeats within a platoon to indicate the correctness or liveness of a collaborating system. Sometimes explicit synchronization (R-14) between participants is necessary. Consider for example an existing platoon, where a new car wants to join and therefore, may influence the behavior of the whole platoon. In that case, it can be useful to distribute the information of a new incoming car to all members in the platoon and wait until each member locally adapts to the new situation.

Adaptation Requirements

R-15 Reconfiguration: There are different variants of adapting the behavior as already introduced in the preliminaries in Section 2.1.1. Applying reconfiguration techniques has the advantage that the possible configuration space is known in advance. Therefore, applying a reconfiguration suits resource restricted systems such as embedded or cyber-physical systems. Furthermore, it enables the changing of the system structure or behavior under timing constraints as necessary for hard real-time systems. Thus, a modeling language for adaptive SoS should support the specification of predefined reconfigurations.

R-16* Adaptation: Adapting the system structure or behavior is another variant to react of changing needs. Adaptation techniques are usually more flexible than system reconfiguration and the configuration space must not be known in advance. Therefore, a modeling language should provide concepts for adapting the specified SoS behavior. The Eurema modeling language focuses on structural adaptation only as emphasized in [175]. This restriction is not required in the context of this thesis. Thus, parameter and structural adaptation as described in the preliminaries in Section 2.1.1 should be considered.

R-17* Meta-adaptation: Meta-adaptation is the ability to adapt the adaptation logic itself, which is specified in the adaptation engine. The Eurema modeling language focuses on the reflection and structural adaptation of feedback loops. This thesis extends this view by the more general demand for meta-adaptation of all entities within the adaptation engine of the adaptive SoS such as runtime models or contained systems.

Running example: Allowing all kinds of adaptations for cars within a smart city might cause safety or governmental law violations. Therefore, predefined software configurations (R-15) are thinkable. For example, a car may offer different driving modes such as platooning, comfort, or sportive. All these modes can be engineered and provided during the development of the car. During lifetime, the driver can choose his preferred configuration and the vehicle changes the behavior accordingly. On the other hand, dynamic adaptation mechanisms (R-16) are thinkable for noncritical applications such as dynamic parking slot services or traffic jam warnings. The requirement of meta-adaptation (R-17) arises by looking at the different adaptive layers inside the SoS. Therefore, at the highest layer, the smart city may adapt the goals and reconfiguration capabilities of the layer below, which comprises for example the behavior of the traffic lights to adapt the overall traffic flow. The changed policy of controlling the traffic may raise further adaptations at the layer below that tackles the vehicles on the road. For example, a car may adapt its behavior and propose a mode change, e. g., from sportive to platooning, because of a high traffic density.

Development Requirements

R-18* Offline and Online: Due to software evolution and aging [16, 77], there is a coexistence of software adaptation activities that are performed offline or online. Performing online adaptation is realized by the adaptation engine, which enables an appropriate derivation of the behavior according to changing needs. However, it might happen that the adaptation engine cannot cope with an upcoming need. In such cases, the software must be externally adapted, e. g., by software maintenance activities, which is called offline adaptation. Andersson et al. [16] show that the coexistence of offline and online adaptation is required. As a consequence, a modeling language should support both kinds of online and offline adaptations as well as their coexistence. This requirement is taken from the Eurema modeling language and further motivated in [175].

R-19 Pattern: Since there is no dichotomy of centralized and decentralized control, there is a range of different patterns determining the degree of decentralization [183]. For example, systems within an adaptive SoS can be organized in a hierarchy, where dedicated controllers are responsible to coordinate parts of the hierarchy below. In contrast, other solutions can be completely decentralized by following the idea of swarm algorithms or independent agents. Being able to specify the coordination among distributed MAPE activities with all their facets is a promising way to model in detail various control patterns [183]. Having a uniform modeling approach enables the specification, comparison, and reuse of solutions to build a pattern catalog at different levels of abstraction. A pattern catalog enables to choose appropriate partial solutions depending on different selection criteria as for example robustness, ability to reach local and global goals, scalability by means of the amount of data to be processed, and overhead of the interaction/communication. The next step after a pattern catalog would be a standardization of coordination elements such as roles, protocols, knowledge exchange, and communication techniques.

R–20 Different Domains: As motivated above, a SoS comprises several system types and thus, tackles software and system engineering concepts from different domains. Specifying the adaptive behavior and collaborations in the overall SoS raises the need of finding modeling concepts that integrate different domains and enable the interaction between various system types. A modeling language must enable the integration of domain specific concepts together with the mentioned requirements of handling knowledge and collaborations among systems at modeling level.

Running example: Because of the size of the smart city, several development processes and maintenance activities are applied for different kinds of systems. Therefore, there is a mixture of offline adaptation activities, such as software updates or the deployment of new functionalities, and online adaptation, such as appearing collaborations, behavioral changes due to changing requirements or hardware failures (R–18). Furthermore, the development patterns (R–19), which are discussed in [183], can be applied in the smart city example such as client-server communication or information sharing during collaborations. Finally, there are obviously different domains (R–20) involved. For example, there are cars, smart houses and intelligent traffic lights that are all engineered and developed from several companies following different development processes and concepts.

Analysis Requirements

R–21 Determining Causal Dependencies: Where the requirements R–01 to R–20 focus on the desired capabilities of the modeling language, the following requirements comprise the understanding and verification of the modeled SoS. The modeling of multiple feedback loops leads to an interleaving of adaptation effects. Therefore, the modeling language should support the understanding of the causal dependencies between adaptation activities, multiple feedback loops, and system interactions. Thus, the modeling language must provide clear concepts that describe causal dependencies between adaptation effects that can be analyzed or verified afterwards on the modeled SoS to investigate the overall emergent behavior.

R–22 Determining Knowledge Interdependencies: Beside the understanding of the causality between adaptation effects, activities and feedback loops influence each other over the manipulated and shared knowledge base. This requires a deep understanding of the knowledge distribution in the adaptive SoS, the coupling of knowledge via collaboration as well as the visibility and access to knowledge. Therefore, the modeling language must offer knowledge management concepts that facilitates the identification of used information in the modeled SoS as well as its contribution to the emergent behavior.

R–23 Static Analysis: Developing adaptive behavior and collaborations between MAPE activities requires verification, for instance, to ensure that an activity performing a certain role fulfills the behavior as defined by the interaction. This can be considered as consistency between the behavior of an activity and the behavior defined for a role in the coordination. Thus, when implementing or even executing coordinating activities, means to ensure that the roles and protocols are properly realized as required. The modeling language should enable static analysis for the modeled adaptive behavior of the SoS. Therefore, the modeling language concepts must have a clear semantic to investigate the designed adaptation and collaboration effects.

R-24 Runtime Analysis: Static analysis investigates and checks the modeled system behavior at development time. As motivated for the emergence characteristic (C-3.1) of an adaptive SoS, the combined, overall behavior of single systems and system interactions is considered as nonlinear and thus, the emergent behavior is hard to predict at development time. Therefore, a modeling language should enable the analysis of the emergent behavior at runtime, which causes the need of runtime analysis capabilities as well as the ability of checking system constraints during the lifetime of the SoS. Additionally, runtime analysis implies realistic runtime introspection of the adaptive SoS for retrieving desired information about the inner system state. Thus, the realizing framework of the modeling language should support monitoring capabilities, which further enables runtime analysis and the checking of runtime constraints.

R-25* Execution and Simulation: After modeling the adaptive SoS behavior, guidelines for supporting the implementation should be provided. One approach could be to use model-driven techniques such as code generation to create initial artifacts to start implementation. Another approach could be the interpretation (direct execution) of the developed models. This facilitates the analysis and reuse of existing models at runtime without the generation of intermediate development artifacts, but requires a framework that is able to directly execute the specified models. Furthermore, an execution of single modeled systems within the SoS enables a simulation of the joint system interactions by subsequently executing the modeled behavior over time, which leads to the overall emergent adaptive SoS behavior. Therefore, the realization framework of the modeling language should support the execution of models as well as their simulation, which complements the analysis of the modeled SoS towards the verification and understanding of the emergent behavior.

Running example: As derived in Section 3.1, an adaptive SoS is characterized by an emergent behavior (C-3.1). Therefore, all kind of causal dependencies (R-21) between the adaptive systems and the knowledge distribution (R-22) must be considered to predict and to analyze the overall SoS behavior at development time (R-23). Furthermore, a smart city must be able to introspect itself and its containing systems to reason about the current state and possible adaptations in the future, which leads to the need of runtime analysis capabilities (R-24). Finally, because the emergent behavior of the SoS is too complex for complete state space verification, simulation helps understanding the modeled system and interleaving adaptation effects, which requires a support for executing the modeled system (R-25).

3.3. State of the Art

This section gives an overview about the state of the art in modeling adaptive SoS according to the derived requirements above. Inspiring approaches are outlined that foster the idea of designing a new modeling language. Furthermore, gaps and open challenges are mentioned. Existing approaches for distributed self-adaptive systems just present specific solutions for particular problems of such systems or they provide architectural frameworks supporting the implementation. All of them do not explicitly specify how the distributed feedback loops are coordinated. However, such specifications are essential for an engineering approach.

There exists a lot of work modeling single feedback loops in control engineering in the embedded, automotive and robotic domain. For example, Kokar et al. [117] provide different

feedback loop designs for embedded systems as well as discuss pros and cons of each variant based on the underlying control theory. Burns [54] comprehensively discusses the formal, mathematical background designing a control loop by considering different real world problems such as timing and uncertainty. The advantage of this formal background in control engineering guarantees specific characteristics of the adaptive behavior such as stability or a fast reaction time. However, adaptive behavior in complex systems often considers dynamic software architectures [134], which cannot be realized by one static, predefined control loop and opens new challenges in software engineering [183].

From the control engineering, feedback loops become a first class concept in the design of software systems as for example in the context of the MDE. In the MDE general purpose modeling languages are used to describe the structural and behavioral part of the software system. From the system engineering perspective, the SysML [89] or Architecture Description Languages offer powerful concepts for the specification of the system architecture, its sub-components, and distribution aspects (R-05). For the software engineering perspective, the UML [87] and SoaML [88] provide basic concepts for the modeling of collaborations (R-07) and interaction behavior (R-08). Because of the generality of these modeling languages, they do not consider specifics of adaptive SoS with explicit feedback loop modeling (R-01), the representation of knowledge as runtime models (R-09), and the coupling of feedback loops with collaborations (R-03). Hebig et al. [96] present an UML profile for the explicit modeling of feedback loops in form of UML components, which introduces a semantical notion of feedback loops into the general UML concepts. However, the internals of the feedback loop behavior (R-01) are encapsulated and therefore hidden in the component specification. Furthermore, the interaction between feedback loops is modeled via abstract component interfaces, whereas the concrete protocol (R-08) is not modeled.

From the autonomic computing domain, Kephart et al. [110] introduce the MAPE feedback loop, which consists of four, predefined adaptation activities. Therefore, the internals of the feedback loop become visible (R-01), but are statically arranged. Brun et al. [49] take the idea of the MAPE feedback loop approach and propose different designs by transferring typical control engineering patterns to a software engineering level. Although there are different variants of controller proposed, the overall feedback loop remains static according to the design decision. Simultaneously, Salehie et al. [156] outline different variants of separating or integrating the adaptation logic into the domain logic and propose a hierarchy of self-* capabilities for adaptive software systems. Thereby, the role of the knowledge (R-09) within the adaptation logic is not defined. Moreover, there is often one feedback loop designed for realizing a single self-* capability. Therefore, the specification of multiple (R-03), interacting (R-04), and possibly distributed (R-05) feedback loops are not considered.

Other approaches as for example Fleurey et al. [73] use domain specific modeling languages for the specification of valid adaptation behavior. Afterwards, a given framework validates the modeled configuration and generates an appropriate adaptation logic. Similar to a domain specific modeling language, Gui et al. [92] provide a standardized component model, which encapsulates the adaptation strategy. Thereby, components can be connected, which forms the overall adaptation logic. A framework uses the standardized component model to generate an appropriate adaptation behavior. From a system engineering perspective, Morrison et al. [139] propose the *ArchWare* ADL for the offline specification of an evolving system architecture. The ADL description is used to generate an adapted system implementation according to the modeled evolution step. Furthermore, Morin et al. [138] generate configuration scripts on the fly to change the system, but they have no explicit feedback loop description. Cetina et al. [57]

present a smart home scenario, where a predefined single feedback loop uses a framework to generate code during system operation, which adapts the behavior according to given user requirements. The key aspect of these approaches is a predefined set of building blocks (e.g., the component model or the domain specific modeling language concepts) to ease the feedback loop realization by supporting code generation. However, the adaptive behavior is generated, whereas the implementation details are hidden in the corresponding framework and therefore not explicitly modeled (R-01). Furthermore, the models are used for validation and generation purpose at development time and are not kept alive at runtime (R-09).

Multiple feedback loops arise, if different concerns or systems must be adapted, whereas each feedback loop focuses on one specific self-* capability. In such case, existing approaches propose one specific solution that tailors the underlying problem. For example, a completely decentralized use case of a self-organizing multi-agent system is described by Klein et al. [114]. In this example, an audio streaming system must apply a sequence of different filter operations on audio packets, which are performed by autonomous agents, before the packet can be provided to the client. The goal of the audio streaming system is the minimization of packet losses by maximizing availability and reliability of successfully processed audio packets. A distributed self-adaptive scenario of a mobile learning application is described in [84], where different tasks are distributed to different student groups. In contrast to the decentralized scenario in [114], the described approach of Iglesia et al. [84] uses a centralized master-slave coordination pattern between mobile clients and the server. However, there are much more specific scenarios, where multiple feedback loops are defined and interact with other feedback loops in a predefined, specific system solution (cf. the comprehensive description of scenarios in own former work in [11]). Nevertheless, those specific system solutions do not focus on providing modeling language concepts to explicitly describe the feedback loop (R-01) and their interaction (R-03), although they pinpoint to a specific realization for the corresponding problem.

Moreover, Weyns et al. [182] present a reference model called *FORMS* for the formal specification of distributed feedback loops. Thereby the focus is on the validation of the specified feedback loop behavior at development time to predict the overall emergent adaptation behavior of the system as well as to support the implementation. However, the formal models are not used at runtime (R-09) and the feedback loop interactions (R-03) are not modeled as first class entities, but rather are hidden in the formalism. The formal approach of verifying feedback loop interaction is further refined by Iglesia et al. [84] and Iftikhar et al. [103] towards an offline and online verification of the modeled feedback loop interaction. In the context of specific solutions for multiple feedback loop interactions, Alvares de Oliveira et al. [15] present a specific synchronization protocol and show its applicability in a cloud computing environment. Moreover, Malek et al. [131] present an approach that solves the redeployment problem of feedback loops and Sykes et al. [164] show an application of a distributed self-assembly scenario. However, all of these approaches focus on a concrete protocol implementation to solve the underlying problem. The feedback loop interactions are not modeled as first class entities (R-03), the delegation of tasks is not supported (R-06) and the role of runtime models is not described (R-09).

A more precise focus on runtime models is given by Blair et al. [38], who introduce the general concept in the context of self-adaptive systems. Challenges are outlined by Weyns et al. [181], who emphasize the demand on an engineering approach that captures partial knowledge and uncertainty within the adaptive system. Assmann et al. [19] propose a three layer reference architecture to cope with runtime information from different system types such as embedded

and cyber-physical systems. However, the reference architecture is more a first sketch of integrating adaptation behavior and runtime models on different layers. The semantic (R-10) and partial runtime models (R-11) are not considered. Götz et al. [86] target the point of partial available knowledge and illustrate the definition of views via a model query language, but lack in a clear semantic and integration of the runtime model information into an overall control architecture. Furthermore, Rajhans et al. [151] focus on the integration of multiple, different, potentially partial domain models during the development of a CPS, but lack in an integration concept of this information at runtime. In general, the mentioned approaches show the potential of runtime models and their application for specific use cases. Unfortunately, the integration of this information into an explicit feedback loop modeling (R-01) is missing. Moreover, the distribution of knowledge (R-05) during feedback loop interaction is not considered.

There are several layered architecture approaches together with an underlying framework or middleware, which provide a clear separation of adaptation concerns. For example, Baresi et al. [25] propose the *SeSaMe* middleware that consists of a component and management layer. The middleware facilitates the modeling of collaboration aspects, but focuses on the definition of black box components instead of feedback loops. The *RAINBOW* framework from Garlan et al. [79] also realizes a two layer architecture, where a single control loop monitors and changes the underlying system below. Similar to the *SeSaMe* middleware, Bures et al. [50] propose the *DEECo* component model. These components can dynamically build so-called ensembles, which are dynamic collaborations. The underlying runtime framework realizes an appropriate scheduling and communication mechanism for data exchange. Kramer et al. [119] propose a generic three layered architecture with a predefined task focus on each layer. On the highest layer, long term goals are handled, whereas the middle layer realizes short term adaptation concerns of the underlying software component architecture. This three layer proposal is adopted by many other frameworks. For example, Edward et al. [70] propose a three layer architecture that is further refined by Tajalli et al. [166] in the *PLASMA* architecture. The highest layer generates plans for the adaptation logic at the middle layer, whereas the middle layer adapts the application logic accordingly. However, the focus is on the adaptation plan generation and internals of the feedback loop as well as their interaction are not modeled. Moreover, the architecture is predefined and extensions or changes in the proposed approach are not discussed. Also previous work [53, 100] proposes the *Mechatronic UML* approach that extends the UML to specify and generate a hierarchical architecture. The approach addresses distinct feedback loops for reconfiguration and planning purposes. However, the adaptation logic is defined during development and cannot be changed during runtime nor is kept alive as runtime models.

Finally, Weyns et al. [183] introduce a set of common patterns for distributed feedback loops such as *master-slave* or *regional planning*. However, they do not address the distribution (R-05) and modeling of knowledge (R-09) nor describe insides of the explicit feedback loop structure (R-01). They end up with challenges of the explicit modeling of collaboration aspects (R-03, R-07) as well as the specification of partial knowledge (R-11). With focus on offline and online adaptation steps within the software engineering process, Andersson et al. [16] comprehensively discuss the problem but do not present a working solution that integrates both processes.

In summary, state of the art approaches provide clear solutions that aim at reducing the development effort to specify the desired adaptation logic. Often, the adaptive behavior is designed in a single feedback loop, whereas the adaptation effects are well understood

with respect to the experience from the control engineering domain. For those approaches, an additional framework usually provides a predefined structure of adaptation steps (e.g., MAPE), where concrete algorithms can be exchanged to realize the desired self-* capability and an appropriate realization is generated for the target platform. Therefore, the adaptation logic remains hidden in the used framework, which limits the support of explicit modeling of individual feedback loops (R-01) and the handling of available knowledge in form of runtime models (R-09). Furthermore, single feedback loops limit the overall application for multiple adaptation concerns, but avoid arising synchronization problems and interactions between multiple feedback loops. In general, there exists only preliminary work of modeling, predicting and handling multiple feedback loops as well as their interaction at runtime. As a consequence, existing approaches offer a predefined layered architecture with well known, but fix components in it. If the individual specification of the adaptation logic is supported (R-01), the approaches do not consider the integration of runtime information (R-09) into the feedback loops or simplify the interaction by predefined interfaces (R-13). In contrast, approaches that focus on runtime models often do not consider their partial usage and their distribution among multiple feedback loops.

Own former work [11, 175] already identifies those gaps and proposes the Eurema modeling language, which is the predecessor modeling language of this thesis. Eurema aims at the explicit modeling of feedback loops (R-01) in a layered adaptive architecture. Thereby, multiple feedback loops are considered and the available knowledge (R-09) is integrated into the adaptation activities of the feedback loop. Therefore, Eurema is suitable for defining the adaptation capabilities of a system and the hierarchical dependencies of those on the layered adaptation engine. However, Eurema is developed with the focus on single self-adaptive systems, but does not aim for the modeling of an adaptive SoS. As a consequence, Eurema lacks in specifying concurrent, distributed feedback loops (R-05) and their interaction by means of collaborations (R-07, R-08). Therefore, the use of partial knowledge (R-11) and different communication mechanisms (R-13) between feedback loops as typical in an adaptive SoS are not considered. Furthermore, Eurema does not discuss possible verification capabilities of the explicitly described adaptation logic. Thus, state of the art approaches do not consider all requirements for the modeling of an adaptive SoS architecture as derived in Section 3.2. The experience of the Eurema modeling language and its limitations lead to the modeling approach of this thesis. An overview about the new modeling language concepts is given in the next chapter.

4. Overview

This chapter gives an overview of the Distributed Eurema with Collaborations (Deurema) modeling language approach. Thereby, it considers the outlined goals of this thesis discussed in Chapter 1 as well as the modeling language requirements from Chapter 3 and pinpoints to the corresponding Deurema concepts. Furthermore, modeling an adaptive SoS with the Deurema approach enables new possibilities with respect to the analysis, simulation and realization of the modeled system solutions for different, application specific domains.

Figure 4.1 gives an overview of the Deurema concepts according to the goals of this thesis. This chapter starts with the modeling concepts of the Deurema language. At first, Deurema explicitly considers the adaptation activities realizing feedback loops as first class language entities that describe the overall adaptation behavior of the system. Thereby, feedback loops realize different self-* capabilities as for example self-configuring or self-healing as shown on the left in the middle layer in Figure 4.1. The explicit modeled adaptation logic operates on application specific behavior, which can be modeled in Deurema following a component-based and rule-based development approach. Second, Deurema considers the available knowledge in the adaptive system in form of runtime models (cf. MART approach introduced in the preliminaries in Section 2.2.3). On the one hand, the runtime models are considered as first class entities and thus, can be seamlessly integrated into the modeling concepts of specifying the adaptation logic. Therefore, a fine-grain access can be modeled on available runtime models by adaptation activities. On the other hand, Deurema supports a classification of runtime models that defines the purpose of the model and gives information about the contained content. Third, the support for multiple, possibly distributed feedback loops raises needs of determining the influence of feedback loop interactions, which is considered by the Deurema collaboration concept. In summary, the modeling with the Deurema language comprises the adaptive behavior, the available knowledge and system collaborations.

The explicit modeling of the adaptation logic together with the available knowledge and the feedback loop interactions is a major step of describing the overall adaptive, emerged SoS behavior. On basis of the Deurema models, this thesis discusses two further research questions targeting the analysis and simulation of the modeled systems as well as its realization in a specific domain. First, static analysis techniques and runtime analysis can be applied, whereas both techniques are used to retrieve metrics, (anti-)patterns used in the Deurema models, or evaluate simulation runs as shown on the top in Figure 4.1. Static analysis is typically used during the development time of the system, whereas runtime analysis can be applied during the lifetime of the adaptive SoS. Answering the research question concerning the realization of the modeled behavior must consider state of the art standards and development approaches, which differ according to the application domain. This thesis discusses the realization of Deurema models in detail by considering one de facto standard from the automotive domain, but also pinpoints to other common frameworks and standards.

Beside the analysis, the overall emergent SoS behavior can be further investigated by model as well as system simulations. Simulation runs can be applied on different levels to verify that the modeled system behavior corresponds to the current goals of the adaptive SoS. Thus, the simulation capabilities of the Deurema language are a crosscutting concern

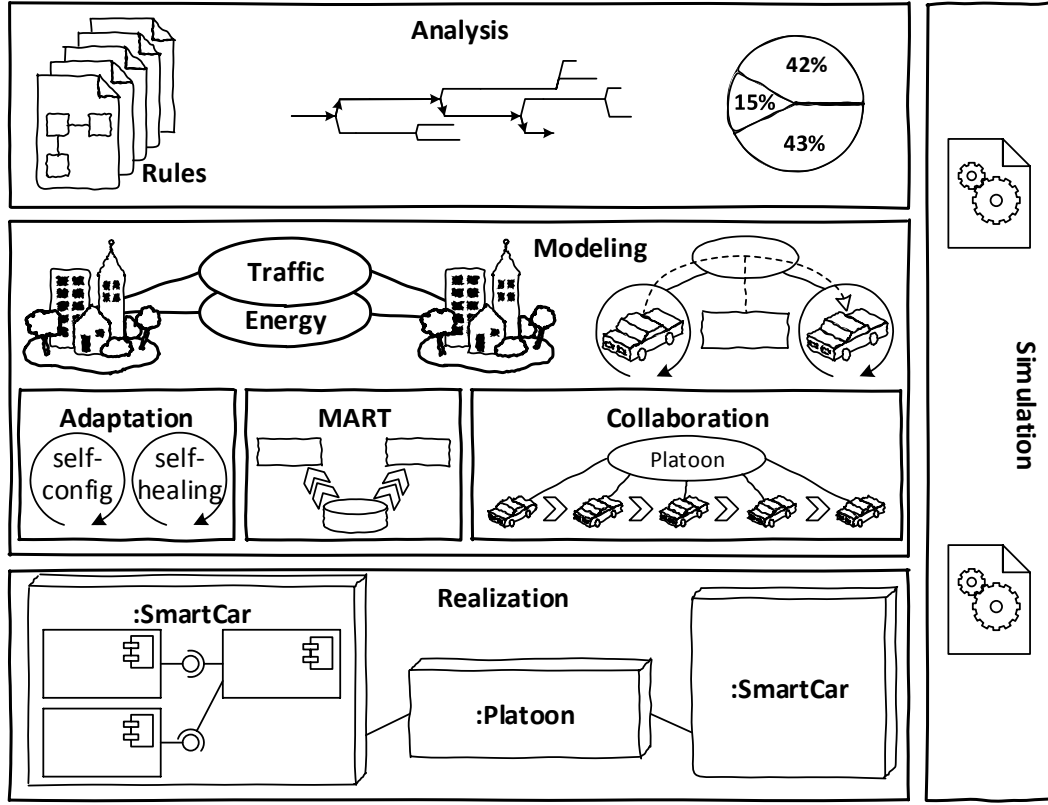


Figure 4.1: Overview

applied during modeling, analysis and realization of the adaptive SoS as show at the right in Figure 4.1. For example, a model-based simulation may be applied very early during the development of the adaptation logic, which requires a clear semantic of the Deurema models that enables their execution. Even an early simulation of the models can pinpoint to collaboration problems between different feedback loops or show that the overall adaptation effect differs from the expectations. In the context of this thesis, a Deurema model interpreter is presented that facilitates the execution of the specified adaptation logic. However, early model-based simulation abstracts from device specific runtime phenomena as for example limited computation power or available memory resources. During the realization, the simulation runs can be refined to cope with domain specific problems such as time or uncertainty. After a complete mapping of Deurema models to a concrete realization, domain specific verification techniques can be used to ensure that the overall adaptive behavior still corresponds to the modeled system solution. During the analysis, simulation supports the execution of different static analysis metrics as well as the parallel execution of the adaptation logic together with runtime analysis rules. Furthermore, logged simulation runs can be used again for analysis purposes, which might lead to changes in the modeled adaptive SoS architecture improving the overall system design. Thus, modeling, analysis and simulation can be successively and periodically used during the development process as well as during the lifetime of the adaptive SoS to verify the modeled behavior against given goals and constraints.

In the following, each perspective of modeling, analysis, simulation, and realization of Deurema models is discussed in detail. Thereby, the modeling language requirements in Chapter 3 lead to design decisions for the Deurema approach, which are outlined, too.

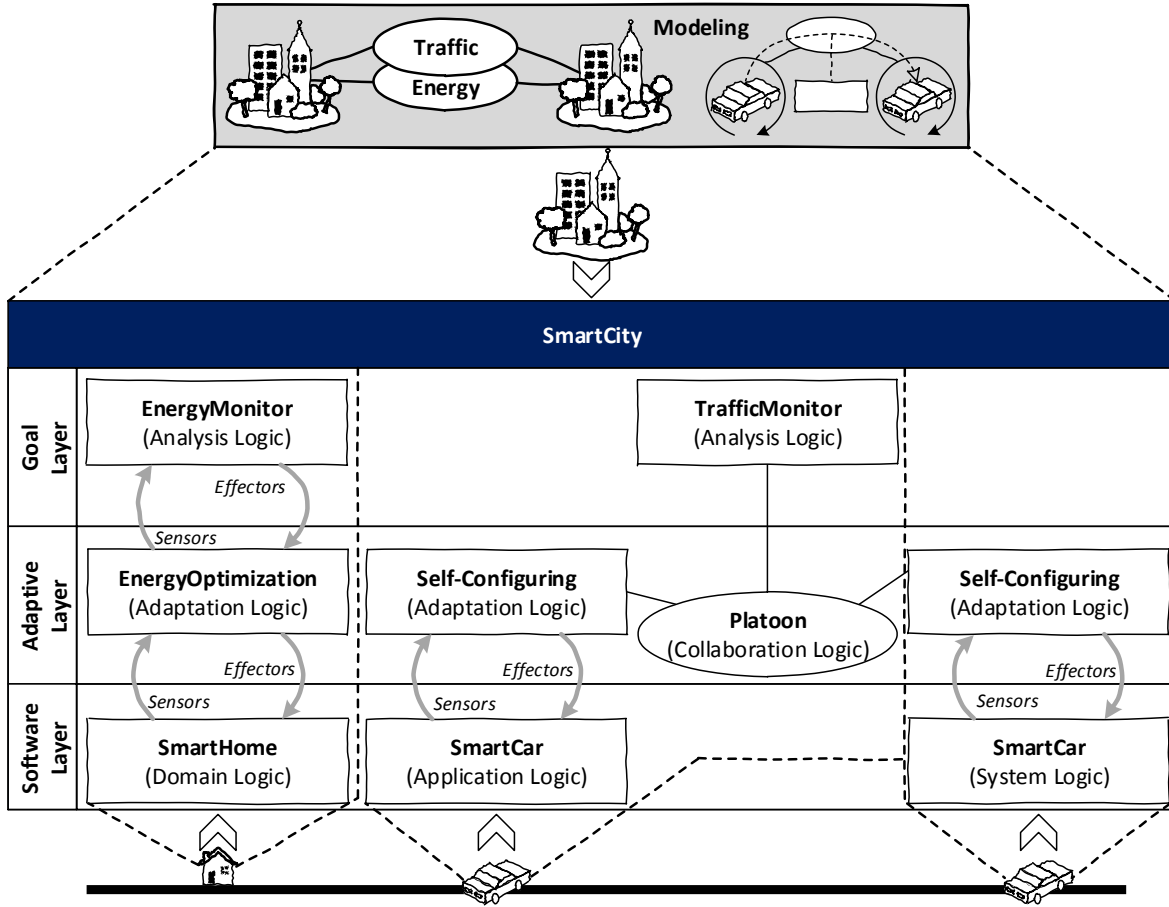


Figure 4.2: Smart city running example: Deurema adaptive SoS modeling

4.1. Deurema Modeling Language

The Deurema language focuses on different aspects modeling the adaptive SoS. First, a SoS comprises several, diverse, independent systems. According to the running example, Figure 4.2 sketches this diversity by showing a smart city system comprising one smart home and two smart car systems. The dashed line in the figure denotes the system borders. In the example, the smart home system contains further behavior. The domain specific behavior, denoted as Domain Logic in the example, realizes the smart home functionality such as closing the windows in the case of bad weather conditions, starting a surveillance service if the user is not home, or providing a lighting service according to given user requirements. On top of the domain logic, an energy optimization adaptive behavior, denoted as Adaptation Logic, improve the smart home behavior with respect to the energy consumption. Again, the optimization is supervised by an energy monitor, denoted as Analysis Logic, which determines the energy optimization of the smart home over time and can be for example used by the smart city to improve the overall power supply. The local behavior of the smart home is placed on different layers in the SoS architecture. Therefore, the energy optimization logic can sense the underlying domain specific behavior and may change (effect) it during the optimization. Additionally to the smart home, the both smart cars in Figure 4.2 also realize individual system behavior such as a self-configuring adaptation logic. Furthermore, they collaborate with each other, denoted

by the platoon Collaboration Logic modeled as ellipses in the figure, which enables a system interaction crossing the boundaries of one system. As a consequence, all contained systems in the smart city example and in particular the collaboration between system entities contribute to the overall emergent adaptive SoS behavior, which is captured by Deurema.

Second, the Deurema approach covers the modeling of adaptation activities (the Adaptation Logic in the example) in form of feedback loops. Those feedback loops operate on local behavior as sketched in Figure 4.2, where the adaptation logic runs on top of domain, application, and system specific logic. Thus, Deurema explicitly considers different variants of local behavior modeling.

Third, the Deurema approach refines the notion of knowledge in the adaptive SoS using runtime models that can be seamlessly integrated into the local and collaboration behavior.

Fourth, Deurema explicitly determines collaboration aspects between systems inside an adaptive SoS. As motivated above, collaborations enable new functionalities that leaves the border of local system behavior. In the example, cars can join into a platoon, e. g., for autonomous driving, which contributes to the overall emergent SoS. Furthermore, the smart city can monitor the platoon collaborations and may realize further traffic flow optimizing goals, which cannot be reached by one individual car system alone. Another effect is that the beforehand local adaptive behaviors also emerge over the collaborations, which can be described by Deurema, too.

4.1.1. Modeling the Adaptation Logic

Figure 4.3 gives an overview of the Deurema aspects with respect to modeling the adaptation logic of an adaptive SoS. Therefore, Deurema introduces adaptation activities as first class entities (R-01) that form a feedback loop. Thus, the adaptation behavior of the adaptive SoS is explicitly captured. As shown in the Self-Configuring feedback loop in the picture, the intra-loop coordination of adaptation activities (R-02) is defined by the control flow similar to UML activity diagrams.

In the example in Figure 4.3, the feedback loop consists of four, subsequently executed adaptation activities, namely Update, CheckTrafficSituation, Optimize, and Effect. The aim of the feedback loop is the enabling of self-configuration capabilities of a smart car depending on the current traffic situation and position of the car (e. g., city, countryside). For example, the smart car may switch in an eco-friendly city mode, if the driver of the car enters a city. If multiple cars follow this eco-friendly driving style, the emergent behavior can lead to a reduction of the overall pollution within the smart city. However, each single activity in the feedback loop performs its action according a common understanding of the current situation of the system, which is determined in a knowledge base.

Beside the concept of feedback loop modeling, Deurema considers the integration of modeled adaptive behavior into different domains (R-20) by supporting component-based as well as graph-based modeling approaches, which is shown at the bottom in Figure 4.3. Because an adaptive SoS consists, among others, of embedded and cyber-physical systems, the component-based development approach is the dominant development paradigm for robotic [47, 48] and automotive [62, 105] systems in that domain. Additionally, own former work in [14] comprehensively discusses a toolchain for state of the art embedded systems following the component-based paradigm. Consequently, Deurema adopts this paradigm for the integration of feedback loops with component-based behavior modeling. The example on the left in Figure 4.3 shows an excerpt of a smart car application logic that is realized by three components Navigation, SensorFusion, and Driving. Similar to feedback loop activities, components operate on

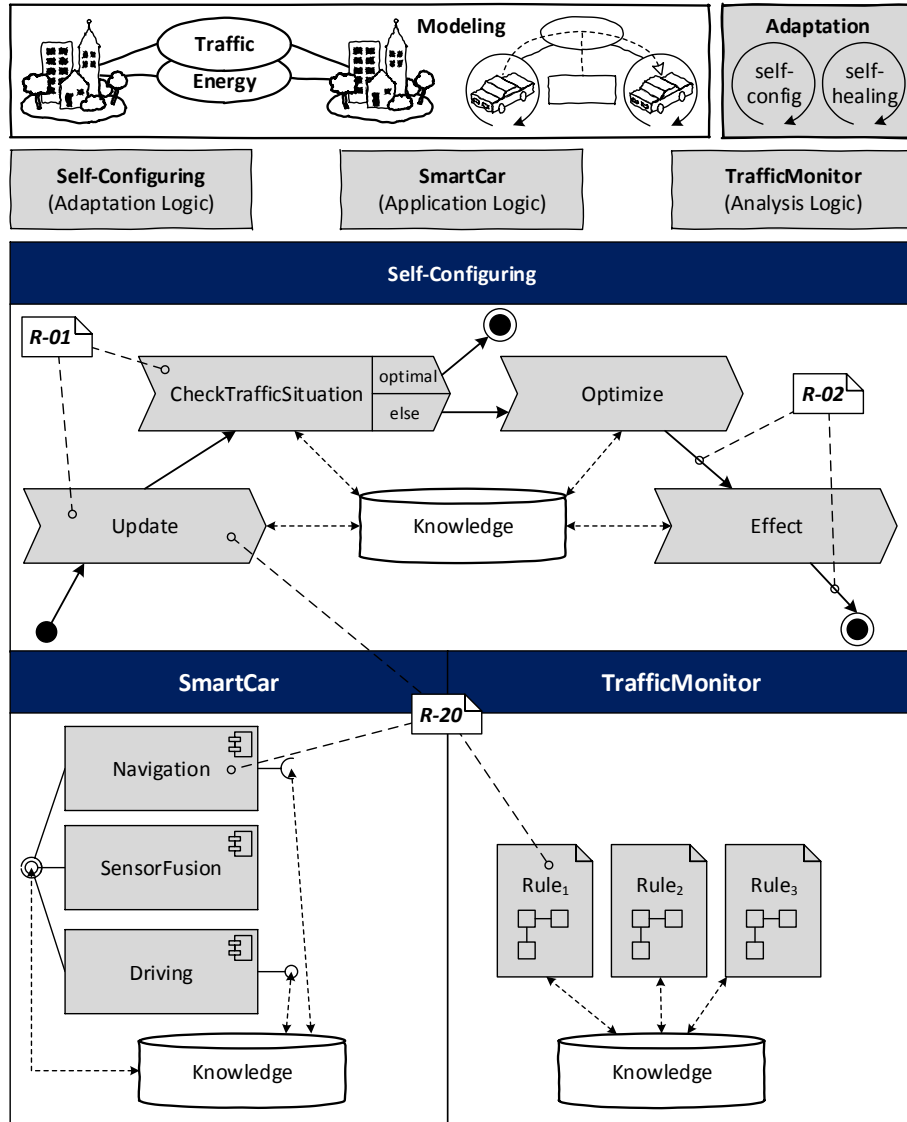


Figure 4.3: Modeling the adaptation logic in Deurema

a common knowledge base for realizing the desired functionality. The knowledge is accessed and provided via ports, which is indicated by arrows in the figure.

The rule-based modeling support of Deurema is motivated by the fact that almost all models in the MDE approach can be represented as graphs. Thereby, model types can be different as for example structural architecture models in form of class or component diagrams as well as behavioral models, which can be activity diagrams, petri nets, or automata. On an abstract level, all of these models can be represented, manipulated, and stored as graphs. Thus, graph transformation rules are a powerful concept for monitoring and changing the models. For example, a graph transformation rule can identify a situation of interest in a model, e.g., an architectural configuration, and change the found match in the model (graph) afterwards. This change will lead to a derived model, which can be another architectural configuration of the system. Graph transformation rules together with the basic concepts of matching and applying the side effect on a graph structure are already introduced in the preliminaries in

Section 2.2.5 and are adopted in Deurema. Thus, Deurema exploits the underlying graph structure of models and provides a first class concept of defining graph transformation rules, which work on arbitrary domain models. The example on the bottom at the right in Figure 4.3 shows three graph transformation rules of a *TrafficMonitor*. A possible application of these rules in the running example could be a set of declarative rules that look for specific traffic situations (e. g., traffic jams). If an observed situation occurs, a corresponding adaptation effect can be triggered. Graph transformation rules work on the same common knowledge base as feedback loop activities and components, indicated by the arrows in the figure.

Supporting the three concepts of modeling the local behavior of systems within the adaptive SoS by means of feedback loops, components or graph transformation rules opens a broad range of application possibilities for the Deurema modeling language.

4.1.2. Knowledge as Runtime Models

As emphasized in the former section, the modeled local behavior and the adaptation logic operates on a common knowledge base. As shown in Figure 4.4, the notion of knowledge is further refined by storing information of interest in runtime models (R-09), whereas the general MART concept is already introduced in Section 2.2.3. First, the Deurema modeling language provides a runtime model categorization, which clarifies the purpose of a runtime model and thus, provides insights into the contained runtime information. Second, Deurema supports arbitrary metamodels from different domains that describe the concepts of a runtime model. Altogether, the purpose, the metamodel, and the Deurema management of the runtime model inside the adaptive SoS define the semantic of the runtime information as demanded by the modeling language requirement R-10.

Deurema seamlessly integrates the runtime models (R-20) in all three possibilities of modeling the local application and adaptation logic as discussed in the former section. Furthermore, it supports the modeling of partial knowledge in arbitrary runtime model views (R-11). This partial information can be enriched via system collaboration (R-03) as well as monitoring the system context (context-awareness) or system inner structure (self-awareness). Additionally, model operations (modeled as dashed arrows in Figure 4.4) explicitly define the access, the amount, and the direction of runtime model usage, which further contributes to a clear semantic (R-10) and partial knowledge usage (R-11).

The examples in Figure 4.4 describing the *Self-Configuration* feedback loop, the *SmartCar* application logic, and the *TrafficMonitor* behavior as introduced in the former section show how the runtime models (modeled as rectangles in concrete syntax) can be used across the Deurema domain modeling concepts. For example, the *Architecture* runtime model contains information about the inner structure of the adaptive SoS. This information is spread in different views into the feedback loop, smart car, and traffic monitor, where it appears as runtime model. Each system part uses the architectural model differently by defining local model operations on the runtime model to retrieve or annotate information. The Deurema execution environment is responsible for the maintenance of different runtime model views, resolution of concurrent accesses, and guaranteeing modeled visibility restrictions between systems insight the adaptive SoS.

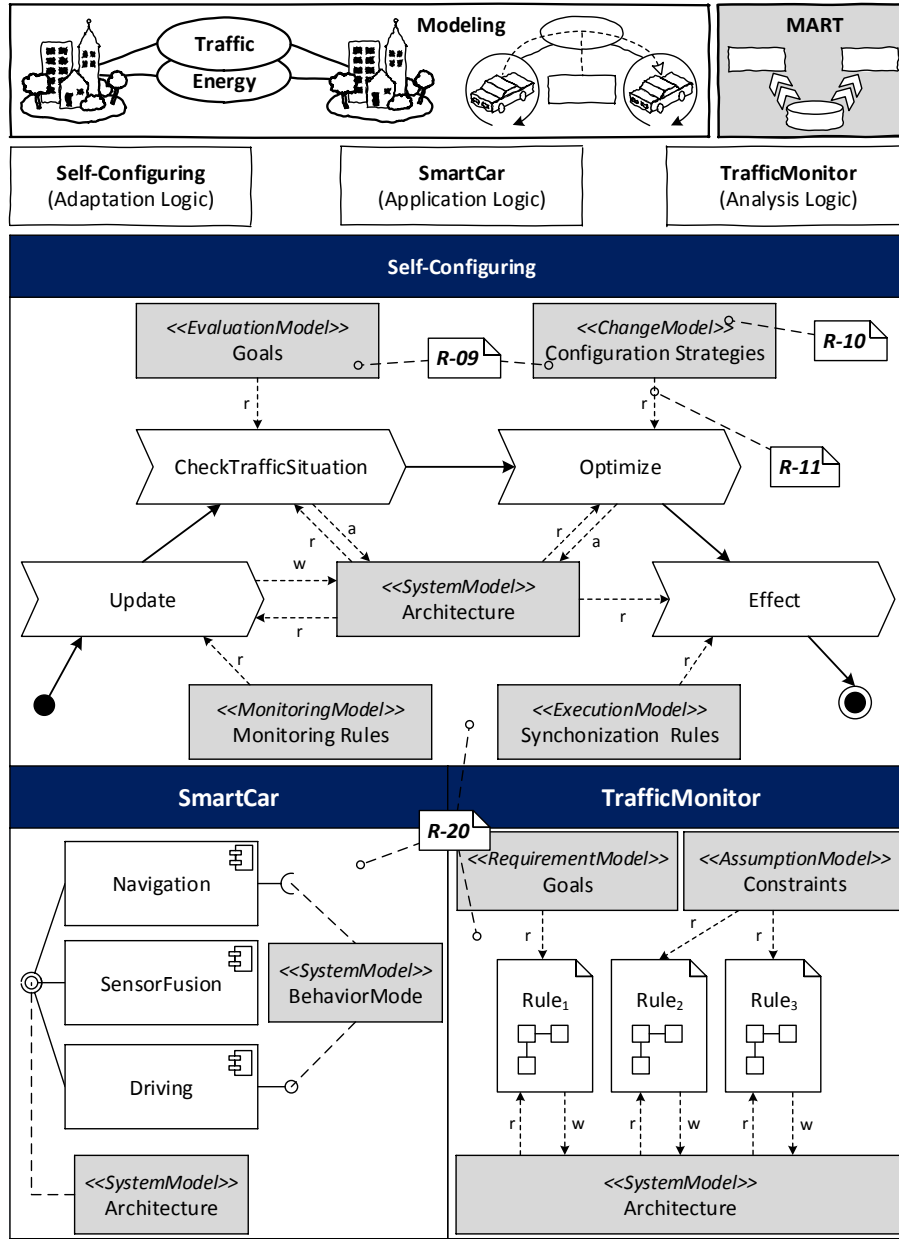


Figure 4.4: Deurema knowledge as runtime models

4.1.3. Modeling Collaborations

An important key aspect in Deurema is the modeling of system interactions (R-03) in form of collaborations. Thus, the modeling language is designed to support those collaborations as first class entities as visualized in Figure 4.5. Thereby, Deurema fosters separation of concerns by explicitly separate collaboration related behavior from the local adaptation activities in the system. A Deurema collaboration defines abstract roles (R-07), whereas each role follows an interaction protocol (R-08). For the communication between collaborating roles, Deurema provides different communication mechanisms (R-13), e.g., for invoking remote services (R-06), transferring knowledge (R-12), or synchronizing the behavior (R-14) between

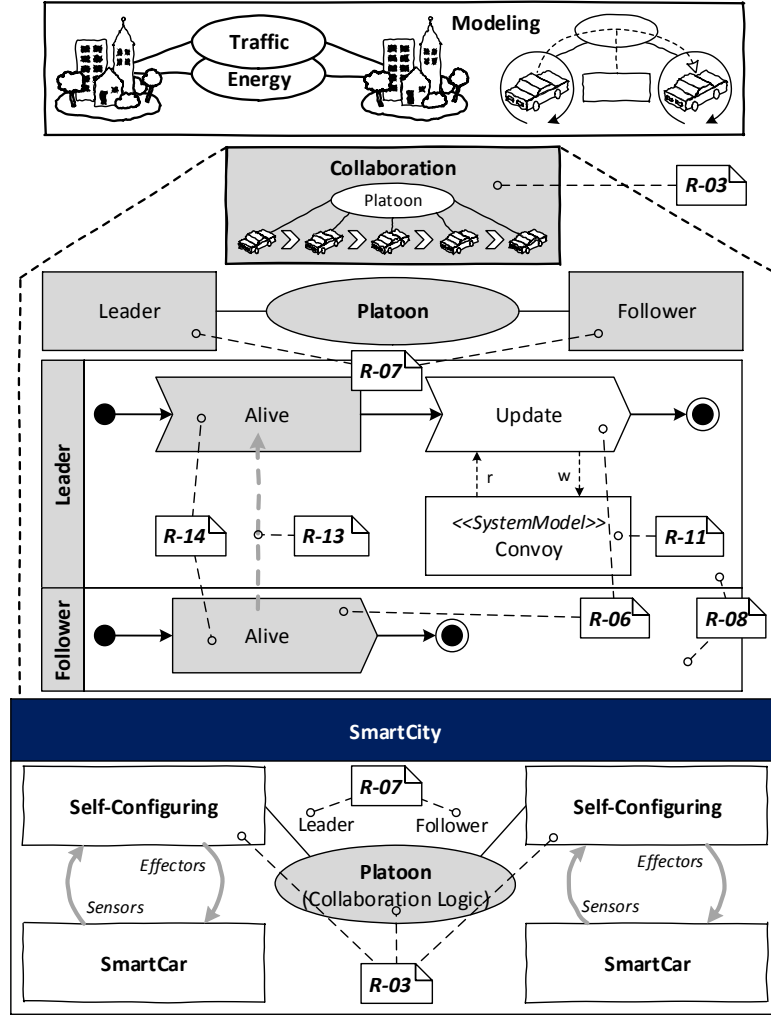


Figure 4.5: Deurema system interactions via collaborations

roles. Considering the running example, Figure 4.5 shows a platoon collaboration between smart cars with the two roles Leader and Follower. Furthermore, the figure exemplarily depicts by the two activity lanes that each role follows a specified protocol. Beside the sending of synchronization messages between roles, the protocol integrates the Deurema runtime model concept for representing the available knowledge and performs local adaptation activities. In the example in Figure 4.5, a local Update activity manipulates a Convoy runtime model, which is only visible in the context of the Leader role.

Defining the collaboration aspects separately from the local adaptation behavior supports the distinct development and analysis of both. However, because collaborations are the major reason of the emergent SoS behavior, Deurema provides additional concepts to integrate the specified interactions into the local adaptive system behavior. A snapshot for an integration example is shown at the bottom of Figure 4.5, where two smart cars with a self-configuring feedback loop participate in the platoon collaboration. Following the well-defined collaboration integration concepts of Deurema enables analysis (R-23, R-24) and simulation (R-25) of interactive behavior together with local adaptation effects. As a consequence, the adaptive behavior can be separately modeled and analyzed first and integrated into a collaboration

context afterwards. Transferred to the running example, the behavior of each smart car can be separately investigated before verifying the interplay of several smart cars within a platoon.

4.1.4. Modeling the Adaptive SoS Architecture

Bringing the Deurema modeling concepts together leads to the possibility of specifying local adaptation behavior by considering an explicit modeling of feedback loops (R-01) together with the intra-loop coordination (R-02). Furthermore, the notion of knowledge is refined and can be used locally for different domains as well as in collaboration protocols. Deurema integrates all these single specified parts towards a description of the adaptive SoS architecture as depicted in Figure 4.6. Typically, an adaptive SoS architecture consists of multiple layers as exemplarily visualized for the SmartCity in the figure. Therefore, Deurema considers different layers, whereas each single modeling concept as described above can be reused assembling the adaptive SoS architecture. Thus, the smart city example contains multiple instances of the Self-Configuring feedback loop and SmartCar component architecture as described above.

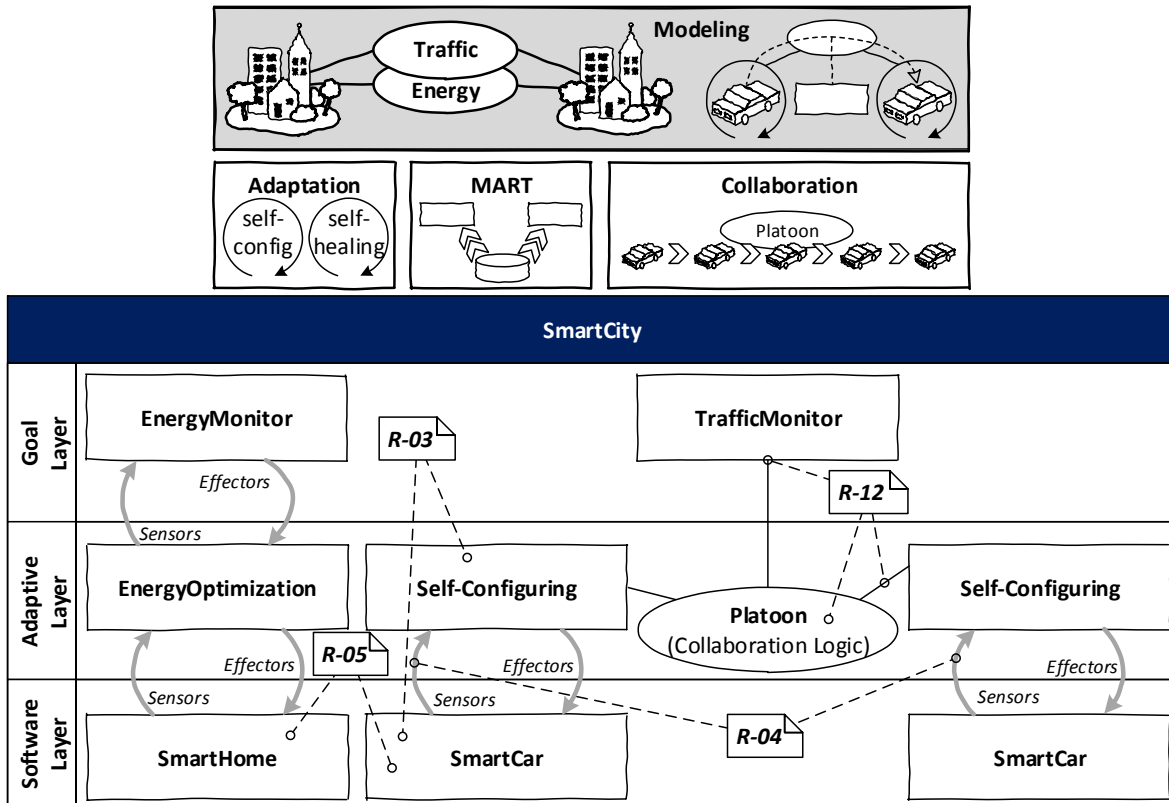


Figure 4.6: Deurema adaptive SoS architecture

Furthermore, the inter-loop coordination (R-03) can be described as shown between each pair of a smart car together with its self-configuring feedback loop, whereas the former triggers (R-04) the latter. Another possibility of describing inter-loop dependencies is the Deurema collaboration concept. In the example, there are three participants for the Platoon collaboration that are the beforehand discussed TrafficMonitor (modeled using the graph transformation rule concept) and two instances of the Self-Configuring feedback loop. The three collaborating system parts follow the specified collaboration protocol, which is the inter-loop coordination.

Thereby, systems may participate in multiple collaborations. In such cases, Deurema provides clear modeling concepts of separating the local adaptation behavior and assigning system interactions to a concrete role in one collaboration instance. As a consequence, although systems may participate in multiple collaborations, each behavioral aspect of the system can be recognized as local behavior or contribute to a concrete collaboration instance.

Additionally, the distribution of systems (R-05) is considered by placing several instances, such as smart cars or feedback loops, on different layers in the overall SoS architecture. The inter-loop coordination, together with the distribution and collaboration of system parts, define the emergent behavior of the adaptive SoS. Due to an increasing complexity of the complete SoS architecture, analysis and simulation support are required to verify that the emergent functionality behaves according to given goals and expectations.

4.2. Deurema Analysis

Presenting the Deurema modeling language is one contribution of this thesis. Additionally, this thesis discusses different possibilities of analyzing the adaptive SoS, which is modeled with the Deurema approach, towards an understanding of the adaptive behavior and verifying it against given goals. This analysis can be applied on different levels during the modeling (development) as well as at runtime of the adaptive SoS. Thus, there are the two dimensions of applying the analysis techniques, whereas this thesis discusses both and refers to the terms *static analysis* (R-23) and *runtime analysis* (R-24) respectively. Beside the application of the analysis techniques during the development or execution of the adaptive SoS, Deurema enables the investigation of adaptive behavior on different levels of granularity as outlined in Figure 4.7.

First, a fine-grain analysis of single adaptive system parts is supported, which can be a single feedback loop as the Self-Configuring example in Figure 4.4, the interplay of components as outlined for the SmartCar example in Figure 4.4, the effects of graph transformation rules as discussed for the TrafficMonitor in Figure 4.4, or the interplay inside a collaboration as shown for the Platoon example in Figure 4.5. Second, due to the emergence of those single solutions via inter-loop coordination such as triggering (R-04) or collaboration (R-03), this thesis presents analysis capabilities to investigate effect propagations of aggregated system behavior. An example for such an effect propagation are distinct feedback loops that interact with each other over a collaboration as shown in Figure 4.7 for the Platoon collaboration between the two Self-Configuring feedback loops and the TrafficMonitor. During the collaboration, knowledge is shared, which might influence the local adaptive behavior of the collaboration participants. Understanding the knowledge distribution in the overall adaptive SoS as well as its influence on local behavior is challenging (R-22), but supported by the Deurema analysis. Another example is the interplay between feedback loops and component-based specification of system parts as shown for each SmartCar, which runs a Self-Configuring feedback loop on top and forms a typical pattern of layered control. Therefore, the Deurema analysis offers rules to identify common architectural and interaction patterns (R-19) as well as to investigate the causal trigger dependencies between systems (R-21). The analysis of the effect propagation is not limited to pairs of systems, but rather can be transitively extended through the overall adaptive SoS. A typical verification scenario could be the testing of the collaboration protocol ensuring that the interactive behavior fits expectations. Afterwards, the collaboration can be integrated into the local adaptation behavior of a feedback loop to investigate, if the interplay between local activities and collaborative behavior still works as expected. Additionally, the

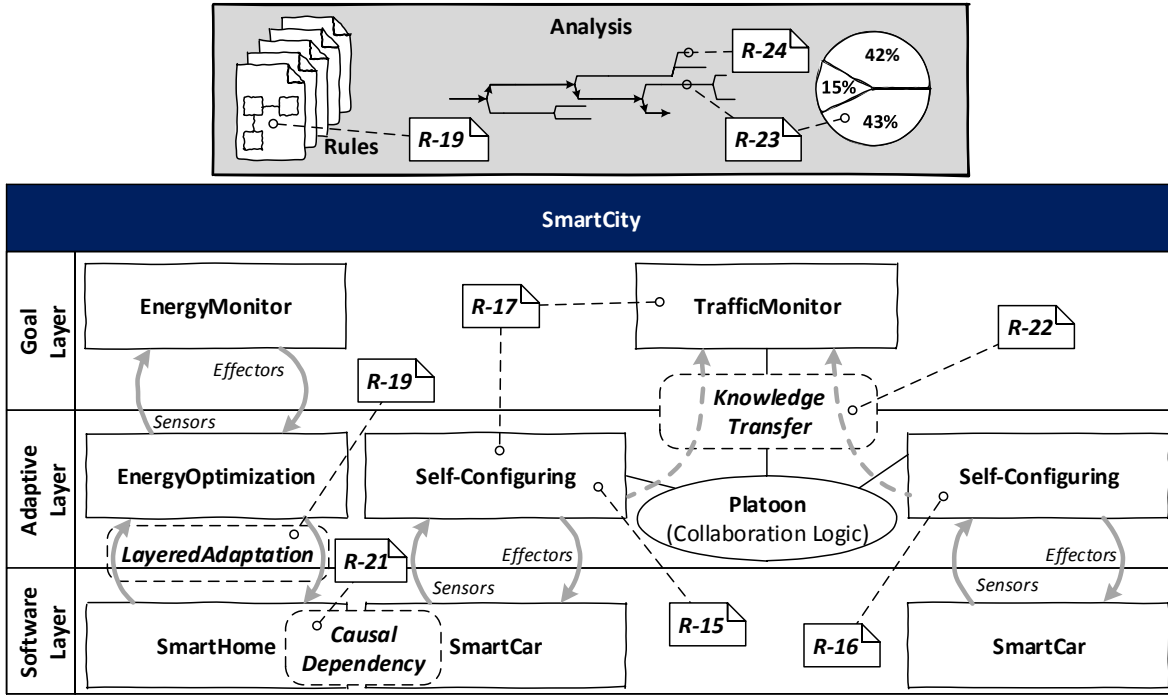


Figure 4.7: Deurema analysis

same analysis rules that are used during the development can be applied at runtime, e. g., as monitoring rules, to observe the varying system state and report arising violations.

As indicated in Figure 4.7, Deurema explicitly captures reflective behavior (R-15, R-16) and meta-adaptation (R-17), where the effects become visible over different adaptive SoS layers that can be analyzed, too. From a system architecture perspective, patterns as well as anti-patterns (R-19) can be detected in the modeled SoS architecture such as layered adaptation or hierarchical control.

4.3. Deurema Simulation

Simulation (R-25) helps understanding and verifying the modeled as well as the realized adaptation behavior of the SoS and can be additionally applied to the available Deurema analysis approach. As shown in Figure 4.8, simulation of the adaptive SoS behavior is a crosscutting concern that can be applied during modeling, analysis, and realization of the Deurema models. Thereby, the execution and simulation of Deurema models is directly supported, whereas the simulation capabilities of the realized system depend on the used standards, technologies, and tools of the corresponding domain. Moreover, traces from the simulation run can be used for analysis afterwards, which might lead to a change in the modeled adaptive SoS architecture.

Thus, early simulation of Deurema models during the development supports the investigation of the modeled collaboration and adaptation effects, whereas a simulation of a concrete realization in the later development process may expose additional domain specific effects. However, in the context of this thesis the execution semantic of Deurema model elements is introduced, which is further realized by an interpreter. The Deurema interpreter directly

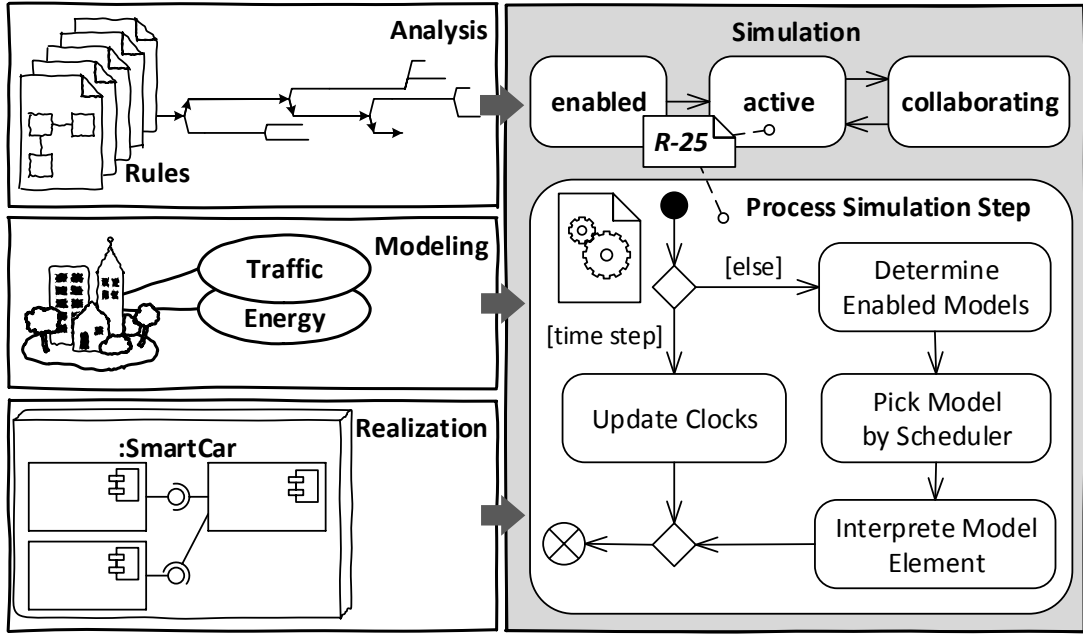


Figure 4.8: Simulating Deurema models

executes model elements and maintains well-defined execution states for each element such as *enabled*, *active*, or *collaborating* as depicted on the right in Figure 4.8.

Furthermore, based on the defined execution semantic of the Deurema interpreter, a simulation framework is presented that enables a *virtual* execution of the adaptive SoS behavior neglecting hardware and domain specific realization details. Additionally, the Deurema simulation framework handles the available knowledge in the system, resolves access and inconsistency conflicts as well as supports different simulation strategies. Thereby, the simulator introduces a timing concept, determines enabled models, supports different scheduling strategies for concurrent behavior, and uses the Deurema interpreter for executing single model elements as sketched in Figure 4.8.

4.4. Deurema Realization

Beside the modeling, analysis and simulation of the adaptive SoS behavior, the realization of Deurema models in the corresponding domain is important. At first, this thesis discusses how the component-based and rule-based domain specific extensions (R-20) in Deurema suit state of the art standards such as AUTOSAR [62] for cyber-physical system or graph based MDE approaches. Second, the realization of Deurema concepts is presented by showing an exemplarily mapping to de facto standard AUTOSAR from the automotive domain. Figure 4.9 depicts this conceptual mapping. In the example, the component-based local adaptive behavior of the smart car is mapped to an appropriate representation of a component architecture in the AUTOSAR standard.

On the one hand, the realized AUTOSAR architecture can be used for domain specific simulations (R-25) of the beforehand specified Deurema models taking target platform phenomena into account. On the other hand, available domain specific software tools such as

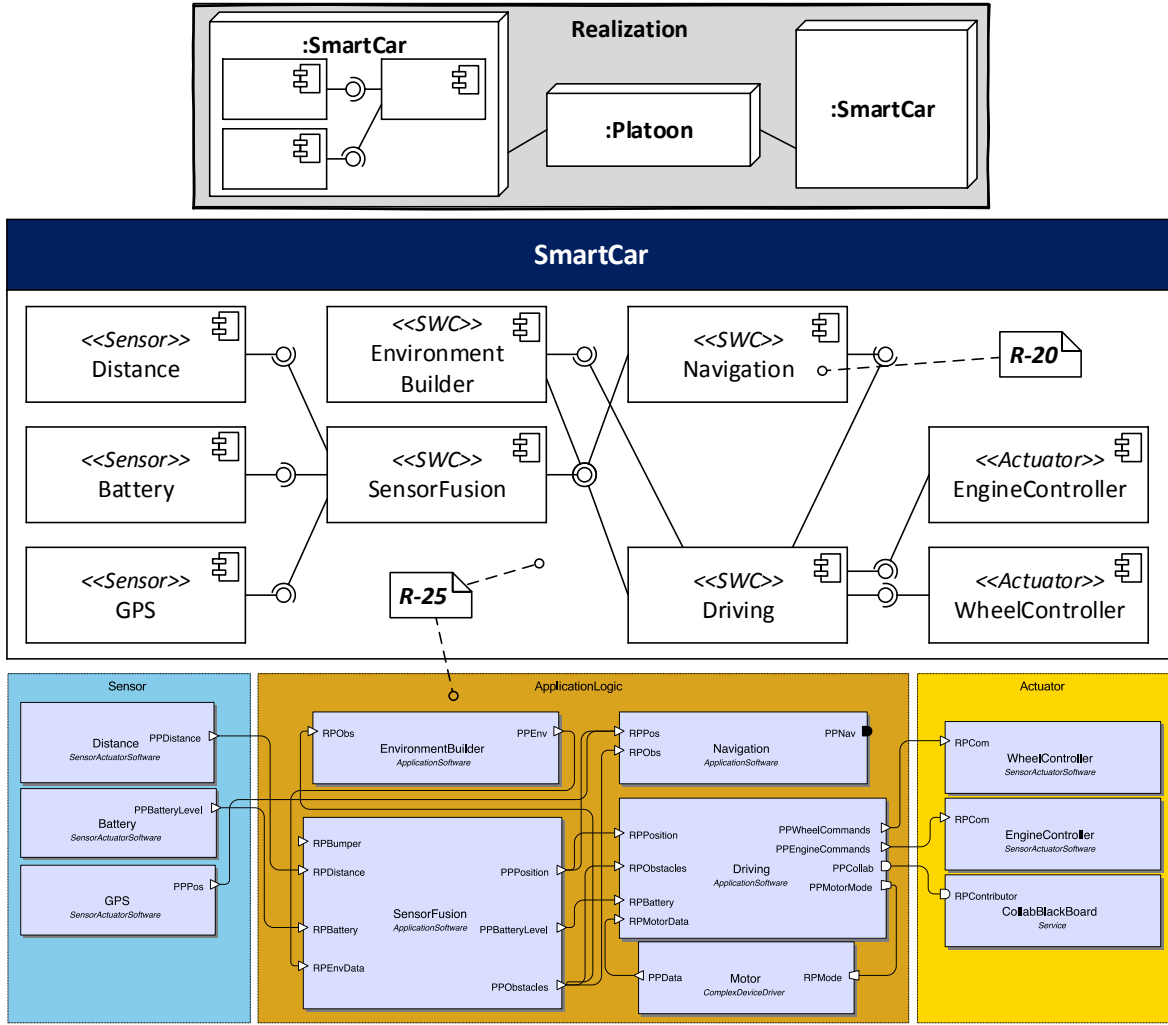


Figure 4.9: Realization of Deurema models

*Matlab*¹ or *SystemDesk*² help implementing the mapped AUTOSAR architecture, e. g., via code generation for the target platform. Furthermore, this thesis discusses how collaborations, feedback loops, and rule-based behavior can be realized in this domain.

In general, a concrete realization of the Deurema models introduces additional refinements of the modeled behavior. For example, in a real physical system, virtual components must be deployed to existing execution capabilities. The idealization of communication must be realized by an appropriate communication mechanism/protocol using available networks such as wireless networks or buses in a car. Additionally real-time constraints or limited memory resources further restrict the available realization space, if a concrete hardware platform is selected. Different middlewares or operating systems require application specific changes to realize the modeled adaptive SoS. Beside the mapping from Deurema models to an AUTOSAR conform architecture, other possible technologies and middlewares for a realization are outlined in this thesis.

¹<http://www.mathworks.com>

²SystemDesk is an AUTOSAR conform software tool from the dSPACE company <http://www.dspace.com>.

In summary, the Deurema approach facilitates the modeling of the local behavior of systems together with its adaptive capabilities as first class concept. The explicit specification of system collaborations is supported to describe the overall emergent SoS behavior. On basis of the Deurema models, this thesis discusses the investigation of the SoS architecture by means of static and runtime analysis as well as model simulation. A conceptually mapping of Deurema concepts to the AUTOSAR standard shows that the modeling concepts can be realized in a concrete application domain. Finally, the modeling language concepts are applied in case studies showing that the Deurema approach is powerful enough to cope with current research scenarios. In the following, the Deurema approach is discussed in detail.

5. Deurema Modeling Language

This chapter introduces the Distributed Eurema with Collaborations (Deurema) modeling language concepts that can be used to specify the behavior of a broad class of adaptive SoS. Deurema is designed for the description of the adaptation logic in independent, possibly distributed systems. Furthermore, the focus is on modeling the collaboration aspects between those systems by designing the interactions and their effects on the adaptive local system behavior. Therefore, the Deurema modeling language extends the concepts from the predecessor Eurema approach, which is already introduced in the preliminaries in Section 2.3. According to the derived requirements in Chapter 3 and the goals of this thesis, the outlined limitations of the Eurema approach are considered in the Deurema modeling language.

At first, the core Deurema concepts are introduced in Section 5.1. Furthermore, Deurema seamlessly integrates the use of runtime models that define the knowledge in the adaptive SoS, which is described in Section 5.2. Beside the knowledge, Section 5.3 discusses the integration of different domain modeling concepts into Deurema. Subsequently, the core, runtime model and domain concepts are used to describe the architecture of an adaptive SoS in Section 5.4. The comprehensively capabilities of collaboration specifications between independent systems inside the SoS are introduced in Section 5.5. Due to the focus on adaptive behavior, the modeling of reconfiguration, adaptation, and meta-adaptation in Deurema are described in Section 5.6. Finally, this chapter closes with a discussion about design decisions and the coverage of modeling language requirements in Section 5.7.

In the following, the Deurema modeling language concepts, the corresponding metamodel elements and the concrete syntax are introduced. Therefore, the running example in Figure 4.2 from the overview in the last chapter is stepwise refined pinpointing to different Deurema concepts. For each concept, this chapter depicts to the corresponding excerpt of the Deurema metamodel as necessary to underline the line of argument. The complete Deurema metamodel can be found in the Appendix A.

5.1. Deurema Core Concepts

A core concept of the Deurema approach is the modeling of a layered, adaptive SoS architecture as sketched for the running example in Figure 5.1. Therefore, a system is considered as first class entity in the Deurema approach. As depicted at the top in Figure 5.1, a system instance can be modeled using the UML object notation with the corresponding «*System*» stereotype. Thus, the both smart cities Potsdam and Berlin are modeled as two system instances. Beside a system instance, the internal behavior of a system is described in a Deurema system template as shown in the middle of Figure 5.1. A system template comprises the architectural, layered structure of the corresponding system, whereas the internal behavioral logic as well as the collaboration logic can be individually placed on the available layers. Thereby, a system template description can have an arbitrary number of layers. Up to this point, Deurema supports the modeling of system templates defining the system behavior and the usage of this template specification by means of system instances, which refer to its corresponding

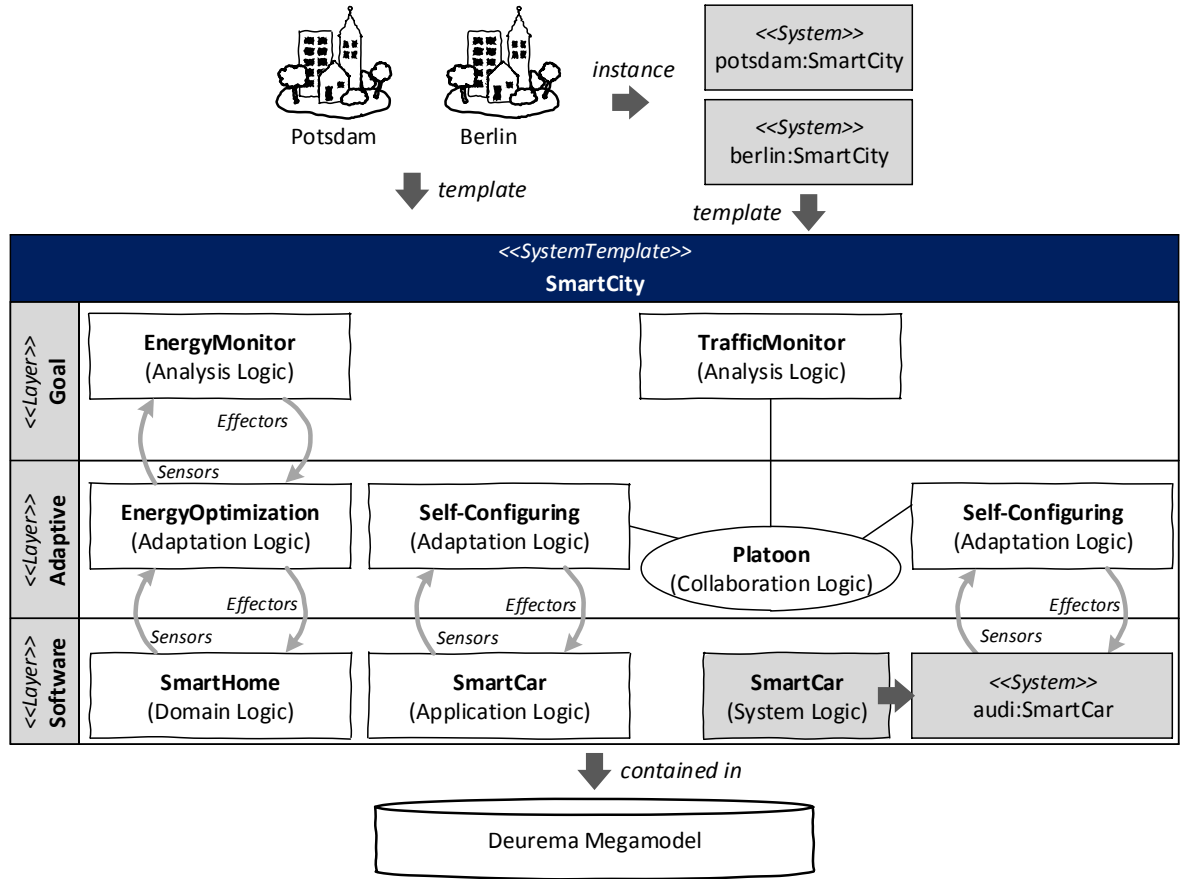


Figure 5.1: Smart city running example: Deurema system modeling

template description. System templates can be instantiated multiple times, which allows the reuse of defined system behavior. In the example, the template description of the smart city is used twice for the `potsdam:SmartCity` and `berlin:SmartCity` system instance.

Due to an adaptive SoS consists of independent systems, Deurema supports the specification of system hierarchies. Other system instances can be placed on the layered system architecture within a system template specification. In the example, the `audi:SmartCar` system instance is placed in the lowest layer in the smart city template description. This system instance refers to another system template description, named `SmartCar`, which again defines a layered system architecture together with the internal system logic. As a consequence, placing arbitrary system instances inside a system template facilitates the specification of system hierarchies. The hierarchical definition of systems and contained subsystems together with their behavior, which can be located on different layers, specifies the overall adaptive SoS architecture.

Deurema follows the idea of the MDE using models as first class entities. Therefore, the adaptive SoS architecture modeled as combination of system template and system instance are considered as model entities. As motivated in the preliminaries in Section 2.2.4, a runtime megamodel is able to maintain individual models together with their relationships to other models. Deurema used such a runtime megamodel approach as model management technique for all models created with the Deurema approach. Thus, the system template description and its contained internal behavior specification are maintained by the Deurema megamodel as shown at the bottom in Figure 5.1.

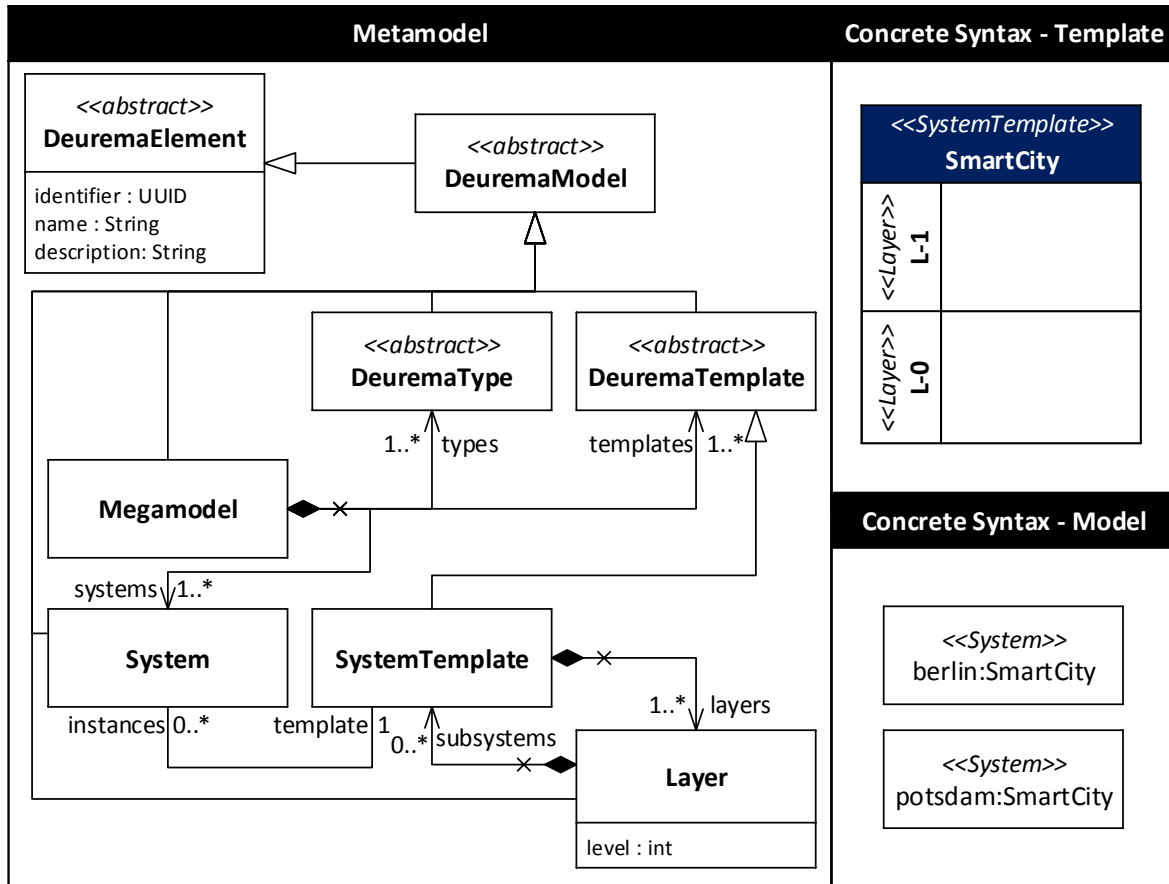


Figure 5.2: Deurema metamodel of core concepts

As a result of treating all Deurema entities as models, the **DeuremaModel** class at the top of the metamodel in Figure 5.2 realizes this concept. Every Deurema element inherits directly or indirectly from this abstract model class. Thus, each element can be treated as a model at the highest level of abstraction. Furthermore, the root element of a Deurema model specification is a megamodel that allows the reasoning about the model elements and their relationships (cf. discussion about MDE and megamodel concepts in Section 2.2). Therefore, the **Megamodel** class directly inherits from the abstract **DeuremaModel** in the metamodel in Figure 5.2. Additionally, the megamodel is a container for all other Deurema elements and thus, maintains the model elements as well as their relationships to other model elements.

As motivated in the example above, Deurema distinguishes between two further concepts, namely *types* and *templates*. Types represent predefined static Deurema elements that can be reused during modeling. Templates are blueprints similar to types, but allow more flexibility through the use of parameter variables that are assigned with concrete values during instantiation. Templates are used to support different variants of behavior descriptions in the adaptive SoS, such as feedback loops or component-based architectures. Both, Deurema types and templates must be instantiated, whereas the instances are handled as model artifacts within the megamodel. Consequently, Deurema comprises the three core concepts of types, templates, and instances. This is similar to the object-oriented programming approach. Types can be compared with the static description of a class, whereas each class can be instantiated multiple

times. Each instance follows the behavior description of the class and thus, behaves similar to other instances with the same class. Deurema templates allow an additional flexibility by introducing well-defined variation points that have to be resolved during instantiation. A variation point is similar to the use of generics in the *Java* programming language and the template mechanism in *C++*. During the execution, concrete Deurema model elements must be assigned to each variation point, whereas this technique is called *"type erasure"* in Java. As a consequence, instances of the same template may show a different behavior depending on the resolution of the variation points, although all template instances follow the same blueprint of the static template specification part. Deurema types are represented by the corresponding `DeuremaType` class, whereas templates are represented by the `DeuremaTemplate` class in the metamodel in Figure 5.2. Furthermore, a system template defines the layered system structure and can be instantiated, which leads to a system instance. The template-instance relationship between system template and system instances is accordingly defined in the Deurema metamodel in Figure 5.2. Additionally, the `SystemTemplate` class refers to an arbitrary number of layers, which defines the layered system architecture. Each layer can contain an arbitrary number of subsystems, which leads to hierarchies of systems and thus to a SoS model.

In summary, the Deurema system template and megamodel are the two main concepts of realizing a specification of the adaptive SoS. A system template defines the layered architecture, which further contains the internal adaptive system behavior in form of feedback loops and the system knowledge defined in appropriate runtime models. System templates can be instantiated and the corresponding system instance can be again placed on a layer in another system template. The megamodel maintains all Deurema model elements and their relationships, which further enables the reasoning about the modeled SoS specification. Consequently, the megamodel contains the modeled system template definition and system instances as well, which is defined by the corresponding containment relations in the Deurema metamodel.

Concrete Syntax

Figure 5.2 shows at the right the concrete syntax of a system, whereas the template notation is depicted at the top and the instance (model) notation at the bottom of the figure. At template level, a system has a name and is stereotyped with `«SystemTemplate»`. It includes an arbitrary number of layers that are modeled as `«Layer»` stereotyped, named lanes, respectively. At model level, systems are modeled as rectangles that refer to their template and may have an additional instance name similar to UML object diagrams.

For better recognition, the concrete syntax of Deurema templates and instances follows the same design principles for the rest of this thesis. Templates have always a stereotype, which includes the string *"Template"*. Furthermore, model instances follow the notation of UML object diagrams, which consists of an arbitrary name, followed by a colon and the Deurema template or type correspondence. Additionally, instances are stereotyped according to the Deurema metamodel element to which the instance belongs to.

The concrete syntax examples on the right in Figure 5.2 show a system template definition, which is named `SmartCity`. Thereby, the template has two layers `L-0` and `L-1`. On model instance level, there are two system instances, which is indicated by the appropriate stereotype. Furthermore, both instances refer to the same `SmartCity` template as shown above, where the first instance is named `berlin` and the second one `potsdam`.

System Template Example

After the introduction of the Deurema core concepts, Figure 5.3 shows the specification of a smart city template, which corresponds to the running example of this thesis. The smart city system template definition comprises two layers, two contained system instances and two feedback loops. At the lowest layer, there are two smart car system instances named *audi* and *bmw*, whereas the corresponding template descriptions are not modeled in detail. On top of the car systems in layer L-1, there are two feedback loops instances defining the internal, adaptive behavior of the smart city. The feedback loops reflect and affect the corresponding smart car system at the layer below and introduce a self-configuring functionality, which allows the switching of the car behavior between autonomous and manual driving. The internal behavior specification of the feedback loop follows the description as motivated in the overview chapter in Section 4.1.

A system template defines the layered architectural structure, which includes other systems and the relationships between systems. However, the internals of behavioral parts as well as the knowledge specification of the adaptive SoS are not modeled in system templates. In the following sections, the internals of systems are discussed by looking first at the Deurema runtime model concept, which refines the abstract notion of knowledge. Afterwards, the specification of feedback loops is explained, which defines the adaptive capabilities of the systems and refines the behavior modeling concept of the Deurema approach.

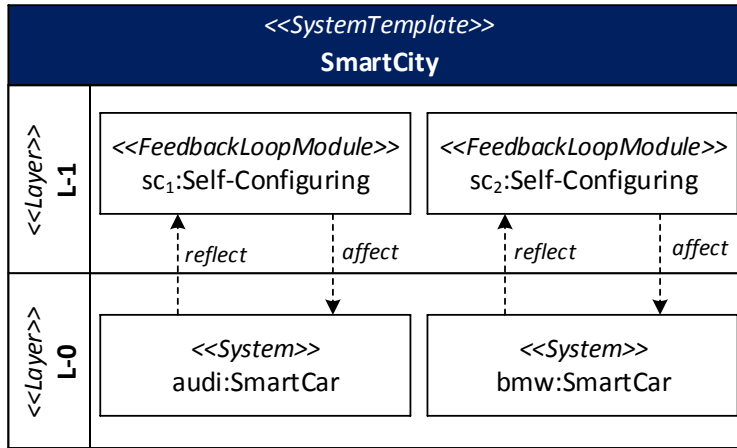


Figure 5.3: Deurema system template example

5.2. Deurema Runtime Models

As motivated in the preliminaries in Section 2.2.3, the Models@runtime (MART) approach proposes the usage of models not only during the development of the system, but rather as abstract system representations that are kept alive at runtime. Therefore, the same models that describe structural and behavioral system aspects during the development can be reused to describe the current system structure or system state at runtime. The advantages of this approach are, among others, the reduction of complexity by focusing on key aspects of interest, whereas models must not be reinvented, but rather can be reused. Furthermore, the same model-based techniques from the MDE can be applied for runtime models as for development models. Therefore, the feedback loops, which realize the adaptive behavior of the system,

can directly operate on the abstract representation of the running system by reading and manipulating the corresponding runtime models. Due to the causal connection, the changes in the runtime model will lead to an appropriate change in the behavior of the underlying physical system.

Inspired by the advantages of the MART approach, the Deurema modeling language considers runtime models as first class entities. Thus, runtime models define the amount of available knowledge within the system template specification as sketched in Figure 5.4. Thereby, the Deurema approach supports the definition of local views for each behavioral entity placed on the layered system template specification. Examples for such behavioral entities are other system instances, feedback loops, or collaborations. As a consequence, each entity has a local view on its available amount of information, which can be individually accessed and manipulated. This local knowledge can be shared during the interaction of systems. Runtime models and the local views on the global available information are maintained by the Deurema megamodel.

Unfortunately, as discussed in Section 3.3, there is no common consensus in literature about the different types of runtime models that define the purpose or intention of a runtime model describing a specific part of the system. Because the Deurema language enables the specification of system interaction via collaborations, available runtime information is shared between systems. Therefore, the available knowledge spreads over time in the adaptive SoS. Furthermore, due to the manipulation of the runtime models and the causal connection, systems become enabled to influence each other via collaborations and the shared runtime models. Thus, the notion of the runtime model purpose must be refined to provide clear modeling concepts that allow the usage of different runtime model types, to define the intention of a runtime model, and finally to allow the reasoning about collaboration effects via shared knowledge during system interactions.

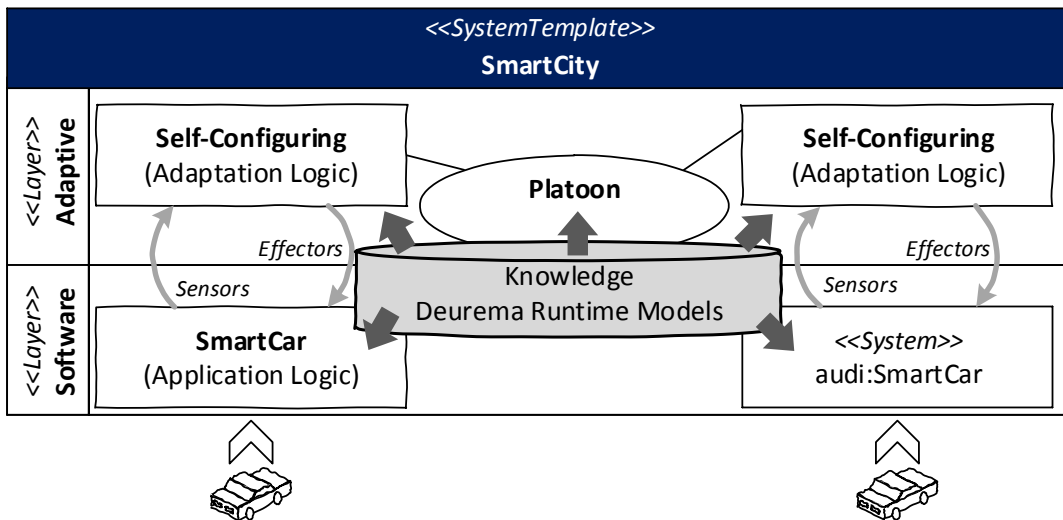


Figure 5.4: Smart city running example: Deurema runtime models

5.2.1. Runtime Model Categorization

This section discusses the Deurema runtime model categorization, which is based on own former work in [9], but allows a seamless integration of the runtime model approach into the Deurema language. The categorization is developed with respect to the Deurema collaboration concept and enables knowledge analysis such as its distribution within the modeled adaptive SoS. In the following, each runtime model category is discussed in detail.

Reflection Models

Runtime Reflection Models describe concerns that are related to the running system in System Models as well as system context in Context Models as shown in Figure 5.5. Therefore, they reflect runtime phenomena on a higher level of abstraction.

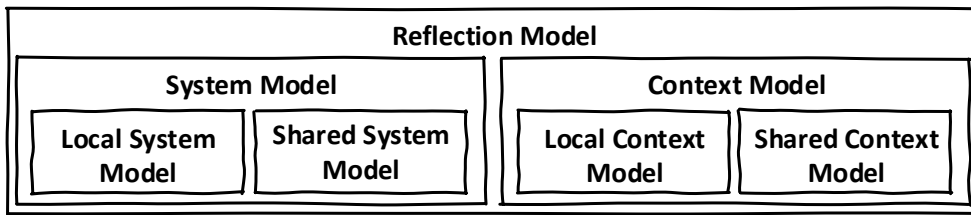


Figure 5.5: Runtime reflection model types

System Models

System Models are directly causal connected and provide reflected *architectural* and *behavioral* views of the system for the key points of interest. The structural system parts are often modeled in form of component or class diagrams as in [38, 79, 174], whereas behavioral system aspects are often modeled as tasks [40], processes (e.g., business process diagrams) or automata [103]. A fine-grain classification of System Models with respect to the visibility leads to the Local System Models and Shared System Models categories. The difference between both runtime model types is explained with the help of Figure 5.6.

Conceptually, there are two layers depicted. The physical layer on the bottom represents the real world that consists in this example of two systems s_1 and s_2 (gray cars in the figure). The software layer on top includes two distinct smart car software systems, where SC_1 belongs to the real system s_1 and SC_2 to s_2 . Furthermore, each smart car software system holds different runtime models. According to the categorization, SC_1 contains a local system model of s_1 and a shared system model of s_2 , where SC_2 has only a local system model of s_2 . For this example, it is not important if the shared model of s_2 is a copy or reference in the software system SC_1 but rather the visibility of that model. It is a shared model, because the SC_1 software system can access and manipulate the model of s_2 , although it is not part of the original, physical system. A local system model becomes a shared system model for another system if it is for example exchanged or accessed during system collaborations. Deurema supports different mechanisms for exchanging runtime models via system interaction such as direct message exchange.

Following the definition of a runtime model, model manipulations in the local runtime models cause local system changes that can be recognized as direct causal connection. In contrast, changes in shared system models affect runtime models in collaborating systems over the indirect causal connection. In the example in Figure 5.6, a change from SC_1 in the model

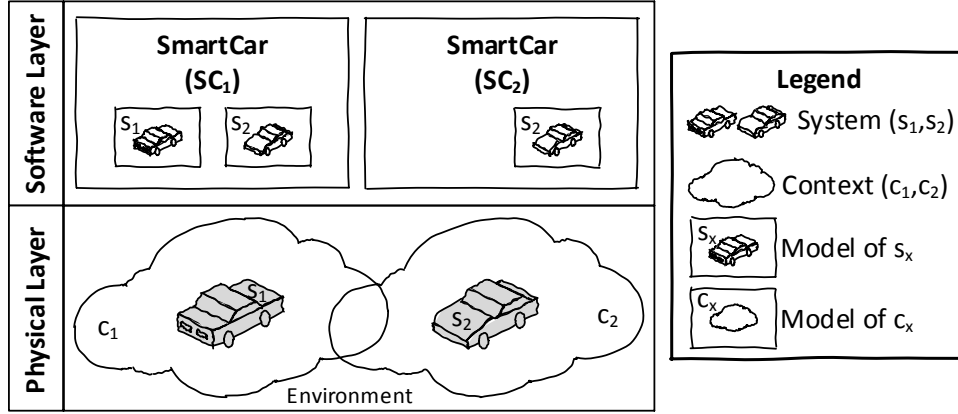


Figure 5.6: Runtime system models

of s_2 will cause a change in the corresponding model of SC_2 and finally, affect the physical system s_2 via the causal connection. The distinction of direct and indirect influence enables further analysis and the reasoning about collaboration effects in the adaptive SoS.

Context Models

A definition for system context is given by Dey as *"any information that can be used to characterize the situation of an entity"* [67]. According to this definition, the adaptable system is considered as the *entity* and the situation is described in runtime Context Models. Additionally, system context and system environment are distinguished. The system context can be captured by the system itself (e. g., via sensors), is a subset of the environment of the system, and maintained in runtime Local Context Models. Furthermore, the system environment is a superset of the system context that additionally contains all not detectable parts of the systems' surroundings. As a consequence of the context definition, Context Models and System Models are conceptually disjoint.

Context information is important for different adaptive SoS such as smart homes [140], mobile web services [95] or search and rescue scenarios [144]. Therefore, the representation of context information varies from sensed 2D maps to large, abstract network graphs describing the communication connection capabilities of the SoS. Often, the current situation is reflected in context models so that the adaptive system is able to react appropriately. As an example, a smart home may close the windows if it starts to rain. Therefore, sensors must be aware of the current weather situation.

Shared Context Models capture additional information about the context of other systems that can be collected via collaborations. The distinction between shared and local context information follows the same line of argument as for system runtime models above and is depicted in Figure 5.7. The contexts c_1 and c_2 (gray parts in the figure) are sensed and represented in the corresponding software systems SC_1 and SC_2 respectively. Furthermore, SC_1 contains additional context information from the system s_2 that is represented in the shared c_2 runtime model.

While the distinction in local and shared runtime models follows the visibility characteristic of information within systems, the difference between system and context is defined by the system border. Another consequence of the separation between system and context is the direction of the causal connection. For system runtime models, the causal connection is bidirectional that means the physical system situation is represented into the model or

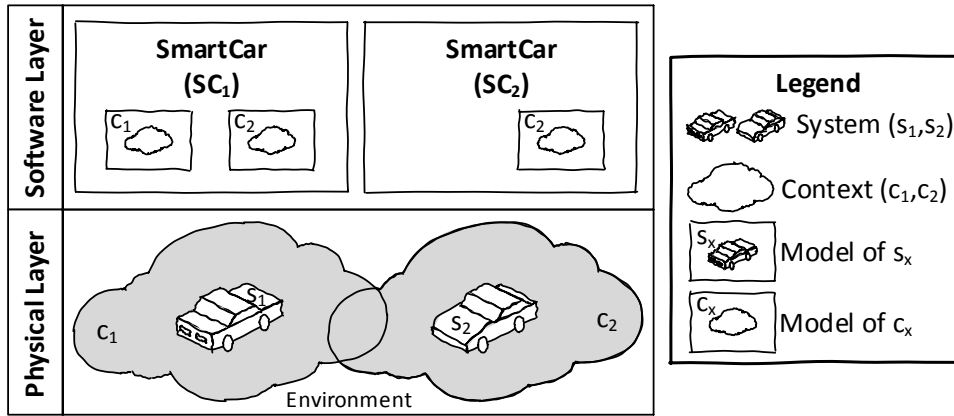


Figure 5.7: Runtime context models

enforces corresponding changes in the model and vice versa. Context runtime models have a unidirectional mapping from physical context information that is captured by sensors to the corresponding model representation, but not into the other direction. Consequently, systems cannot directly influence their context by manipulating the runtime models. Of course system types such as embedded systems or CPS can have physical effectors that influence the context. For example a heater can emit thermal radiations that increase the overall temperature in the context of the heater. However, the control over the heating behavior as well as the representation of the heating capabilities (effectors) are captured in the corresponding system runtime models, whereas the context runtime models represent the sensed temperature (situation).

Adaptation Models

Figure 5.8 depicts the category of runtime Adaptation Models that describe the expected and possible solution space of the system. Thereby, Evaluation Models determine the overall system specification, which includes the modeling of functional and non-functional properties. Furthermore, Change Models describe variants of the adaptable software system.

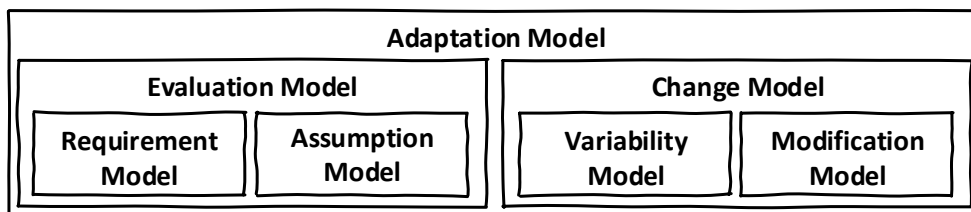


Figure 5.8: Runtime adaptation model types

Figure 5.9 shows the difference between evaluation and change models. Conceptually, there is a possible solution space for the complete system that fulfills all requirements, goals, and needs. A violation of requirements leads to a leaving of the system from the possible to the invalid solution space. Furthermore, within the valid possibilities of realizing a system, the evaluation models describe the specification of the system, which directly targets the current system solution (white cloud in Figure 5.9). Whereas, change models enable an evolving of

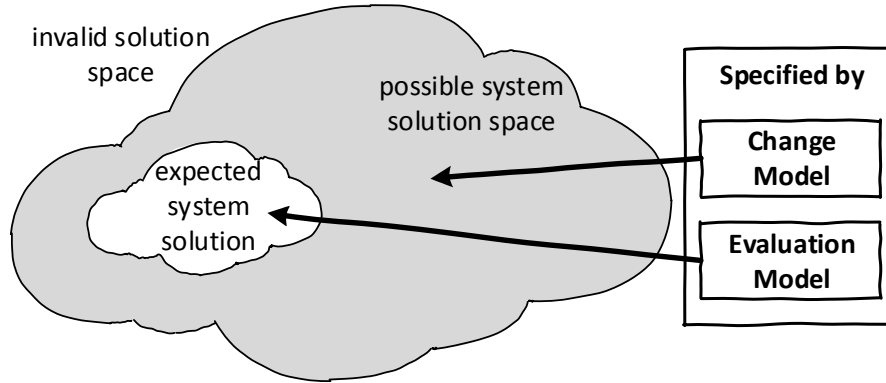


Figure 5.9: Change and evaluation model

the current system solution that can be described as moving it in the inner valid solution space (gray cloud in Figure 5.9).

Evaluation Models

As defined in [188, p. 27], the system specification consists of requirements and assumptions about the context. On the one hand, requirements are often declaratively modeled and define the amount of functionality and capabilities a system must satisfy to fulfill customer needs. On the other hand, assumptions are also called domain knowledge and describe demands on the system context to guarantee proper execution conditions and correct system functionality. Therefore, the runtime model categorization captures the system specification in Requirement Models and Assumption Models accordingly.

The need of reflecting requirements and handling them as first class entities is described in [33]. Furthermore, some requirements, as for example goal models, can be operationalized, which enables online validation and verification of the system. Examples for Evaluation Models are the KAOS goal model used in [122], Quality of Service (QoS) constraints as used in [56], the Object Constraint Language (OCL) in [176] that describes architectural constraints on models, and the Linear Temporal Logic (LTL) in [85] for checking consistency of system adaptation processes. Beside concrete system goals, requirement runtime models may contain utility functions that optimize important criteria of the system such as throughput or energy consumption. Optimizing the system behavior according to given utility functions raises the need for system adaptation.

An analogy for assumption models is the assertion concept of object-oriented programming languages such as Java or C++. Such assertions define assumptions about the expected execution context or system state during the lifetime of a software program. Similar, an assumption runtime model defines such constraints, which describe the expected situation of the system or context.

Requirement models are the main trigger for an adaptive system to change its behavior during execution. They must be considered over the whole lifetime of the system and the adaptation logic should react to changes in the requirement model properly. Changes in requirements can have different reasons as for example changing user needs, environmental conditions or the aging of the hardware/software system itself. Following the proposed categorization, Evaluation Models are the source respectively driving force for changes in the system, but the target respectively needs of an adaptation are annotated directly in the corresponding system runtime model.

Change Models

Beside the expected solution space of the system in the Evaluation Models, Change Models describe possible solutions, where the system might adapt to. Figure 5.10 shows the difference for the two categories of change models. Variability Models explicitly model the possible solution space in form of selected, meaningful configurations, which is a subset of all possible system solutions, similar to software product lines. During the adaptation process, one configuration can be chosen that will cause predefined effects on the system (e.g., exchange of a component or a mode switch). Different system configurations are often optimized concerning given, well-selected criteria. For example, considering the throughput of a server and the energy consumption of it as two important, but conflicting criteria. It might be useful of modeling two distinct system configurations, whereas each configuration optimizes the system behavior according to one criterion. During runtime, the best fitting configuration can be selected according to the current goals.

In contrast, Modification Models implicitly model the possible solution space. Thus they define a superset of all possible solutions, by specifying all permitted modifications of the Reflection Models. Normally, such modifications are described as rules comprising trigger conditions and operations. They can be combined to an arbitrary sequence realizing complex system changes. Considering the example above, it might be inefficient or not possible to specify all meaningful variants of combined goals saving energy and optimizing the throughput of a server. If the utility functions together with a predefined set of possible system modifications are given, the system can monitor both metrics at runtime and modify the behavior step by step via the available modifications towards an optimized combined behavior.

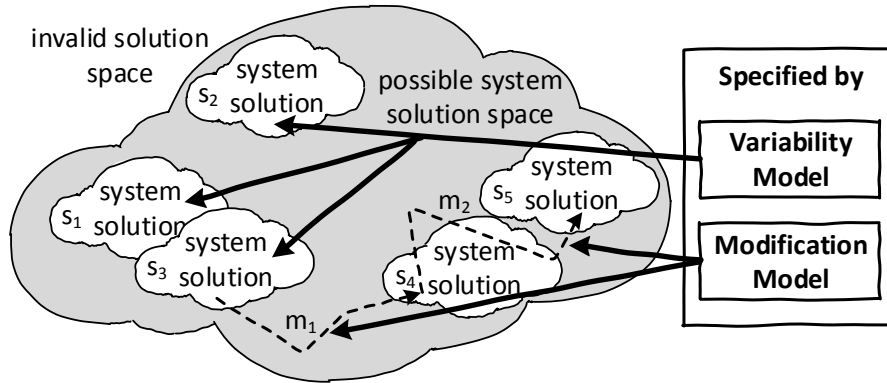


Figure 5.10: Variability and modification model

The conceptually example in Figure 5.10 depicts three possible system solutions s_1 , s_2 , and s_3 defined by configurations in the variability model. Furthermore, the system solution s_4 can evolve from s_3 by applying the m_1 modification and s_5 from s_4 by m_2 . All possible modifications are specified in the corresponding runtime modification model type. Because runtime models can be represented as graphs, change models are often realized by graph transformation rules that operate on the corresponding system runtime models.

In general, McKinley et al. distinguish static, configurable adaptation, which is “*parameter adaptation [that] modifies program variables*” [134] and dynamic adaptation, which is “*compositional adaptation [that] exchanges algorithmic or structural system components*” [134]. Both types can be realized by variability and modification models, where examples from the literature for each combination are subsumed in Table 5.1.

Table 5.1: Change model examples for parameter and dynamic adaptation

	Parameter Adaptation	Dynamic Adaptation
Variability Model	(1) Embedded Systems [117]	(2) Smart Homes [57, 138]
Modification Model	(3) Micro-Grid [76]	(4) Graph Transformation [100, 174]

For combination (1), variability models for parameter adaptation are often used for embedded systems as outlined by Kokar et al. [117]. This combination of a limited configuration space together with predefined parameters is very suitable for embedded systems. Resource restrictions as limited CPU or energy power can cause problems for dynamically deploying components as done by dynamic adaptation activities. Furthermore, the timing aspect becomes important, whereas the overall adaptation must still guarantee safety requirements. For example, a reconfiguration in a car must always guarantee the correct functionality of the airbag in the case of an accident.

Concerning combination (2), Cetina et al. [57] use variability models in the context of smart homes for reconfiguring a predefined set of smart devices in a house according to different user scenarios. For example, if the user is at home, the security alarm is disabled but the lights react on moving activities of the user. If the user leaves the house, the security component is enabled and the movement sensor functionality is reconfigured to detect possible intruders. This example uses a variability runtime model, which includes different configurations, whereas the application of one configuration triggers the dynamic (un)loading of software components. A similar example is given by Morin et al. [138], who use feature models to dynamically derive the system architecture.

Often, modification models are used for adapting the system from one configuration to another, whereas the adaptation process includes multiple modifications rather than an all-in-one deployment of the whole configuration. As a consequence, modification models allow a fine-grain manipulation of the system during adaptation, whereas the explicit modeling of all configuration possibilities is not appropriate (e. g., because of the total number of all configurations) or the change from one configuration to another is not feasible in one step. In combination (3), an example of using modification models for parameter adaptation is presented by Frey et al. [76], who reduce the power consumption of smart lamps and heaters within a house. The power consumption is modeled as parameters (e. g., for lamps full or reduced – 100 watt or 30 watt), which are changed between different phases during one simulation of a smart electrical micro-grid scenario.

Finally, the use of modification models for dynamic adaptation in combination (4) are common for scenarios, where adaptation triggers together with adaptation actions are used to change the overall architecture of the system. For example, Hirsch et al. [100] use graph transformation rules for varying the system architecture for coordination purposes during collaboration in mechatronic systems. Furthermore, Vogel et al. [174] use triple graph grammar rules for the runtime deployment of components in an adaptive web shop example.

In summary, Change Models describe the solution space of possible system adaptations. Due to the causal connection, applying the Change Models on System Models will enforce the desired adaptation in the underlying physical system.

Causal Connection Models

As last category, Causal Connection Models establish the beforehand mentioned causal connection of runtime models to the underlying physical system (cf. Figure 5.11). Monitoring Models retrieve information from the running system and update the information in the corresponding Reflection Model. Additionally, Execution Models propagate changes from the Reflection Model back to the running system. Therefore, Monitoring Models describe the mapping of system-level observations to the abstraction level of the Reflection Models and Execution Models translate runtime model adaptations to system adaptations. Usually, causal connection models depend on implementation specifics in the adaptable software and can be realized by MDE techniques such as graph transformation or bidirectional triple graph grammar rules as done in [175]. In the context of embedded systems, hardware sensors and effectors might be available that realize bridging the gap between real system observations and their software representation. However, in most cases additional, application specific filter or adapter must be realized to reach the envisioned level of abstraction from low level runtime phenomena to high level runtime model representations.

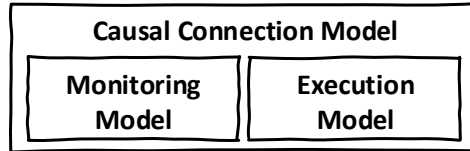


Figure 5.11: Runtime causal connection model types

5.2.2. Runtime Model Integration

The presented taxonomy for runtime models leads to a refined notion for the knowledge in the MAPE-K feedback loop approach, which is introduced in the context of self-adaptive systems in Section 2.1. Deurema follows the MAPE approach and uses the presented runtime model categorization above to integrate the knowledge into the feedback loops of the adaptive SoS behavior. Figure 5.12 shows the typical usage of runtime models in an idealized view of MAPE activities within one feedback loop. First, the monitor activity uses Monitoring Models for establishing the causal connection to the adaptable software part. Afterwards, observations of interest that can be sensed are written to the abstract system and context runtime model representations, which are located in the Reflection Models. Therefore, this reflected information is the basis for all analysis and planning activities of the adaptation engine. Furthermore, the abstract representation of the running adaptable software part in the reflection models allows a fully decoupled analysis and planning of adaptation activities, although the real system does not stop its operation. The analysis, whether an adaptation of the system is necessary or not, is supported by requirements and assumptions of the system that are available in Evaluation Models. Planning activities perform the adaptation directly on the reflected system runtime model by considering appropriate strategies and configurations that are stored in Change Models. Finally, the planned adaptation changes are synchronized with the adaptable software underneath by applying given execution strategies contained in the Execution Model.

Deurema extends the idealized, simple view of Figure 5.12 with respect to multiple feedback loops that can contain an arbitrary number of different adaptation activities. Furthermore, the adaptation activities can access an arbitrary number of runtime models. Depending on

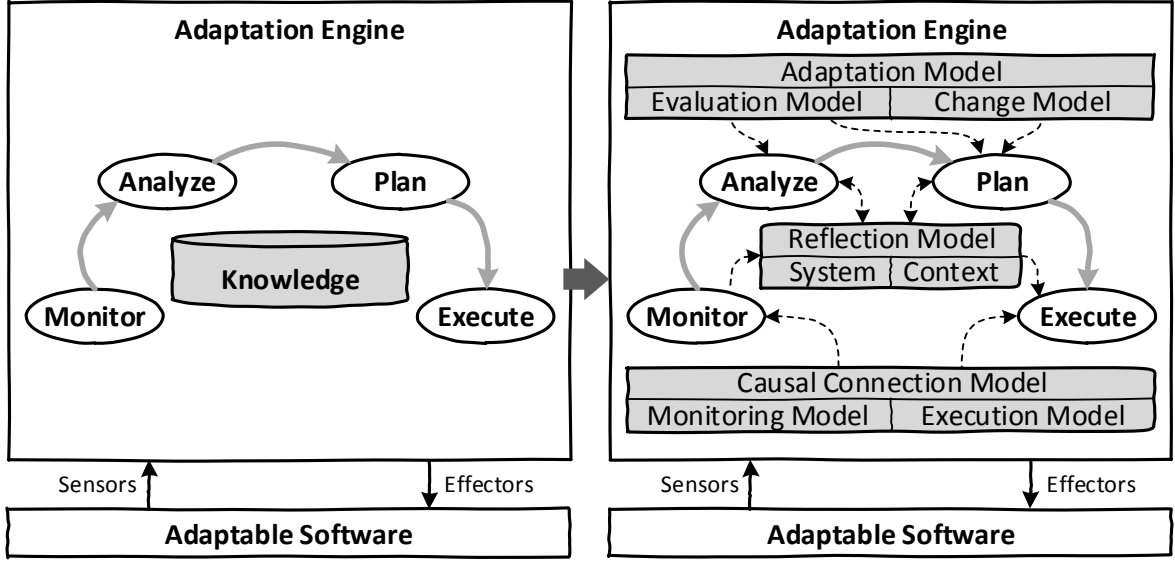


Figure 5.12: MAPE feedback loop with runtime models

the architectural structure of the adaptation engine, it is possible to specify hierarchical and collaborating feedback loops. In such cases, different runtime model types help identifying typical control architectures such as layered, hierarchical, or distributed. For example, Causal Connection Models can be used to decouple the adaptable software from the adaptation engine by simultaneously maintaining the causal connection between both. The same principle of sensing information and synchronizing adaptation changes can be applied between hierarchical feedback loops that are located on different layers within the adaptation engine. Thus, a feedback loop on a higher layer can reflect ongoing adaptation activities of a lower layer, which further enables meta-adaptation as introduced by Hillman et al. [99] and discussed in the preliminaries in Section 2.1.1.

Salehie et al. [156] present a hierarchy of self-* properties, where *self-awareness* and *context-awareness* are considered as primitive properties in an adaptive system. They are the basis for higher level adaptive capabilities such as *self-healing* or *self-optimizing*. Runtime model types together with an analysis of their usage enable a precise definition of the modeled adaptive system behavior. For example, a distributed system of mobile devices, as presented by Han et al. [95], has information about its context. If this information is kept alive at runtime in the corresponding context reflection models and if adaptation activities access these runtime models, the overall system can be considered as *context-aware*. The same line of argument holds for the other primitive properties, as for example *self-awareness* (cf. [70]) that arises by using a system runtime model or *requirement-awareness* (cf. [33]) that is caused by the use of an evaluation runtime model.

However, assigning available knowledge to the corresponding runtime model category depends on the application domain and the captured information. Therefore, the developer of the adaptation engine must choose one or more appropriate categories for each specified runtime model. For example, due to a specific realization or design decisions during the development, one runtime model may contain information about the inner system state and its context. In such cases, the runtime model belongs to both categories System Model as well as Context Model and represents a combination of both. Furthermore, if runtime models

are automatically derived from development models, inference techniques may be applied to retrieve the corresponding runtime model category, which is not in the focus of this thesis.

In summary, the runtime model categories divide the available knowledge of the adaptation engine in well-defined parts. Moreover, each category targets another purpose of available system information. As a consequence, the adaptation engine includes a set of distinct, loosely coupled runtime model types that are accessed by and arranged around different feedback loop activities. Finally, the categorization enables the identification of different dependencies between runtime models and feedback loop activities that are used to define analysis rules as well as retrieving characteristics of the overall adaptive system.

5.2.3. Runtime Model Example

For visualizing the overall interplay of different runtime model types, Figure 5.13 shows an exemplary view from the self-configuring feedback loop of a smart car from the running example in the overview in Section 4.1.1. On the bottom, the physical situation is depicted, which shows the smart car in a normal traffic situation within the smart city. For this example, the driver of the car decides to switch from a manual to an autonomous driving mode, which can be activated by pushing a button in the user interface of the car. The layer on top of the physical situation description consists of two parts that are the software component architecture of the smart car encapsulated in a corresponding software layer and the self-configuring feedback loop, which is placed within the layer on top. The component architecture of the smart car contains the domain logic to realize the manual or autonomous driving, where the internals of the components are not important for this example. Furthermore, the feedback loop on top reflects and affects the behavior of the smart car and consists of four adaptation activities designed according to the MAPE reference architecture from [110].

Before the adaptation logic realizes the switch from manual to autonomous driving, the system already executes a *collision prevention assistance*¹ functionality, which senses the context of the car and brakes in the case of detected obstacles. The collision prevention logic is conceptually depicted by the *SensorFusion* and *Navigation* component, where the gray background denotes the current state of the execution. Thus, the smart car currently performs the checks if an obstacle is in front of the car by evaluating the sensor data in the *SensorFusion* component.

For this example, the switch from the driver from manual to autonomous driving triggers the execution of the feedback loop in the adaptation engine at the top in Figure 5.13. First the *Update* activity performs a (1) read operation on the monitoring model. The read update rules allow the retrieving of the current system state of the smart car, thus establish the causal connection, and represent the system state as automaton. Therefore, the update activity applies the monitoring rules and (2) writes the retrieved information into the reflected runtime representation in the system model. Thereby, the complex component architecture of the smart car is transformed to a more abstract state representation in form of an automaton. Such abstractions of reflected system phenomena are typically for the runtime model approach, which enables the handling of complexity and focus on key points of interest. Sensing the context of the smart car and annotating the information in the corresponding context runtime model can be performed in the same way, which is omitted for clarity in the figure.

¹Collision prevention assistance systems already exist in modern cars. More information can be found at <http://media.daimler.com/dcmmedia>.

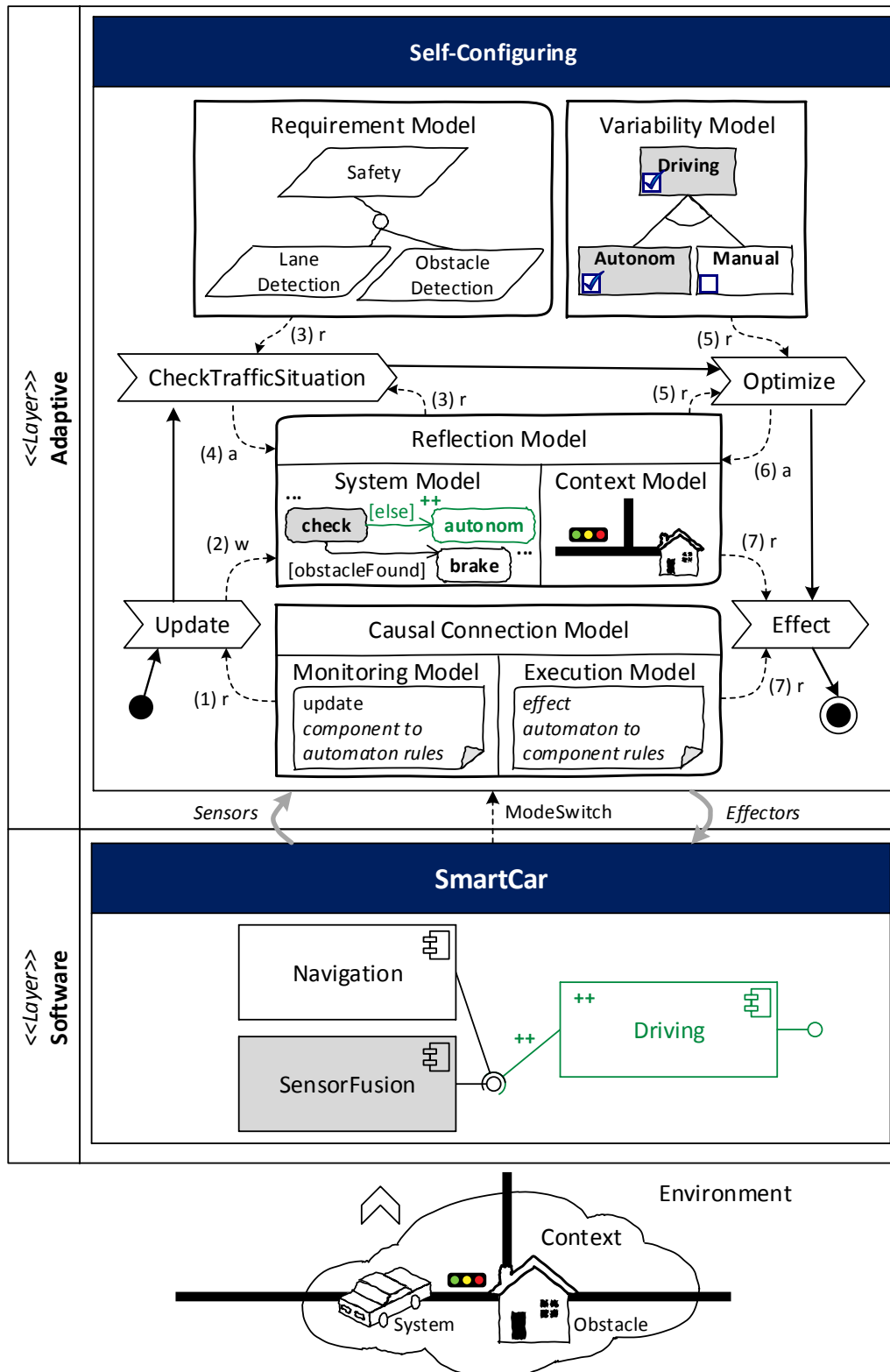


Figure 5.13: Sketch of a self-configuring smart car with runtime models

After all information is up-to-date, the analysis of the reflection model can be performed, which is done by the `CheckTrafficSituation` activity. The `analyze` activity first (3) reads the system model together with the current requirements for the system. In the example, the requirements are specified in form of a goal model, where the depicted excerpt includes a required lane and obstacle detection as safety feature for the autonomous driving. The analysis detects that the current system model must be adapted to perform according the given goals and the desired mode switch from the driver, which is (4) annotated directly in the reflection model. Consequently, the planning activity `Optimize` (5) reads the need for an adaptation from the reflection model together with a set of possible system configurations modeled in a variability runtime model. The gray part in the variability model in Figure 5.13 denotes the preferred current configuration, which is the autonomous driving feature. The result of the adaptation plan is (6) annotated into the system model, which leads, in this example, to a new state in the automaton representation named `autonom` (marked green with a ++).

The last step is the synchronization of the adaptation plan to the smart car and thus, to the running physical system. Therefore, the `Effect` activity (7) reads the planned changes together with rules that allow a transformation from the automaton in the system model to an executable component architecture for the underlying smart car software system. As an effect of the synchronization (causal connection), a new component `Driving` (green and marked with ++) is deployed in the smart car. Thus, the adaptive car, shown in the physical layer, is now able to drive autonomously, which is the expected functionality that was triggered by the driver.

This theoretical single run of an adaptation loop illustrates the use of runtime models and the effect of the causal connection. On the one hand, runtime models are an abstraction of runtime phenomena for key points of interest, which enables a decoupled analysis and planning of the necessary adaptation steps. On the other hand, all retrieved information and changes must be synchronized with the underlying system in dedicated points in time to enforce the desired adaptation behavior.

5.2.4. Runtime Model Metamodel

After having a common understanding of the different runtime model purposes and their usage by adaptation activities, the Deurema metamodel for runtime models is described. Runtime models are considered as explicit entities in Deurema specifying the complete available knowledge of the system. Therefore, runtime models are modeled as Deurema types that are contained in the megamodel as depicted in Figure 5.14. Furthermore, each runtime model has at least one purpose that follows the categorization as comprehensively discussed above. In Deurema, a runtime model type represents the global available knowledge in the system. Therefore, each runtime model can have an arbitrary number of views spread throughout the system. Views can be considered as individual instances of the runtime model, which usually contains only parts of the complete available knowledge. Furthermore, views belong to a specific part of the system behavior defining the available local amount of information on which the system functionality can operate on. This concept enables maintaining individual views in different system parts on the global available knowledge in the overall SoS. Therefore, Deurema supports the modeling of visibility restriction and partial knowledge with two implications. First, operations and manipulations of the runtime models are enforced over the corresponding views and cannot be applied directly into the global runtime model type. Second, runtime models are directly contained in the megamodel and views are contained in the corresponding system parts that are responsible for the maintenance of the contained

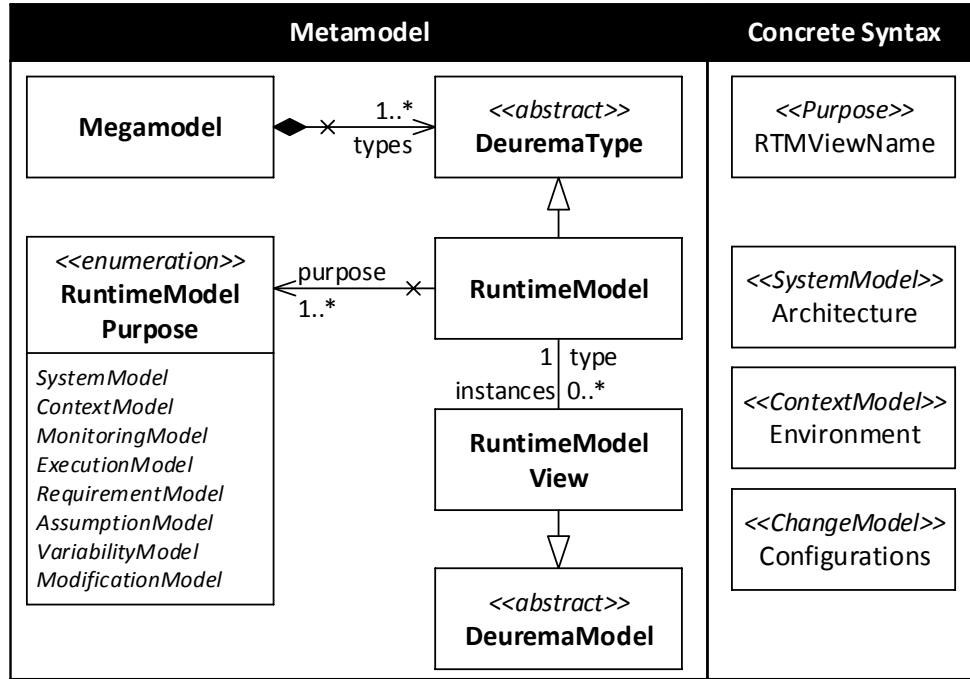


Figure 5.14: Deurema runtime model

views. Pinpointing the responsibilities and the amount of knowledge using runtime model views allows further analysis of knowledge distribution throughout the overall SoS. In the example of the former section, the feedback loop operates on such a local runtime model view, which is restricted to one smart car. If multiple smart cars want to share its local information with other smart cars, they must collaborate with each other exchanging the information of interest. Deurema does not restrict the content of a runtime model and thus, supports arbitrary domains as long as the runtime model content is defined by a corresponding metamodel. The metamodels for the runtime model content are typically application-specific and therefore not discussed in detail. In general, Deurema allows arbitrary metamodels that facilitates the use of the modeling language for a broad range of self-adaptive systems.

Concrete Syntax

The concrete syntax of a runtime model view is a rectangle that includes the purpose as stereotype and a name as shown at the right in Figure 5.14. In the example, a runtime model view named *Architecture*, which is defined as system model, is shown. Runtime model views can be contained in system parts that specify the local system behavior such as module templates (cf. Section 5.3) or collaboration interactions (cf. Section 5.5.3). The distinction between runtime model and view is important for the analysis, execution, and simulation of Deurema models. However, for the modeling of knowledge usage in module templates and interactions there is no difference so that the term *runtime model* is equally used with its corresponding views for the rest of this thesis unless the distinction is explicitly necessary.

5.2.5. Runtime Model Summary

Figure 5.15 summarizes the dependencies between the runtime model categorization as well as the use of runtime models and corresponding views in Deurema. First, the runtime model

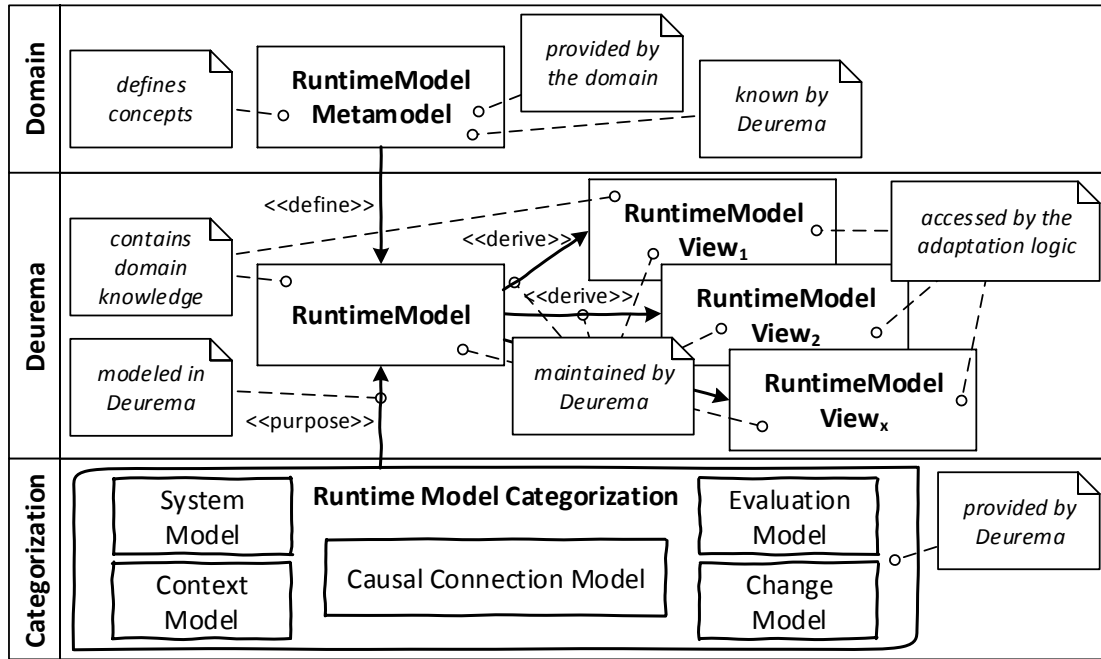


Figure 5.15: Deurema runtime model summary

in the middle layer in Figure 5.15 contains the domain knowledge with respect to the key points of interest. Arbitrary views can be derived from the runtime model content, which is directly modeled within the Deurema language. Furthermore, the adaptation logic, which is also modeled in Deurema, performs its functionality on the derived views only. Thus, runtime models, views, and the derivation of the views are directly supported and maintained by Deurema. Second, as shown at the top in Figure 5.15, the domain specific content of the runtime model is defined by a corresponding runtime model metamodel. Because the metamodel is domain specific, it must be provided by the corresponding domain, but is known by the Deurema modeling language. Third, the runtime model categorization provides the purpose of the runtime model that has to be specified in Deurema for each runtime model. The purpose gives insights into the contained content on a higher layer of abstraction, which enables reasoning about the runtime models and their used views. The runtime model purpose is orthogonal to the concept definition of the metamodel and must be appropriately set from the system developer according to the definitions in the categorization. Fourth and finally, the Deurema megamodel maintains the runtime models and their derived views. Beside the handling of runtime models that represent the available knowledge in the adaptive SoS, the modeling of the system behavior is an important core concept, which is comprehensively discussed in the following.

5.3. Deurema Module Templates

Up to this point, the former Section 5.1 introduces the Deurema concepts allowing the specification of a layered system architecture, whereas the corresponding system template description and system instances are maintained in the Deurema megamodel. Furthermore, Section 5.2 highlights the core concept of knowledge representation within an adaptive SoS

by using the MART approach. Thereby, all runtime models and corresponding derived views are maintained by the Deurema megamodel, too. Beside the layered system modeling and the specification of the knowledge, the modeling of the internal system behavior is important. The Deurema modeling language specifies aspects of the adaptive system behavior in *module templates*. A module template encapsulate a self-contained behavioral aspect of the system, e. g., a feedback loop, which operates on the local available knowledge base represented by the local runtime model view definition. Deurema supports different variants of modeling the local, internal behavior of the system by offering different module template types. For using the behavioral template specification within the layered system architecture, module templates must be instantiated and the template instances, named *modules*, can be placed on the layers of the system template specification.

Figure 5.16 shows a sketch of the smart city running example system template specification. The different variants of module instances are highlighted in gray in the figure, whereas each type is comprehensively described in the following. Thereby, the highlighted modules in Figure 5.16 are stepwise modeled using the Deurema approach.

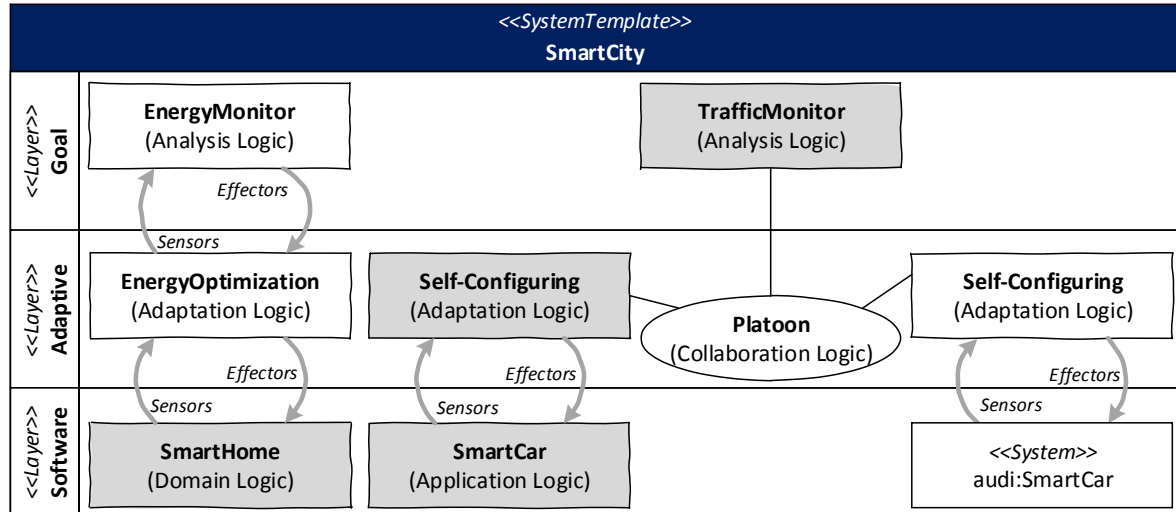


Figure 5.16: Smart city running example: Deurema modules

As depicted in the metamodel in Figure 5.17, module templates directly inherit from the DeuremaTemplate class. Thus, templates as well as their internal behavior specifications are contained in the megamodel. On the one hand, module templates can be considered as blueprints that describe a part of an adaptation aspect. On the other hand, they are designed to follow specific modeling approaches from different domains. Deurema supports four module template types that are *software*, *feedback loop*, *behavior*, and *application module templates*.

Software modules are considered as black boxes that support the specification of legacy adaptation behavior or parts of the adaptation engine, where the internals are unknown. However, software modules can be useful in early phases of the specification of the adaption logic in Deurema. Because they hide internals, they can be used to model the architectural structure of the adaptation engine and the trigger dependencies between modules without specifying all collaboration details and the concrete adaptation steps. Furthermore, even without those details, an analysis of the system architecture and module dependencies is possible. Additionally, simulation runs may help understanding how the interplay and the

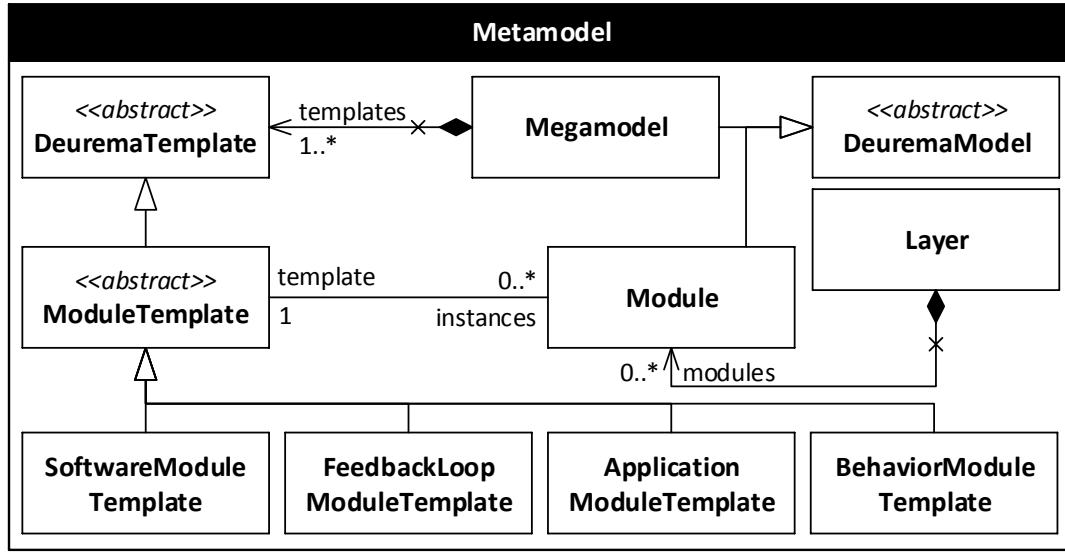


Figure 5.17: Deurema module templates

architectural design of the adaptation engine work and if those are able to fulfill expected requirements.

Feedback loop module templates capture adaptation activities, the intra-loop coordination, and the usage of runtime models. This module type is originally introduced by the Eurema modeling language as discussed in the former Section 2.3. In Deurema, feedback loop modules are extended by inter-loop coordination aspects, which enable a separation of concerns between local adaptation behavior and global collaboration activities.

Application module templates are designed with respect to the component-based development approach. Components in software engineering are used to foster separation of concerns by encapsulating resources and functionalities in well-defined parts of the systems. Therefore, components support a modular and cohesive development of a software system, which follows well established design principles. Furthermore, components can be assembled via well-defined interfaces to realize higher level functionalities. Especially with the focus on this thesis on adaptive SoS, which are further emerged from all kind of embedded and cyber-physical systems, a component-based specification of the adaptation logic helps integrating the Deurema modeling language in such domains. Examples for a dominant software development following component-based approaches are the robotic domain [47, 48] and the automotive industry with the AUTOSAR standard [62, 105].

Finally, behavior module templates are designed with the focus on the declarative description of the adaptation logic in form of graph transformation rules following the modeling of combinations of trigger-action conditions. The behavior module concept in Deurema is motivated by the fact that almost all models in software and systems engineering can be represented as graphs. Different domains use all kind of different model types to describe the structure, behavior, interface, deployment, requirements or context of a system. For example, timed and hybrid automata, petri nets, or state machines are well suited to describe timing, concurrency, and physical aspect of embedded systems as well as cyber-physical systems [126]. Furthermore, the UML defines various model types covering a broad range for modeling software system aspects such as activity, sequence, class, and object diagrams. However, almost all models created by these different modeling techniques and diagrams from

various domains can be represented in form of a graph structure. Therefore, working with these models and analyzing them could be done by using graph manipulation and reasoning techniques. Furthermore, Deurema keeps those models alive and represents them as runtime models defining the available knowledge in the system. With this motivation in mind, behavior modules provide the powerful concept of graph transformation rules as introduced in the preliminaries in Section 2.2.5 describing the adaptation logic and enabling the integration of a large range of model types from various domains.

In summary, all module template types are designed with the focus on a specific development approach or techniques that allow the integration of existing modeling approaches into the Deurema modeling language. On the one hand, Deurema can be used to specify the adaptation logic in such systems without learning completely new concepts and applying already known modeling concepts. On the other hand, integrating those different concepts into one modeling language that is still analyzable and can be used for simulation is very challenging. Furthermore, Deurema allows the usage of all different template types in the overall modeled SoS, seamlessly integrates the adaptation knowledge in form of runtime models throughout the different templates, and supports the interaction as well as triggering between different template types.

Module templates are deployed by means of module instances on a system layer in a corresponding system template, which denotes the availability of the encapsulated functionality and defines the internal behavior of the system. Furthermore, placed module instances can be executed, which enables a simulation of the system behavior. The corresponding module instance refers to its template description, whereas the model executor follows the modeled behavior in the blueprint specification. Consequently, there can be several module instances in the system that refer to the same template, but may be different concerning their assigned variables and execution state. Thus, modules and their corresponding template definitions follow the same design principle as discussed for system templates and system instances in Section 5.1. First, templates may contain variables that have to be resolved at deployment. Therefore, two module instances can have a different configuration of template variables at runtime. Second, module instances are considered as independent entities in Deurema that pass through different states during the lifetime of the adaptive SoS. This is similar to object instances, which are defined by a common class in an object-oriented programming language, but can have different variable assignments and thus, different internal states. Consequently, module instances contain individual runtime model information in their local view, which leads to different internal states, although the general adaptation logic is defined in the corresponding module template.

In the following, the variable concept for module templates, their responsibility for runtime model views, each template type, and the triggering of modules are discussed in detail.

5.3.1. Template Variables and Runtime Model Views

As mentioned before, templates are different to Deurema types with respect to the use of variables. This is reflected in the metamodel in Figure 5.18, whereas module templates contain an arbitrary number of variables. Furthermore, each variable has a type, which is specified by a reference to an abstract class `VariableType` in the metamodel. Each module template offers different variable types by refining the `VariableType` class and therefore, provides a set of individual variable types that fit to the design approach of the template type. For example, a feedback loop module template contains adaptation activities that operate on runtime models following the MAPE design approach. An appropriate variable type for this template type

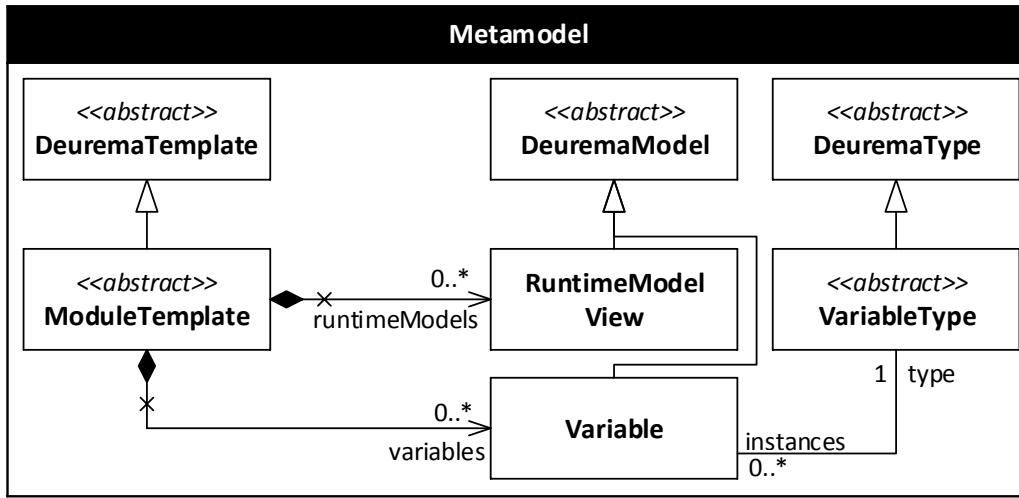


Figure 5.18: Variables and runtime model views in module templates

is an activity variable type, which is a placeholder for different adaptation strategies. At deployment of the module template, which means that a module instance is created and placed on the system template definition, the variable can be replaced by one variant of a concrete adaptation step. The Deurema variable concept is the basis for reconfiguration capabilities of a module, because the well-defined configuration points (variables) can be assigned with different concrete values at runtime. In general, the variable concept allows the distinction between predefined reconfiguration and arbitrary adaptation steps as described later in Section 5.6.

Beside variables, module templates maintain individual runtime model views as indicated by the runtimeModels containment reference in the metamodel in Figure 5.18. Views are defined by applying model queries on the corresponding global available runtime model information. Therefore, Deurema allows the individual specification of those parts from the global available knowledge that are key point of interest for each module template. Furthermore, global available knowledge, which is maintained by the megamodel (cf. Figure 5.14), is decoupled and becomes local knowledge in the corresponding module template. Each module is responsible for maintaining the view, distributing information to local adaptation activities and synchronizing the access on the local view as individually discussed for each template type in the following.

5.3.2. Feedback Loop Module Template

The first discussed module template type determines the modeling of the adaptation logic within the system as shown for the smart city example in Figure 5.19. Feedback loop module templates are originally introduced by the Eurema modeling language for modeling the adaptation activities of one feedback loop within the adaptation engine. The Deurema modeling language follows the external adaptation approach and considers the proposed four different adaptation activities forming the MAPE feedback loop of the reference architecture from Kephart et al. [110] as introduced in the preliminaries in Section 2.1.1. Therefore, the highlighted adaptation activities in the sketch in Figure 5.19 show a possible realization of such a feedback loop. In Deurema, the modeling of a feedback loop comprises individual adaptation activities, the intra-loop coordination of those activities, the specification of local available knowledge in form of runtime models, and the manipulation of runtime models by

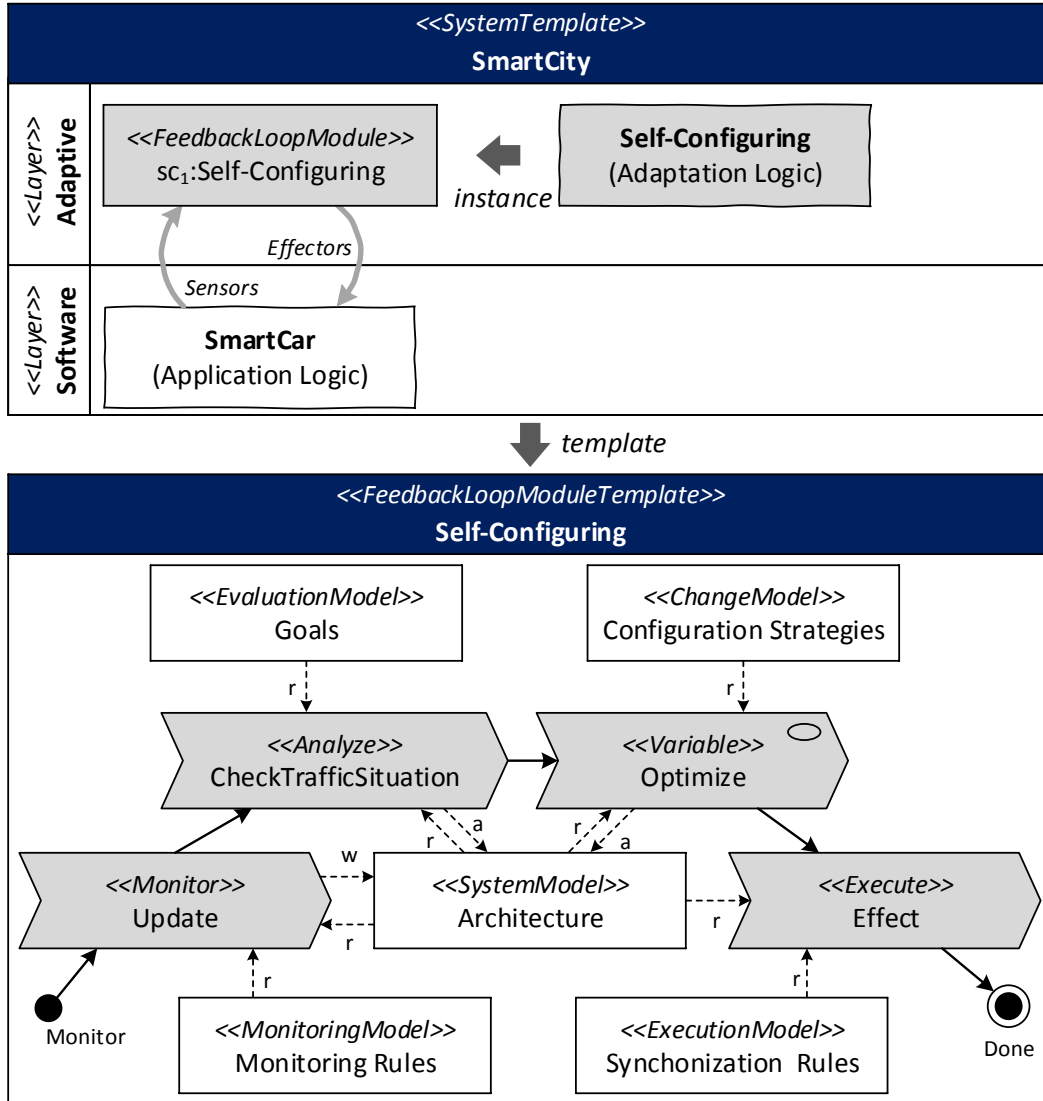


Figure 5.19: Smart city running example: Deurema feedback loop template

adaptation activities via model operations. In the following, these concepts are introduced and the corresponding excerpt of the Deurema metamodel is explained.

As shown in the metamodel in Figure 5.20, the Deurema modeling language adopts the possible operation types for modeling the feedback loop behavior from Eurema that are *initial*, *decision*, *final*, *destruction*, and *activity node*. Furthermore, operations are directly contained in the feedback loop module template. An initial node is the starting point for the feedback loop. Decision nodes branch the control flow by evaluating their guard conditions. Activities define the adaptation steps, which form the adaptation capabilities of the feedback loop. Final nodes complete the execution of a feedback loop, whereas destruction nodes destroy the availability of the feedback loop functionality. Normally, feedback loops are periodically executed starting at an initial node, performing the activities, and ending at a final node. If the last node of the feedback loop is a destruction node, the feedback loop is removed from the modeled adaptive SoS behavior, which can be used to model one-shot adaptations.

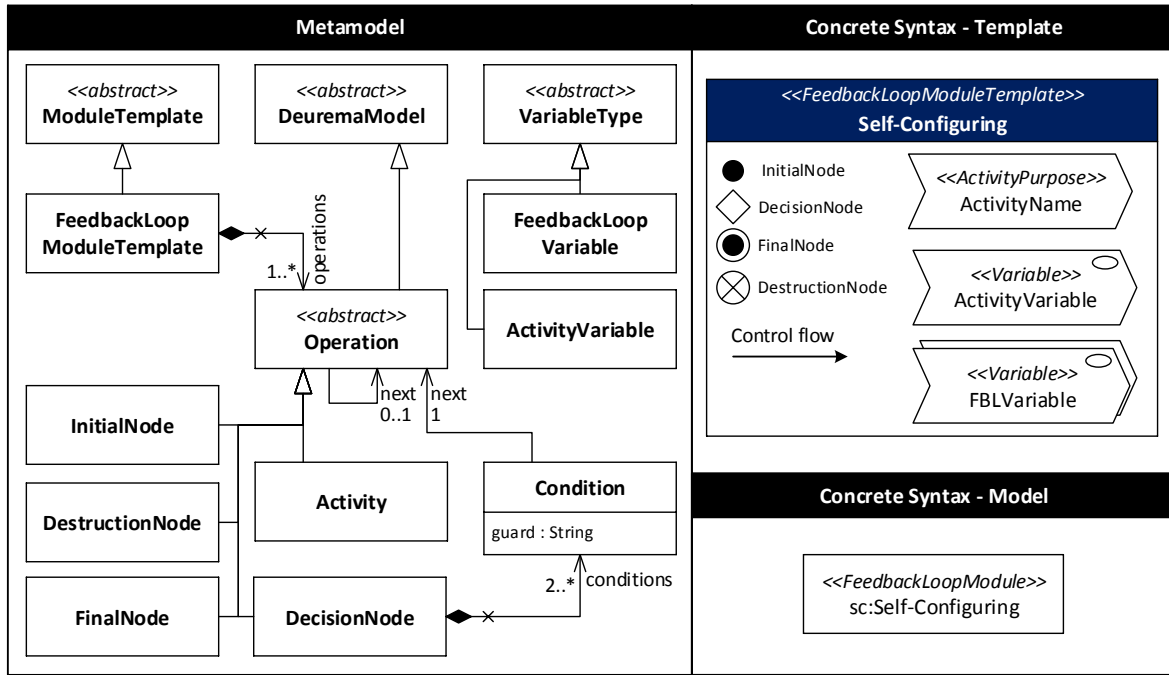


Figure 5.20: Deurema Feedback Loop Diagram (FLD)

As emphasized in [175], one-shot adaptation loops are useful to apply updates (patches) to the adaptive system behavior, which are usually developed offline and applied to the SoS afterwards.

One difference to the Eurema modeling language is the use of complex activities that are now replaced by the Deurema variable concept. Eurema uses complex activities to aggregate multiple adaptation activities into one complex activity, which enables a decoupled development of different aspects of the feedback loop. However, the internals of the complex activity must be deployed at development time in Eurema, whereas the aggregated behavior becomes visible in the feedback loop. Complex activities are replaced by the more powerful Deurema variable concept because of the following reasons. First, variables are placeholder in the feedback loop, which allows the same decoupled development of different variants of the feedback loop behavior as the complex activity concept in Eurema. Second, variables must be assigned during the instantiation of the corresponding module template, which resolves the placeholder with a concrete part of an adaptation behavior. Third, a complex activity is very specific for the use in feedback loop templates, whereas the variable concept can be used across other template types in Deurema. Fourth, variables are also used during system reconfiguration, where the adaptation behavior can be replaced during the lifetime of the overall SoS, which is supported as first class concept in Deurema. In summary, the Deurema variable concept can be equally used across all Deurema template types as well as it enables a generic handling of the reconfiguration possibilities in the modeled adaptive behavior.

As depicted in the metamodel in Figure 5.20, Deurema distinguishes between feedback loop variables and activity variables. The former allow the replacement of the variable by different feedback loop configurations, whereas the latter restrict the reconfiguration to single activities.

Concrete Syntax

The concrete syntax of feedback loop module templates is modeled in Feedback Loop Diagrams (FLD), whereas the concrete syntax of the elements is depicted on the right in Figure 5.20. For better recognition, the different node types are similar to the UML notation. Additionally, nodes are labeled with a name, which is used to identify correct entry and exit points of a feedback loop. For example, a feedback loop may have two initial nodes starting with different adaptation activities afterwards. If the feedback loop is triggered, the name of the initial node must be specified, which corresponds to desired starting point of the feedback loop. This is similar to destruction and final nodes, where the name denotes the specific end point of the feedback loop behavior.

Activities in a feedback loop are depicted as hexagon block arrows. The control flow is modeled as arrow, which is defined by the next reference in the metamodel. Furthermore, decision nodes may have multiple branches, which are realized by the contained conditions. Variables are labeled with an additionally ellipse in the upper right corner of the hexagon block arrow, which is uniformly used for all variable types for the rest of this thesis. Feedback loop variables have an additional double border indicating the possible replacement by multiple activities. The instantiation of the feedback loop module template follows the Deurema instance notation as explained in Section 5.1 and has the additional stereotype «FeedbackLoopModule».

Modeling Adaptive Behavior in Feedback Loop Templates

In feedback loops templates, activities are the entities that realize the adaptive behavior and manipulate runtime models. Conceptually, the concrete adaptation behavior of a single activity depends on the problem that has to be solved and thus, is an application specific piece of functionality in the corresponding domain. Therefore, Deurema supports different variants of integrating this application specific behavior into the modeling of the overall feedback loop, namely as *black box*, *gray box*, or *white box*.

In general, Deurema considers every element as abstract model entity as outlined in Section 5.1. Furthermore, every element that is related to a domain specific piece of functionality, is also considered as a model, which is enriched by a behavior description visible (white box) or invisible (black box) in Deurema. Thus, Deurema considers those elements as behavior models, which is defined in a corresponding abstract `BehaviorModel` class in the metamodel in Figure 5.21.

In Deurema, each activity has a corresponding type. On the one hand, the activity type defines the purpose of the activity that follows the MAPE approach from [110] as introduced in the preliminaries in Section 2.1.1. Therefore, the purpose is defined by an enumeration in the metamodel. On the other hand, the activity type refers to its behavior description, which is realized by the inheritance with the `BehaviorModel` class. Thereby, the activity type is also considered as Deurema type element, which can be instantiated by an arbitrary number of activity instances. The advantage of separating activity instances and types is the possible reuse of the same piece of adaptation behavior (defined in the activity type), which must be specified once and may appear in form of an activity in arbitrary feedback loops afterwards. Thus each activity knows its type, which is indicated by the `type-instance` reference in the metamodel in Figure 5.21.

As mentioned above, the concrete behavior specification of an activity type is domain specific and thus, leaves the boundaries of the Deurema modeling language. However, Deurema uses a *trigger-action* specification mechanism to support different variants of including the domain behavior into Deurema models. Both, the trigger and the action are defined as attributes of

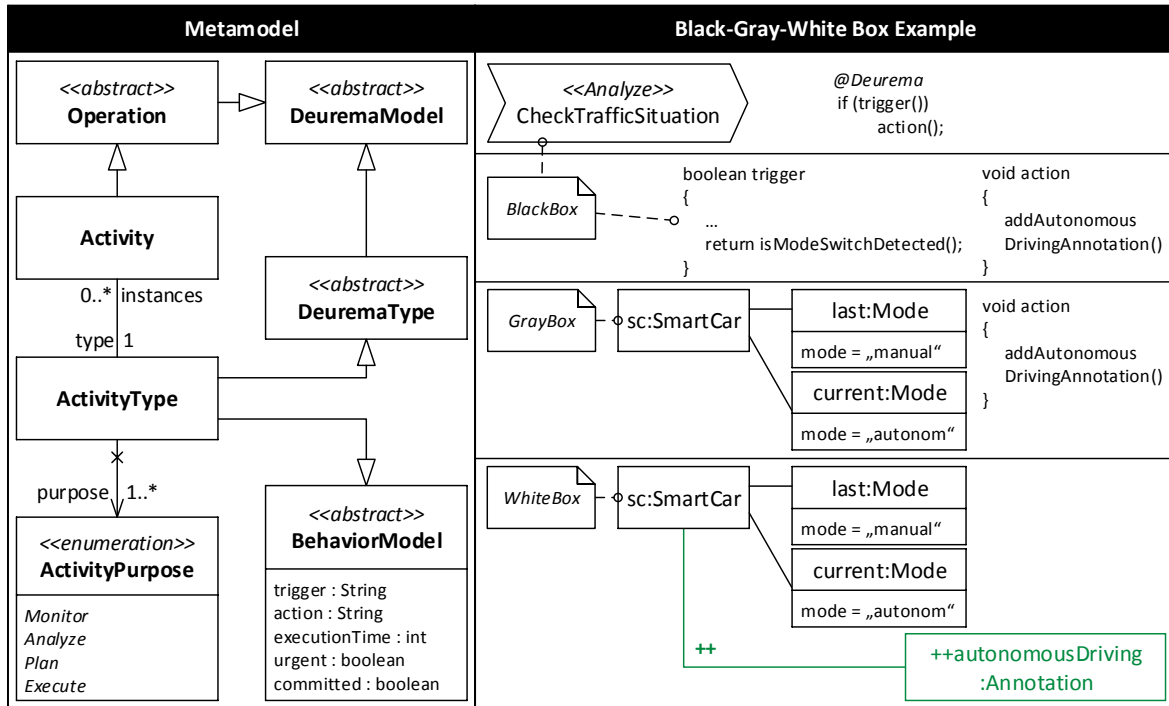


Figure 5.21: Feedback loop activities as behavioral models

the BehaviorModel class in the metamodel. The action refers to the concrete implementation of the behavior model that can be invoked by the Deurema interpreter. The activity type inherits the attributes from the BehaviorModel class with the consequence that Deurema is enabled to execute the specified adaptation effect. The optional domain trigger enhances the information about the activity type by specifying a trigger condition, which defines when the activity is allowed to perform its action. There are three possibilities if the feedback loop execution reaches a certain activity. First, there is no trigger condition specified, which indicates that no further restrictions are modeled for the activity. Therefore, the activity action is invoked by the interpreter, which leads to the implemented adaptation effect. Second, a trigger condition is modeled and the trigger condition is fulfilled, which has the same execution effect as in the first case. Third, a modeled trigger condition is not fulfilled, where as a consequence, the execution of the specified adaptation action is skipped. The last case ensures that adaptation effects are only performed within the feedback loop execution, if they are necessary.

There are two further variants how Deurema handles the specified trigger and action of a behavior model. Either the implementation internals are known by Deurema or they appear as black box by referencing a domain specific (unknown) implementation. The meaningful combinations are depicted in Table 5.2. In the following, the three combinations are explained in the context of feedback loops with a concrete activity example, which is depicted on the right in Figure 5.21. The example shows the CheckTrafficSituation activity from the self-configuring feedback loop of the smart car. The purpose of this activity is defined as analysis step denoted by the stereotype «Analyze». This activity decides if a mode switch from manual to autonomous driving must be performed as already comprehensively discussed in the runtime model example in Section 5.2.3. However, the following concepts of specifying domain specific

Table 5.2: Trigger and action combination for behavior models

	Black Box	Gray Box	White Box
Trigger	✗	✓	✓
Action	✗	✗	✓

✓: known by Deurema; ✗: unknown by Deurema

behavior hold for all Deurema elements that inherit from the behavior model class and are not specific for activity types.

The first combination in Table 5.2 is a black box activity, where the trigger and the actions refer to an internal, unknown implementation. Thereby, the specification of the trigger is optional as discussed above. Conceptually, a black box activity can be implemented as shown in the upper right in Figure 5.21. The pseudocode denotes that these details are not known by the Deurema modeling language. Therefore, there can be an arbitrary trigger and action implementation, whereas the former checks whether the activity actions is performed or not. Deurema will execute the black box domain trigger implementation first and in the positive case the black box action implementation afterwards. In the pseudocode example, the trigger is implemented as a function that checks for a detected mode switch. Furthermore, an action function adds a corresponding annotation in the runtime model that denotes the need of an adaptation for subsequent feedback loop activities (cf. also Figure 5.19).

The second combination shows a gray box activity, whereas the trigger condition is known by Deurema and the action refers still to an unknown implementation. In this case, "known by Deurema" means that the trigger condition is modeled as graph pattern. In general, Deurema directly supports the specification of behavior in form of graph transformation rules, which further enables the reasoning about the modeled behavior. The concept of modeling behavior with graph transformation rules is already introduced in Section 2.2.5. In the case of a known trigger condition, a declarative graph pattern must be specified that has no side effects (the LHS must be equal to the RHS of the graph transformation rule). In the example on the right in Figure 5.21, the trigger condition comprises a graph pattern with three objects. The pattern describes a *SmartCar* that has information about its last and current mode, whereas the last mode was the manual driving task and the current mode describes the need of an autonomous driving behavior. Thus, the pattern detects a mode switch from manual to autonomous driving. Deurema can directly execute this graph pattern, which is the searching for an appropriate match in the available local knowledge base. If a match is found, the black box action is invoked by the Deurema interpreter, which leads to the same side effect as in the first combination described above. In contrast to the black box trigger (the pseudocode above in the figure), the trigger condition modeled as a pattern is visible and maintainable in the context of the Deurema modeling language. Thus, the black box activity turns into a gray box activity due to the additional insides, whereas the trigger condition is known and can be efficiently handled by the Deurema interpreter, but the effect in form of the action remains as application specific black box implementation.

The third combination in Table 5.2 denotes a white box activity, whereas the trigger condition as well as the action are known by Deurema and specified as graph transformation rule. After searching for a match by executing the LHS of the graph transformation rule,

the action is performed by applying the side effect of the RHS of the rule. The example in Figure 5.21 shows a graph transformation rule, where an `Annotation` object is created that denotes the need of an adaptation to a new driving mode of the smart car.

In summary, Deurema considers feedback loop activities as behavior models that follow the black-gray-white box concept depending on the visibility of the domain trigger condition and performed adaptation action. Whereas black box behavior can encapsulate an arbitrary domain functionality, white box behavior is specified in form of graph transformation rules. In general, all behavior models can be executed by the Deurema interpreter and thus, can be integrated into a simulation of the overall adaptive SoS behavior. Furthermore, for white box behavior models (e. g., activities) the domain specific internals are known and can be efficiently handled by Deurema. With respect to the execution and simulation, behavior models have timing properties that are an execution time, urgent, or committed property shown in the metamodel in Figure 5.21. The Deurema simulation framework supports different strategies. In early development phases, simulation may ignore the execution time of a behavior model (zero execution time semantic), where the overall effects of the feedback loops are investigated to ensure that the modeled control algorithm works as expected. Later, the execution time can be measured, e. g., for black box behavior models, or even predicted for white box models, if further details of the execution platform are known.

Runtime Models in Feedback Loop Templates

The last missing part of defining the adaptive behavior in feedback loops in Deurema is the integration of runtime models in the feedback loop templates defining the local available amount of knowledge on which the adaptation activities can operate on. As shown in Figure 5.22, each activity defines model operations that link to a corresponding runtime model view, which is contained in the same template definition as the activity itself. A model operation is specified by a type and arbitrary knowledge queries represented by the `ModelOperationQuery` class in the metamodel. The model operation type defines the kind and therefore the direction of runtime model access, whereas Deurema supports `read`, `write`, `annotate`, `create`, and `destroy`. Reading a runtime model retrieves data, which has no side effect on the knowledge base. Therefore, the data flow of a reading model operation is from the runtime model view to the activity, which is depicted by a dashed arrow in concrete syntax in the Self-Configuring feedback loop template in the middle of Figure 5.22. All other model operations manipulate the runtime model, which leads to a modification in the available knowledge base. Consequently, modifying a runtime model implies a data flow from the activity to the corresponding view as shown for the writing operation in the concrete syntax example in the figure. Without loss of generality, all read model operations are applied before activity execution (step (1) in the figure). The retrieved knowledge can be freely used and changed by the activity during its execution (step (2) in the figure).

The Deurema model query concept is inspired by graph based query languages such as SPARQL [63] or EMF-IncQuery [171]. In general, such query languages define the desired amount of information declaratively, whereas the query can be effectively applied on the underlying graph database [186]. Because runtime models are considered as labeled graphs, model operation queries can be used to define the information of interest. Therefore, queries are considered as Deurema behavior models and thus, as executable elements that retrieve the concrete data from the runtime model view as needed by the activity. In general, the amount of accessed knowledge is defined by those executable queries in Deurema, whereas the direction is specified by the corresponding model operation type. Because a model query is a behavior model, it follows the same black-gray-white box idea as explained for activities above.

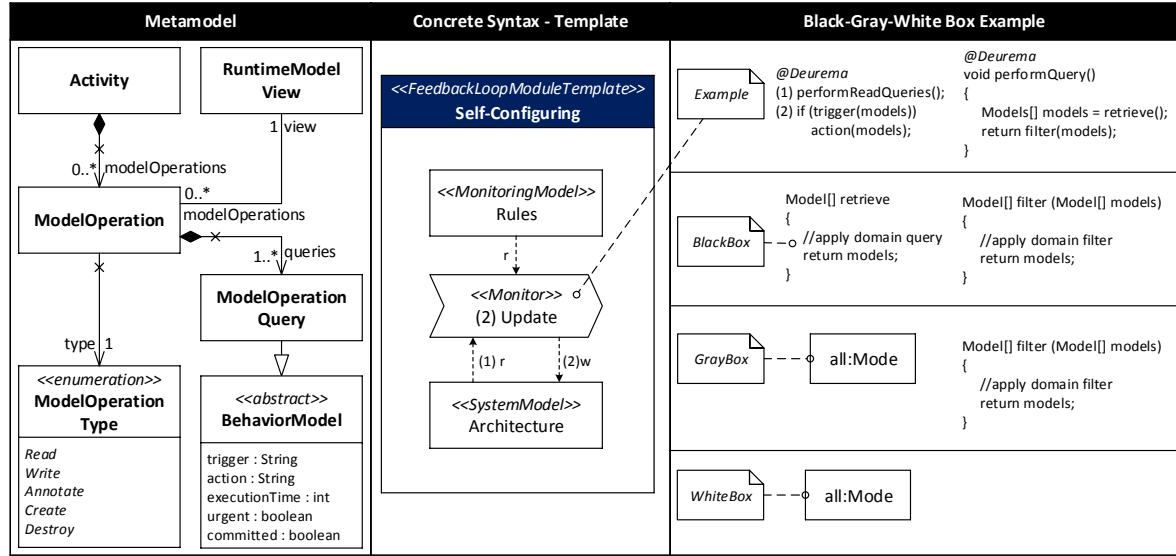


Figure 5.22: Deurema FLD runtime models

The trigger and action attributes from the **BehaviorModel** class define the implementation of a model operation query, which is further named as *retrieval* step (trigger attribute) and an additional *filter* step (action attribute). The retrieval step is the application of the query on the runtime model, which returns the specified amount of information. The optional filter step can be used to prepare the data for its execution by the activity, which might include transformation of data formats or a preprocessing that cleans the data. Therefore, the retrieval of data has no side effects on the knowledge base, whereas the filter action may change the runtime model data that can be noticed as additional side effect.

Straight forward, a black box query is characterized by an unknown, domain specific retrieval and filtering of data as shown on the right in the example in Figure 5.22. For a gray box query, the retrieval is performed by the Deurema modeling language, whereas the corresponding query is defined by a declarative graph pattern (graph transformation rule). In the example, the pattern consists of a single **Mode** object, which denotes that all nodes with the type **Mode** are retrieved from the knowledge base. The retrieval of information corresponds to finding all matches according to the LHS of the graph transformation rule (query). If this query belongs to the step (1) in the example in the middle of Figure 5.22, the query is performed on the **Architecture** runtime model with the effect that all **Mode** objects (matches) are retrieved. In contrast to a white box query, a gray box query has an additional, domain specific filter action. The filter is performed on the retrieved matches and may have side effects that are not known for Deurema. A white box query has no additional filter operation and thus, the query consists of declarative graph pattern with no side effects as shown in the figure. Each gray and white box query consists of exactly one graph pattern. Therefore, multiple model queries can be combined to describe the overall model operation, which is performed by Deurema on the corresponding runtime model.

Because feedback loop activities and model queries inside the model operations are considered as Deurema behavior models, there are several possible combinations of specifying black-gray-white box behavior. Table 5.3 shows the two dimensions with respect to the reasoning about the domain functionality defined by adaptation activities and the access to the domain knowledge defined by model operation queries. In general, the more information is known by

Table 5.3: Domain knowledge and domain functionality dimensions

		Reasoning about Domain Functionality →			
		Behavior Model			
		Black Box	Gray Box	White Box	
Reasoning about Domain Knowledge ↓ Model Operation Query	Black Box	✗ trigger ✗ action ✗ retrieve ✗ filter	✓ trigger ✗ action ✗ retrieve ✗ filter	✓ trigger ✓ action ✗ retrieve ✗ filter	unknown amount of knowledge access
	Gray Box	✗ trigger ✗ action ✓ retrieve ✗ filter	✓ trigger ✗ action ✓ retrieve ✗ filter	✓ trigger ✓ action ✓ retrieve ✗ filter	known knowledge access but unknown filter action
	White Box	✗ trigger ✗ action ✓ retrieve ✓ filter	✓ trigger ✗ action ✓ retrieve ✓ filter	✓ trigger ✓ action ✓ retrieve ✓ filter	known amount of knowledge access
		unknown domain trigger and action	known domain trigger	known domain trigger and action	

✓: known by Deurema; ✗: unknown by Deurema

Deurema the merrier are the analysis capabilities to reason about the adaptation effects of the modeled feedback loops. For a better understanding, the handling of two extreme cases are described. In the first case, the availability of no domain specific behavior information by using black box queries and activities is discussed. In contrast, for the second case, all information is modeled in Deurema using white box queries and activities. Both cases are described by considering the step (1) performing the read model operation on the Architecture runtime model and executing the Update monitoring activity afterwards as shown in the middle of Figure 5.22.

For the complete black box case, all model queries for the read model operation are executed first. Thereby, the Deurema interpreter subsequently invokes the black box retrieval and filter operation for each query, where the results are collected. Afterwards, the black box trigger from the adaptation activity is invoked on the aggregated, retrieved models. The trigger decides whether the adaptation action is invoked or not. In the positive case, the retrieved models are transferred to the domain specific adaptation action. Because the specification of the domain trigger is optional, the retrieved models are directly hand over to the domain action in the case that no trigger is modeled. The black box implementation of the activity uses the runtime model information during its execution and performs arbitrary modifications on them, which are tracked by Deurema. Because the internals of the data retrieval as well as the adaptation effect are unknown, Deurema can neither predict the amount of used

knowledge nor reason about the manipulations done by the activity during the specification (development) of the adaptation logic. However, Deurema can monitor the knowledge access and manipulation by executing the black box queries and the activity, which enables the reasoning about the modeled feedback loop behavior during simulation.

For the complete contrary case, where all internals are modeled within the Deurema modeling language by means of graph transformation rules, the general execution order remains the same, but executing the query as well as the activity do not leave the border of the Deurema modeling language. Therefore, the execution of the model operation queries for step (1) is realized by searching for an appropriate match of the modeled graph pattern in the corresponding runtime model. According to the example, all *Mode* objects are retrieved from the *Architecture* runtime model. The matches are collected for each query, whereas a white box query has no additional side effect (filter operation). Afterwards, the LHS of the activity, which is also modeled as graph transformation rule, is executed. If a match on the retrieved runtime models is found, the adaptation effect is performed by the application of the RHS of the corresponding graph transformation rule. According to the example in Figure 5.21, an *Annotation* object is created on the beforehand found match. Consequently, the complete behavior is known by Deurema at development time, which facilitates static analysis of the modeled adaptation behavior without performing a simulation. Of course, the adaptation effects can be monitored during simulation in the same way as for the black box case above.

For all other combinations in Table 5.3, the general execution order of queries and activities stays the same. The available information and therefore the static analysis capabilities of Deurema depend on the chosen black-gray-white box combination.

Feedback Loop Template Example

After all Deurema concepts for modeling a feedback loop are introduced, Figure 5.23 shows the template definition of the *Self-Configuring* feedback loop from the smart city example at the beginning of this section. The feedback loop runs on top of a smart car as sketched in Figure 5.19. The feedback loop template contains three activities and one activity variable that are arranged in the causal order *Update*, *CheckTrafficSituation*, *Optimize*, and *Effect*. Furthermore, each activity has a purpose modeled as stereotype. According to the modeled template, the monitoring activity *Update* performs a read on the *Monitoring Rules* as well as *Architecture* runtime model and updates the current architecture of the underlying smart car system afterwards. Subsequently, the traffic situation is checked by the corresponding *CheckTrafficSituation* activity, which reads the current goals and may trigger a need for an adaptation, e.g., by changing the driving mode as outlined in Section 5.2.3. The *Optimize* variable defines a placeholder for different strategies of adapting the architecture, whereas the *Effect* activity synchronizes the planned adaptations with the underlying smart car system.

Feedback loop module templates can be instantiated (deployed) in an arbitrary layer of a system template definition. For example, the *Self-Configuring* feedback loop is used twice as sc_1 and sc_2 module instance in the *SmartCity* system template in Figure 5.3. Both module instances follow the same template definition as explained above and thus, show similar behavior during execution of the activities. A variation point is the *Optimize* variable, which can be resolved differently during the deployment of the modules.

In summary, the Deurema modeling language determines adaptation activities and their coordination that describe the adaptive behavior of the feedback loop. Furthermore, feedback loops comprise the available knowledge in form of runtime models that are accessed by model operation. Variables are used to define placeholder for different variants of one adaptation activity. Finally, feedback loop templates must be instantiated to corresponding feedback loop

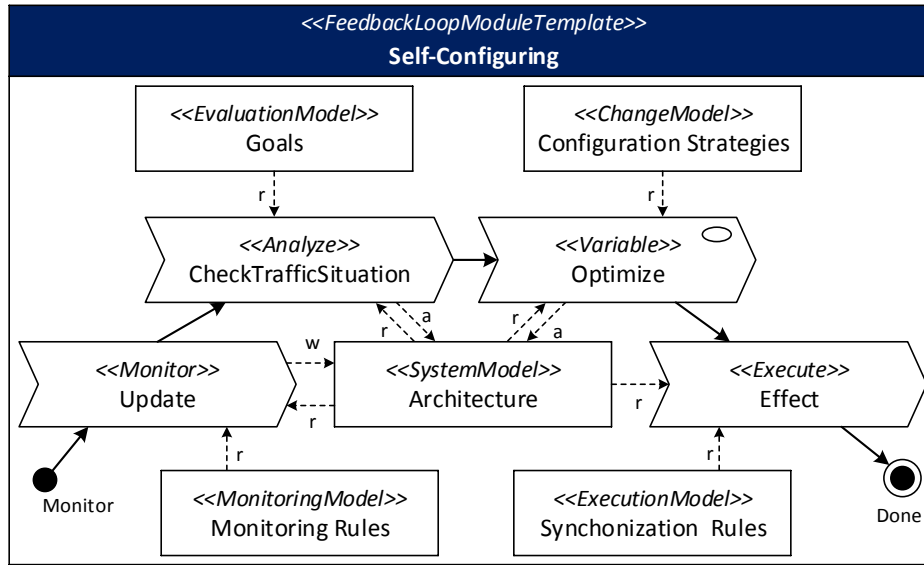


Figure 5.23: Deurema Self-Configuring feedback loop template

modules, which can be reused in the adaptive SoS architecture. Therefore, feedback loops are a core concept for specifying adaptation effects in the system. In the following, three additional concepts are introduced to define adaptive behavior within the SoS as well as to integrate different domain specific extensions into Deurema that are namely, *Software Module Templates*, *Application Module Templates* and *Behavior Module Templates*.

5.3.3. Software Module Template

Feedback loops define adaptive capabilities of the SoS that changing the overall system behavior according to given requirements. Thereby, the system structure, state, or behavior is represented as runtime models in the local knowledge base of the feedback loop. This section discusses an additional Deurema concept of defining domain specific behavior on module level and integrating it into the system template definition. Figure 5.24 sketches the refinement of the domain specific functionality of a smart home. The domain functionality appears as software module instance in the smart city system template definition. Furthermore, the module instance refers to a corresponding software module template description, which defines the behavior as trigger-action condition pinpointing to the domain specific implementation.

Software modules are the Deurema concept to integrate adaptive black box behavior as for example legacy software parts into the adaptive SoS specification. The former section already outlines, how black box behavior can be integrated into the Deurema language by means of black box adaptation activities within a feedback loop. Therefore, the software module template class in the Deurema metamodel inherits from the behavior model class as shown in Figure 5.25. Similar to black box activities, a software module template is defined by an optional domain trigger and a corresponding action that points to the implementation of the domain functionality. Furthermore, the trigger can be modeled as declarative pattern that defines a situation in the domain, which further restricts the execution of the black box behavior only for those cases, where Deurema finds an appropriate match in the knowledge base of the adaptive SoS.

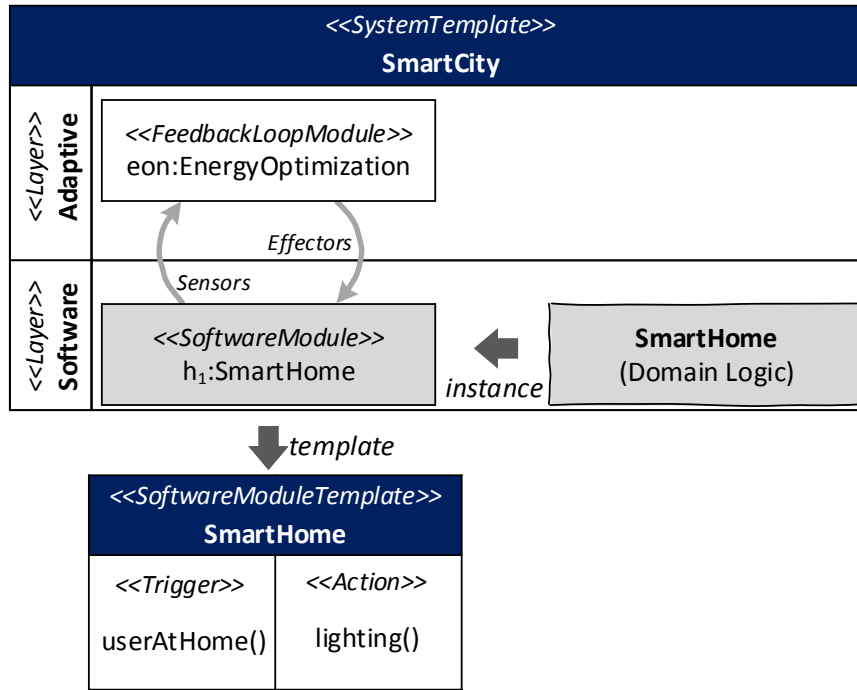


Figure 5.24: Smart city running example: Deurema software module template

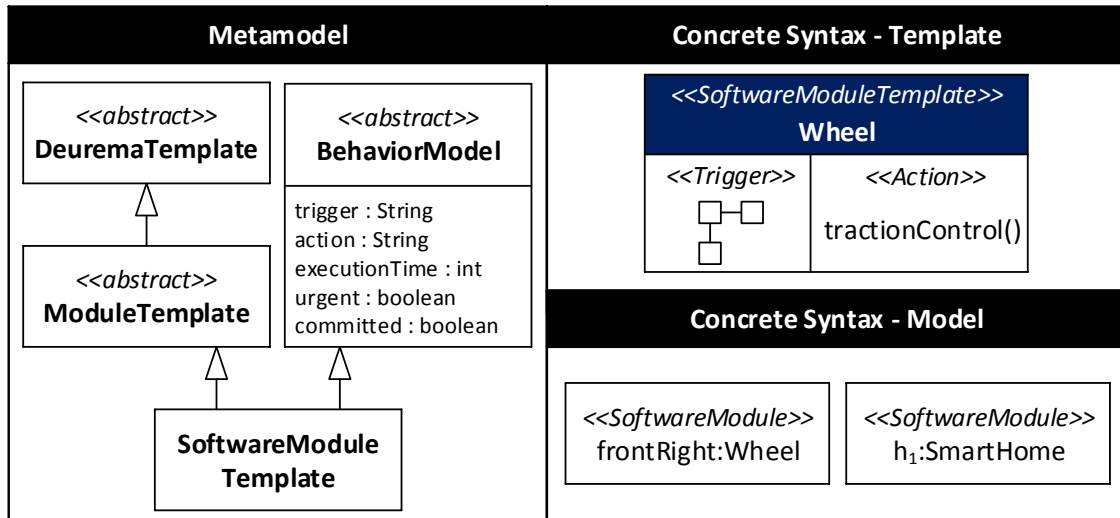


Figure 5.25: Deurema Software Module Diagram (SMD)

In contrast to the activity concept above, software module templates must refer to a black box implementation in the domain, which is not known in detail by Deurema. Thus, a white box specification for software module templates is not allowed, which would obviously contradict the intention of this concept. As a consequence, on the one hand, Deurema considers black box behavior on module level, which can be reused in several instances in the layered system architecture. On the other hand, the black box becomes a gray box, if an additional trigger condition is modeled as graph pattern similar to gray box activities as discussed above. However, the adaptation effect remains application specific in form of a

black box action, which can be invoked by the Deurema interpreter. Therefore, the Deurema execution environment decides at which point in time the software module is executed and can monitor, but not predict, the corresponding behavioral effects.

Concrete Syntax

Software module templates are modeled in Software Module Diagrams (SMD) as shown at the right in Figure 5.25. They follow the Deurema naming scheme and thus, are labeled with their name and the stereotype «SoftwareModuleTemplate». The optional trigger condition is modeled on the left, where the name of the corresponding action is specified on the right in the template definition. Furthermore, software module instances are modeled as rectangles in object notation that refers to the corresponding template type.

Software Module Template Example

Figure 5.26 shows a black box software module template on the left and the same module template with an additional trigger condition as gray box on the right. According to the running example of the thesis, the black box behavior of a wheel in the smart car is modeled, whereas the internals of the adaptation logic are not known. Both software module templates perform a tractionControl() action if they are executed by the Deurema interpreter. Furthermore, the template on the left has no trigger condition specified, with the consequence that no additional check for its applicability is performed.

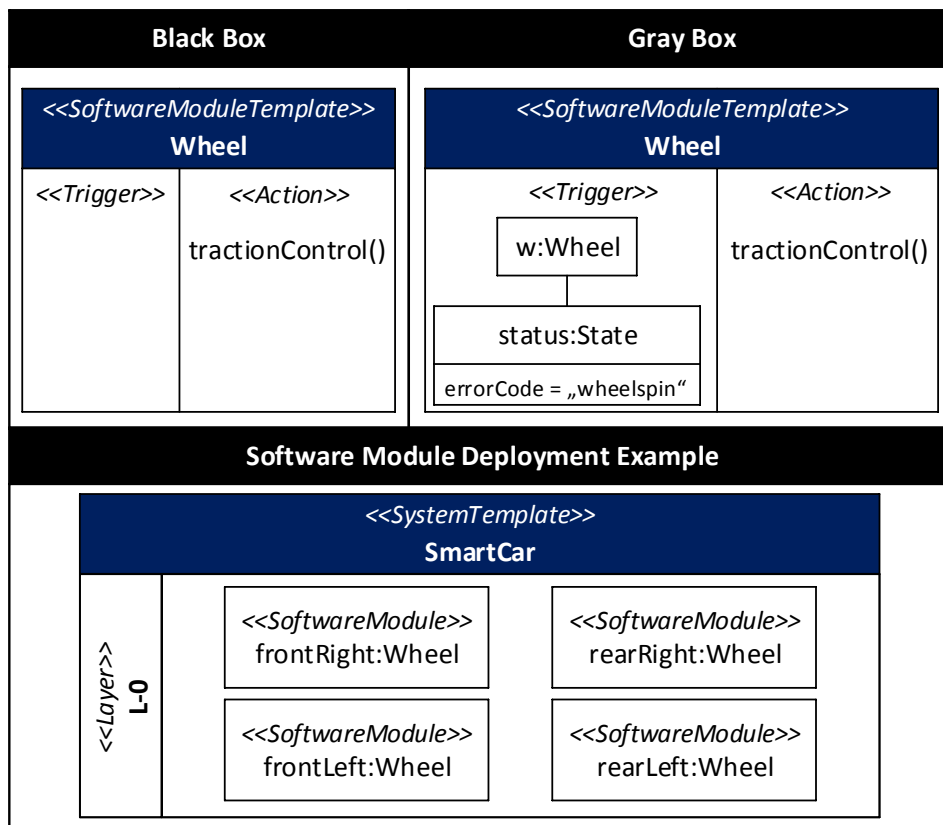


Figure 5.26: Deurema SMD example

In contrast, the software module template on the right specifies additional domain information by defining a trigger condition, which is in this example a pattern consisting of a Wheel and its corresponding State. Only if a wheelspin is detected, the traction control is necessary and therefore, invoked afterwards. Consequently, the software module template performs the modeled action only if the corresponding trigger situation occurs. In Deurema, the trigger can also be specified as black box implementation, which is not shown in the figure, but comprehensively discussed for adaptation activities within feedback loops in the former section.

At the bottom, Figure 5.26 shows an exemplary instantiation of the software module template at the top of the figure. According to each physical wheel in the smart car system, a Wheel software module is deployed at the layer L-0, where each module independently performs the traction control functionality. Thus the modeled template functionality is reused four times for defining the smart car architecture.

In summary, Deurema software module templates can be used to integrate domain specific black box behavior into the system template definition. Due to the concrete implementation is hidden, Deurema has only limited possibilities of analyzing the behavioral effects of software modules during the development. However, software modules can be executed and the effects can be observed during a simulation.

5.3.4. Application Module Template

Beside the integration of black box behavior modules, Deurema supports the explicit modeling of the system behavior. One possibility is the specification of application module templates considering the software component development paradigm, which is typical for embedded, robotic, and cyber-physical systems. Figure 5.27 sketches the refinement of the autonomous driving behavior of a smart car. Following the Deurema template-instance paradigm, the module template defines the internal behavior of the smart car application logic, which is for this module template type a component-based modeling approach. Furthermore, the template can be deployed in the smart city system template description as module instance refining the behavior of the overall system. Similar to the Deurema feedback loop modeling, application module templates contains a local knowledge base defined by runtime models. Thus, application module templates comprises the two concepts of component-based behavior modeling and the integration of knowledge by means of runtime models.

As shown in the metamodel in Figure 5.28, an application module template contains an arbitrary number of components, where each component has a type according to its purpose. Inspired by the AUTOSAR standard [62] and typical component types in embedded systems [14, 126], Deurema supports three component types, namely Sensor, Actuator, and SoftwareComponent (SWC). The component types are defined in the Deurema metamodel accordingly.

Sensor components are the encapsulated software representation of hardware sensors that are able to monitor the context or state of the system. Consequently, sensors capture runtime information, may prepare sensed data for further usage, e.g., convert analog values to a predefined range of digital values or clean data from sensor noise, and finally provide it in the corresponding runtime models. Software components usually process the provided data from the sensors and realize the analysis and planning of the adaptation logic. Furthermore, they may calculate control values that are read by the actuator components. In contrast to sensor components, an actuator component encapsulates hardware parts of the system that are able to physically influence the system or its context. Therefore, actuator components may

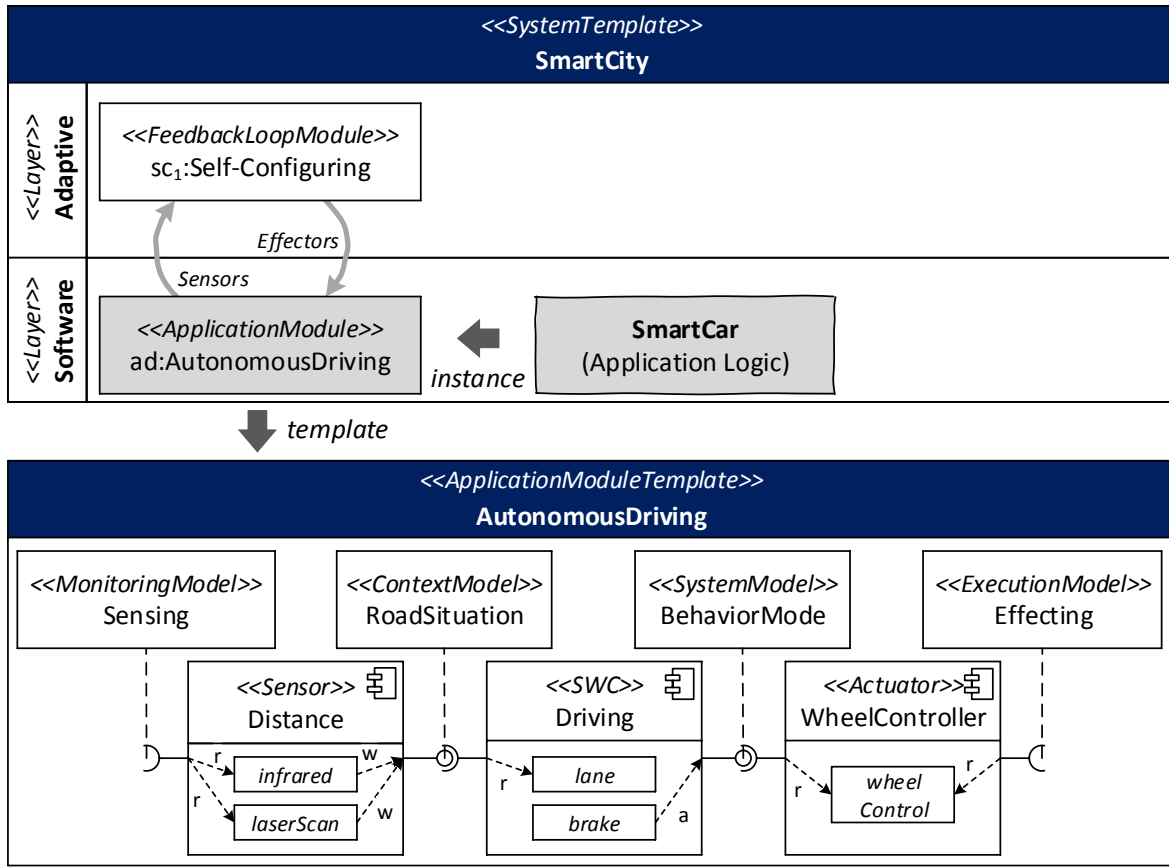


Figure 5.27: Smart city running example: Deurema application module template

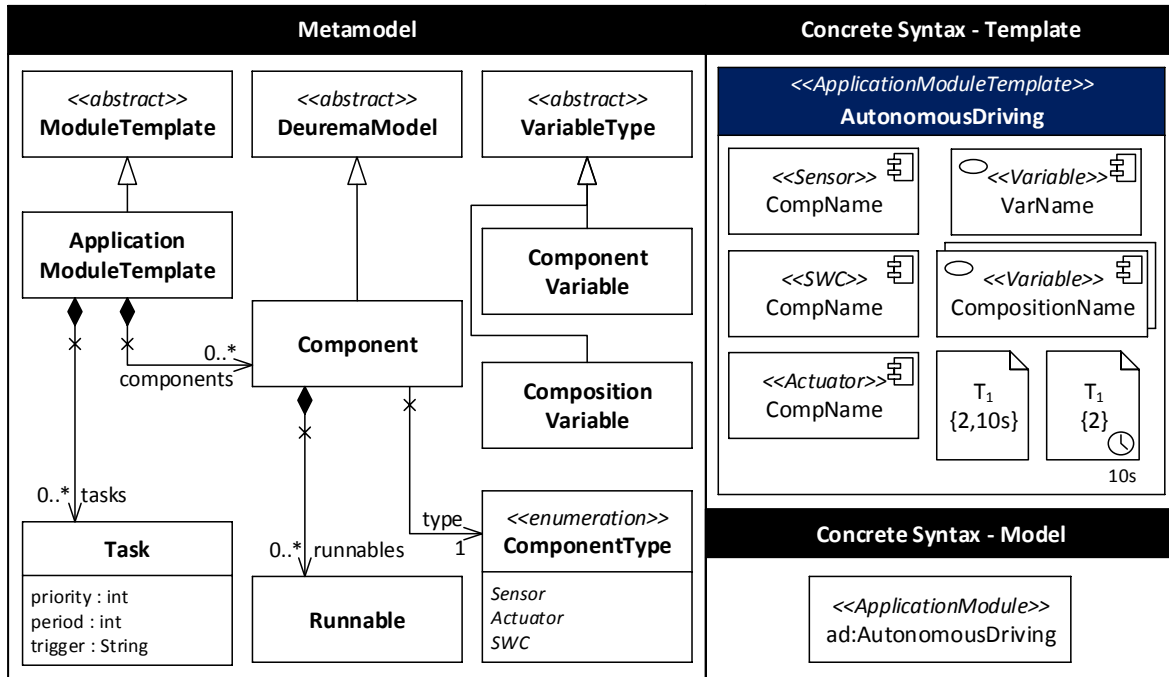


Figure 5.28: Deurema Application Component Diagram (ACD)

translate runtime model information to control commands for the physical execution platform, e.g., convert digital control commands to analogue values or derive a concrete sequence of execution commands for the underlying hardware.

With respect to the component architecture of an application module template, Deurema supports two types of variables. A component variable is placeholder for a single component, which allows the exchange of the complete component during system reconfiguration. Furthermore, a composition variable is a placeholder that enables its replacement by multiple components. Therefore, meaningful parts of the component architecture can be aggregated to a composition variable, which allows the dynamic replacement of the complete part of the system architecture.

Where components represent the structural architecture of an application module template, tasks are the basic behavioral units for scheduling. Tasks are characterized by the two properties priority and time, whereas the former defines the importance of the task and the latter the minimal waiting time between two applications of the same task. The task with the highest priority is executed first in Deurema, which can be used to group tasks according to their importance. Consequently, the execution order of tasks with the same priority is undefined. Additionally, the application of a task can be further restricted by an optional trigger condition. A domain specific trigger can be used to ensure that the task is only executed in a specific situation. The trigger can refer to a domain specific implementation, whereas the internals are unknown by Deurema, or to declarative graph pattern, which is known and maintained by Deurema. In the latter case, Deurema searches for a match of the modeled graph pattern, which enables the execution of the task. Thus, the handling of a trigger condition follows the same semantic as discussed for behavior models such as activities in feedback loops.

Finally, the functionality of a component is modeled in atomic behavioral units called *Runnables*.² For the execution of the adaptive behavior, the runnables, which are located in a component, must be assigned to a task. In the following, the concrete syntax as well as the modeling of adaptive behavior in application module templates is explained in detail.

Concrete Syntax

As shown on the right in Figure 5.28, application module templates are specified in Application Component Diagrams (ACD). Components are modeled as rectangles with an UML component symbol in the upper right corner. Furthermore, components have a name and are stereotyped with the corresponding component type that can be «Sensor», «SWC», or «Actuator». Following the Deurema notation guidelines, component variables have an additional ellipse symbol in the upper left corner and have the stereotype «Variable», whereas composition variables have a double border. Finally, tasks are the behavioral scheduling unit of application module templates and are modeled as rectangular notes. Tasks have a name and the two properties priority as well as period are modeled in curly brackets under the name. The period can be optionally modeled using a clock symbol in the lower right corner of the task. The concrete syntax of an application module follows the same design principles as for feedback loop modules, which is a rectangle in object notation that has the stereotype «ApplicationModule».

Modeling Adaptive Behavior in Application Component Templates

Components are containers that aggregate adaptation behavior and define the architectural static structure in Deurema ACD. A runnable represents a piece of functionality that realizes a domain specific adaptation behavior. Runnables are contained in components as shown

²The name *Runnable* is inspired by the AUTOSAR standard [62].

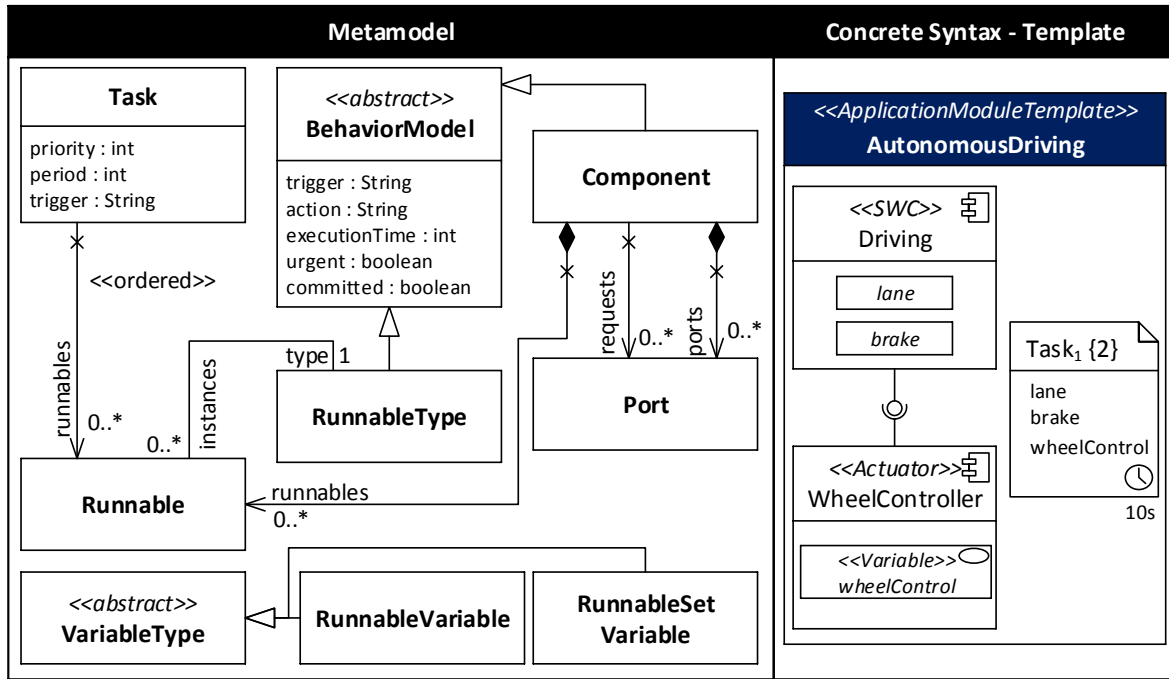


Figure 5.29: Deurema ACD runnables and ports

in the Deurema metamodel in Figure 5.29. Runnables are defined by a corresponding type, which encapsulates the behavior definition of the runnable. Therefore, Deurema considers runnable types as behavior models, which implies the same black-gray-white box semantic for runnables as discussed for feedback loop activities in Section 5.3.2.

Up to this point, components define the static architecture and runnables the behavior in an application module template. The execution sequence of runnables is specified by a task. Thus, runnables must be mapped to tasks, whereas the mapping order defines the execution sequence. Each runnable can be assigned to an arbitrary number of tasks and can be mapped multiple times. On the one hand, the separation of a runnable and its type definition allows the reuse of the same piece of functionality in different application module templates or within the template in different components. On the other hand, because each runnable can be mapped multiple times to a task for its execution, the same piece of functionality can be easily reused without changing the component architecture or the runnable type. Runnables can provide or request runtime models from ports, which are managed by the component. Ports are used for sharing runtime models with other components.

Note, because runnables are the basic unit of scheduling, Deurema does not support the direct execution of components. Components are container that group a set of available functionalities on an architectural level. Thus, the direct execution of black box (legacy) components is not supported, but can be alternatively modeled. A similar construct of a black box component in Deurema is a component that contains a single black box runnable, which must be further assigned to a task for an appropriate execution.

For reconfiguration purpose, a runnable variable is a placeholder for a single runnable, whereas the runnable set variable is a placeholder for multiple runnables. Therefore, the former is appropriate to exchange single function, whereas the latter can be used to replace a

whole function group. Both variable types are defined at the bottom in the metamodel in Figure 5.29.

At the right side of Figure 5.29, runnables within a component, the port concept, and assignment of runnables to tasks is depicted in concrete syntax. Runnables are modeled as rectangles labeled with a name that are contained in the corresponding parent component. In the example, the runnables `lane` and `brake` are located in the software component `Driving` and thus, define the available functionality of the driving component. Furthermore, the runnable variable `wheelControl`, which is denoted with an additional ellipsis and corresponding stereotype, is contained in the actuator component `WheelController`. A runnable set variable can be distinguished from a normal runnable variable by a double border. For ports, the ball socket notation from the UML is used, where a socket denotes the need and a ball the provision of runtime model data. The available runnables in the components can be assigned to tasks, which is exemplarily depicted on the right in Figure 5.29. The task $Task_1$ with a priority of two and a period of ten seconds has a reference to all three runnables and will execute them as ordered sequence `lane` runnable, `brake` runnable, and finally the `wheelControl` variable. As for feedback loop templates, variables must be resolved during the deployment of the corresponding application module template.

Runtime Models in Application Component Templates

Because runnables are considered as independent, atomic execution units realizing the adaptation behavior in ACD, they can read/write runtime models from/to ports with specified model operations. Figure 5.30 depicts the Deurema metamodel and concrete syntax with respect to the handling of runtime models in an ACD. At first, components contain ports that enable the use of runtime models at the architectural level. Second, ports directly reference a runtime model view, which can be seen as interface for this port. Deurema uses runtime models directly to describe the outgoing and incoming knowledge of a component instead of specifying an extra interface description. As for feedback loops, the runtime model view is contained in the application module template and defines the local available knowledge.

Therefore, components can connect each other over ports that reference the same runtime model view, which further enables a sharing of runtime model information between these components. In concrete syntax, the runtime model is directly assigned to the port with a dashed line as shown on the right in Figure 5.30. Because components and corresponding ports define the availability of a runtime model, the concrete access to the information is modeled via model operations performed by a runnable. Thereby, Deurema uses exactly the same way of specifying the retrieved amount of data via model queries as explained for activities in feedback loops.

Additionally, the model operation type, e. g., read or write, performed by the runnable on a given port of the component, defines the type of the corresponding port. A modifying model operation, e. g., write or annotate, is related to a provision of information, which leads to a provided port modeled as ball in concrete syntax. In contrast, the reading of data without manipulation is modeled as socket (requested) port. As for feedback loop module templates, the model operation itself is represented as dashed arrow, which is labeled with the type and following the data flow from the runnables to the corresponding port or vice versa.

In the example, the `lane` runnable reads the context model `RoadSituation`, whereas the `brake` runnable annotates information to the `BehaviorMode` system runtime model. There is no assumption about the execution order of the read and modifying model operation, unless the corresponding runnables are assigned to a task. Furthermore, once a runtime model is read by a runnable, the knowledge is locally available in the context of the component. Thus, a

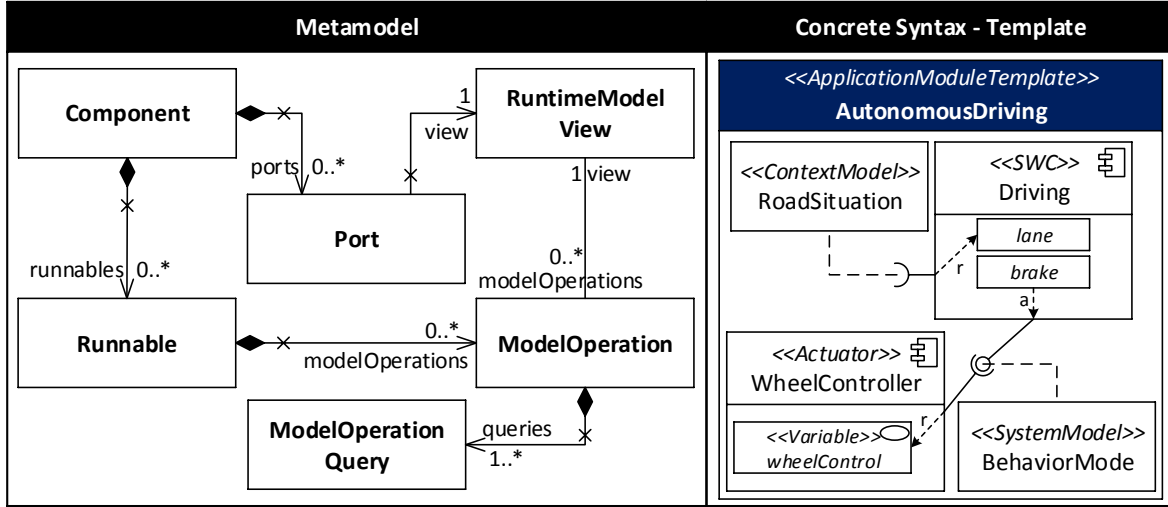


Figure 5.30: Deurema ACD runtime models

component aggregates a set of functionalities in form of runnables, defines the ports for the possible exchange of information, and maintains the availability of read information for all runnables within the boundaries of the component.

Conceptually, there are three combinations that arise using different model operations in two different components, which share one runtime model over one port. Figure 5.31 shows on the left all combinations together with the overall data flow between the components. For all three combinations, two software components C_1 with the runnables R_1 , R_2 and C_2 with R_3 , R_4 are shown. For simplicity, the runtime model for each port is only shown for the third combination, which is an architectural system model, but is assumed for all combinations. For case (1), the decisive model operations for the component C_1 is the writing of data by runnable R_2 , which turns the port into a provided one. The runnable R_3 in the second component performs a read operation on the runtime model, which indicates a requested port. Therefore, there is a data flow from component C_1 to C_2 in the first example.

The difference to the (2) case is the missing writing operation of R_2 , which is now a reading of the runtime model. Thus the beforehand provided port turns into a data requesting port. This example shows the difference to traditional component port interfaces, where always one component offers data, which can be consumed by other components. In Deurema, there is no dedicated interface description and the runtime model is directly connected to the port. Because the runtime model views are maintained by the parent template, read-only access on available local knowledge base can be described in Deurema. Therefore, the data flow for the second case is from the runtime model to the components C_1 and C_2 .

In contrast, the third combination (3) describes a situation, where both components provide data indicated by the write model operation of R_2 and R_3 . Consequently, the overall data flow is from both components to the linked runtime model.

In summary, the Deurema application component port concept is not limited to pairs of data provider and requester. The model operations performed by the runnables define the port type. Furthermore, the port interface is equal to the linked runtime model. The amount of data, which is read and written, is defined by the model queries that belong to each model operation as shown in the metamodel in Figure 5.30.

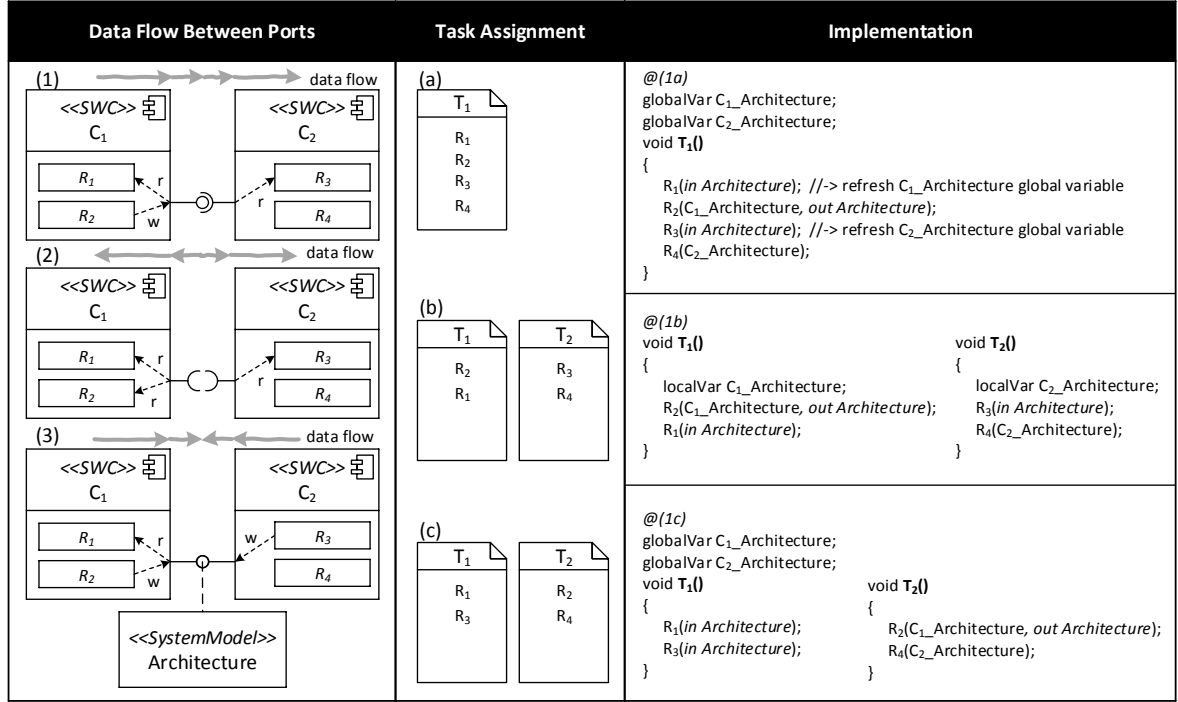


Figure 5.31: Deurema ACD port and task combinations

Due to the locality principle of read runtime model data within one component, different possibilities of realizing the access to the runtime models must be considered by assigning runnables to tasks. Figure 5.31 depicts in the middle column three different task assignments for the runnables in C_1 and C_2 on the left. Variant (a) defines a single task T_1 , where all runnables from R_1 to R_4 are subsequently assigned. The second alternative (b) defines two different tasks, where each task contains only runnables from the same component. Finally, the last configuration (c) defines two tasks, where the runnables from the two components are mixed.

With respect to the execution semantic of different runnable assignments, the pseudocode in the right column in Figure 5.31 shows a possible implementation for each combination of the port combination (1) with the three variants (a), (b), and (c). The key aspect of the pseudocode is that the different task assignment and the order of the runnables lead to completely different behavior concerning the executed functionality as well as visibility of runtime model data. On the one hand, Deurema restricts the available knowledge to the explicit modeled access via ports by the specified model operations. On the other hand, knowledge is visible for all runnables in the boundaries of the same component, if it is retrieved once. Thus, the pseudocode in the figure uses global and local variables as well as *in* and *out* keywords in parameters to visualize the visibility of data and their access. Therefore, variables are internally used by Deurema to capture the last retrieved snapshot of the accessed runtime model. The *in* and *out* keywords represent a direct access to the runtime model over a port. Furthermore, in the pseudocode, tasks are implemented as methods, runnables are depicted as functions, the parameter defines the available knowledge, and the execution order of the function within the parent method is equal to the order of runnables in the task assignment.

Consequently, the realization of variant (1a) in the upper right in Figure 5.31 shows one method $T_1()$. This method contains four functions according to the runnable assignment of the task. As shown on the left, the runnable R_1 reads the architecture model, which is mapped to an incoming parameter of the corresponding function. Afterwards, the architecture runtime model information is available for all runnables in the component C_1 . Thus, the global variable $C_1_Architecture$, representing the content of all read data from the Architecture runtime model performed by runnables within component C_1 , is updated after the runnable R_1 performs a read model operation. Furthermore, R_2 is executed after R_1 as defined by the task mapping. The runnable R_2 gets the updated architectural model information as parameter, which can be further used by the internal runnable behavior. The realization for the runnables R_3 and R_4 is straight forward with the same line of argument as above. For this variant, the pseudocode uses global variables to cache the retrieved knowledge from the architectural model. This is necessary, because runnables from different components are mapped to one and the same task. Furthermore, each variable restricts the access to the available knowledge according to the modeled containment of runnables in components.

The variant (1b) uses two tasks for the runnable assignment, whereas each task executes only runnables from one component. In this special case, the access to knowledge can be differently realized by using local variables in the corresponding methods. Therefore, there are two methods for the two tasks on the right in Figure 5.31. The variables can be restricted to the context of the method, because all runnables in the method (task) are from the same component. The realization of the function parameter for each runnable follows the same principle as explained above. One special characteristic of this realization has to be noted. The runnable R_2 accesses the local variable before the runnable R_1 updates the information afterwards. The order is defined in the task mapping and therefore a correct realization. As a consequence, R_2 always reads the content of the variable from the last execution of the task. Therefore, the data is at least as old as one period of the task and the local variable, which contains the runtime model information, is updated by executing R_1 afterwards.

The last variant (1c) groups the runnables by their access to different tasks ignoring their belonging to components. Therefore, the access is realized by global variables as already explained for variant (1a). The difference is that the execution order of the two tasks dictates the age of the data. The execution order depends on task properties as for example the period and priority as well as on the used scheduling algorithm.

In summary, the modeling of different components with runnables and their assignment to tasks dictates the realization of possible execution of the runnables. The Deurema interpreter ensures the correct access to runtime information as exemplarily visualized by the pseudocode in Figure 5.31. Furthermore, the causal order of updating and reading data is defined by the order of the assigned runnables respectively by the execution order of tasks. Additionally, because runnables can be executed in different tasks, race conditions and data inconsistencies are also handled by the Deurema execution environment. Tasks with the same priority can be differently scheduled during a simulation, which leads to different adaptation effects of the modeled behavior. Therefore, Deurema supports different simulation variants concerning the scheduling and timing behavior of tasks as well as runnables. Because runnables are considered as behavior models, they can be handled in the same way as feedback loop activities during a simulation.

Application Component Template Example

After explaining the concepts of Deurema ACD, Figure 5.32 shows an example of an application module template realizing an autonomous driving functionality within a smart car as motivated

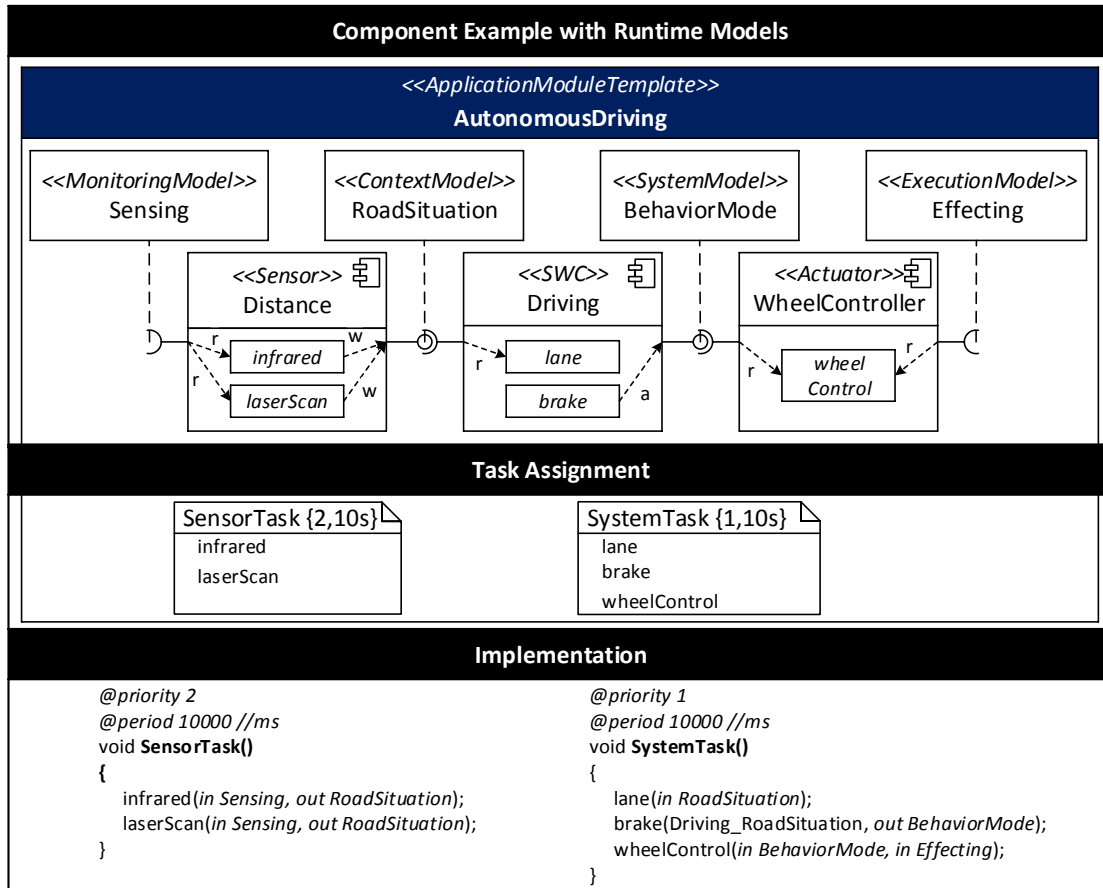


Figure 5.32: ACD example

at the beginning of this section. First, there are three components depicted. A sensor component named *Distance* contains two runnables that realize a distance measurement using an infrared and laser scanner respectively. Therefore, both runnables read the monitoring model *Sensing* that contains information about how to access the physical sensors of the smart car. The measured information about the detected distances and eventually found obstacles such as other cars is written to the *RoadSituation* context model over the corresponding port. The software component *Driving* contains two runnables, where the *lane* runnable reads the current road situation and the *brake* runnables considers if an emergency brake is necessary. The desired behavior is annotated to a system runtime model named *BehaviorMode*. The actuator component, which has control over the wheels of the smart car, reads the advised behavior together with an execution model that has information about how to access the physical brakes of the car.

Modeling the architecture of an application module is the first step. Second, all modeled runnables in the architecture are assigned to tasks defining the execution order. Thereby, one sensor task contains the two runnables from the *Distance* component and has a priority of two with a period of ten seconds. The other task contains the remaining runnables, whereas the software component runnables are executed before the actuator runnable. Due to the higher priority, the *SensorTask* is always executed before the *SystemTask*. Finally, one possible

realization of the task mapping is shown in form of pseudocode at the bottom of Figure 5.32 that follows the same principles as discussed before.

5.3.5. Behavior Module Template

Behavior module templates are designed to describe the adaptation system behavior in form of graph transformation rules by modeling trigger-action conditions. As motivated at the beginning of this chapter, this allows an integration of various model types such as automata, architectural models, or behavioral diagrams from different domains by considering all of these model types as graph structures. Furthermore, Deurema already considers the knowledge representation in form of runtime models as graphs. The abstract graph representation enables the application of the powerful concept of graph transformation rules, which is further suitable for analysis and verification. Additionally, graph transformation rules have a nonambiguous semantic that defines the manipulation of the underlying model. The use of single graph transformation rules for specifying the white box behavior of an activity in feedback loops or a model operation query is already discussed in Section 5.3.2. However, behavior module templates introduce the use of graph transformation rules as first class concept, whereas the rules interact with the available runtime models to perform the adaptation of the system.

Figure 5.33 sketches the use of two trigger-action rules, which investigate an architecture system model for traffic jam detection (R_1) and the start of an intelligent traffic system (R_2). In general, rules within a behavior module template supervise the available local knowledge according to their trigger condition. If the situation (trigger) is found in the runtime models, the specified action of the rule is performed. Rules are considered as independent. The execution of rule actions over time defines the behavior of the corresponding module template.

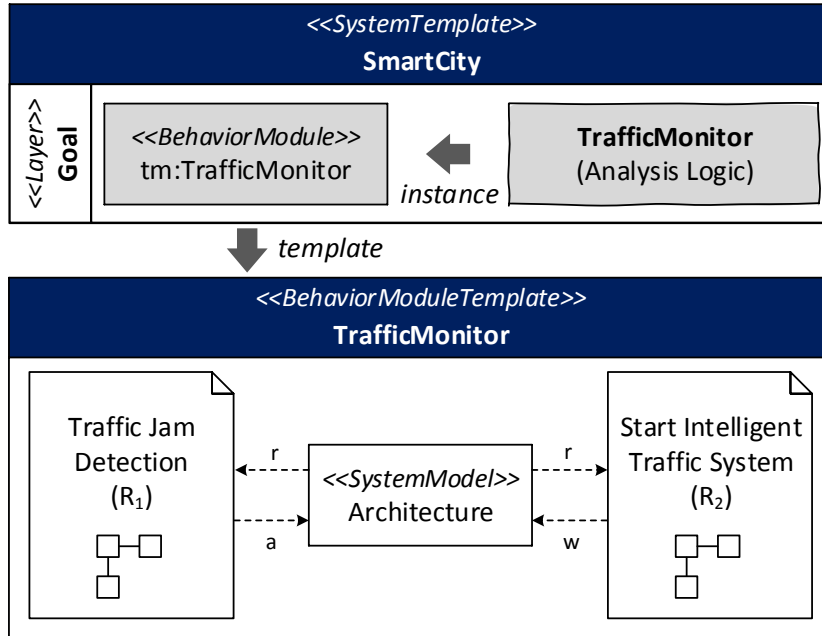


Figure 5.33: Smart city running example: Deurema behavior module template

As for feedback loop and application module templates, behavior module templates must be instantiated and placed on a layer in the system template. Due to the declarative character of trigger-action rules, they are usually used to supervise system requirements as goals or

constraints. Furthermore, rules can be used for analyzing the underlying knowledge base. In this case, the action of a rule can be used to inform the developer (at development time) or the system (at runtime) about a possible violation of requirements or found system situation.

The Deurema metamodel defines that behavior module templates contain an arbitrary number of graph transformation rules as depicted on the left in Figure 5.34. Thereby, the concrete graph transformation rule is defined by the rule type, which can have an arbitrary number of rule instances. As for activities, the type-instance separation introduces a clear type level that defines the available functionality (graph transformation rules) once, which can be reused in and across behavior module templates. Furthermore, Deurema supports the modeling of graph patterns over arbitrary domain models, whereas the concepts of the domain models are defined by a metamodel.

Rule types are considered as behavior models and thus, can be executed by the Deurema interpreter. Consequently, each rule execution follows the black-gray-white box paradigm as explained for activities in Section 5.3.2. Thus, the LHS of the graph transformation rule type defines the trigger condition and the RHS corresponds to the action, whereas both attributes are defined in the BehaviorModel class in the Deurema metamodel. A behavior model can be determined as white box, if the trigger condition and action are modeled in Deurema. Transferred to a graph transformation rule, the LHS and RHS are explicitly specified. A difference to feedback loops is that the causal order is not explicitly defined between rules and thus, rules are considered as independent to other rules within the template definition. Instead of a control flow concept, rule instances refer to a set of additional properties, which further restrict the overall applicability of a rule. As shown in the metamodel in Figure 5.34, rule properties are a *priority*, a *local* and *global application* count, a *period*, and a *probability* together with the number of available *trials* of applying the rule. Therefore, the behavior in form of a graph transformation rule is defined in the rule type, whereas the rule instance refines that behavior by additional properties. The assignment of properties may differ between two rule instances that refer to the same type, which allows on the one hand, a reuse of behavior and on the other hand, a fine-grain adjustment of the adaptation logic to the current needs in the behavior module template.

Beside rules, a behavior module template can contain variables as shown at the bottom in the metamodel in Figure 5.34. A rule variable is a placeholder to exchange a single rule denoted by the corresponding RuleVariable class in the metamodel. Furthermore, a RuleSetVariable allows its replacement by multiple rules during system reconfiguration. As for feedback loop module templates, variables must be assigned to concrete values during the instantiation of the behavior module template.

Concrete Syntax

Behavior module templates are modeled in Behavior Rule Diagrams (BRD), where the concrete syntax for a TrafficMonitor example is shown at the right in Figure 5.34. The BRD contains the behavior module template that has a name and the stereotype «BehaviorModuleTemplate». In the example, the template specifies a TrafficMonitor behavior module. Furthermore, rules are modeled as rectangular notes that emphasized the declarative character of the graph transformation pattern. Rule properties, which are discussed below, are specified in curly brackets below the rule. Finally, variables contain an additional ellipse symbol in the upper left corner of the note denoting the placeholder characteristic, where the rule set variable has an additional double border.

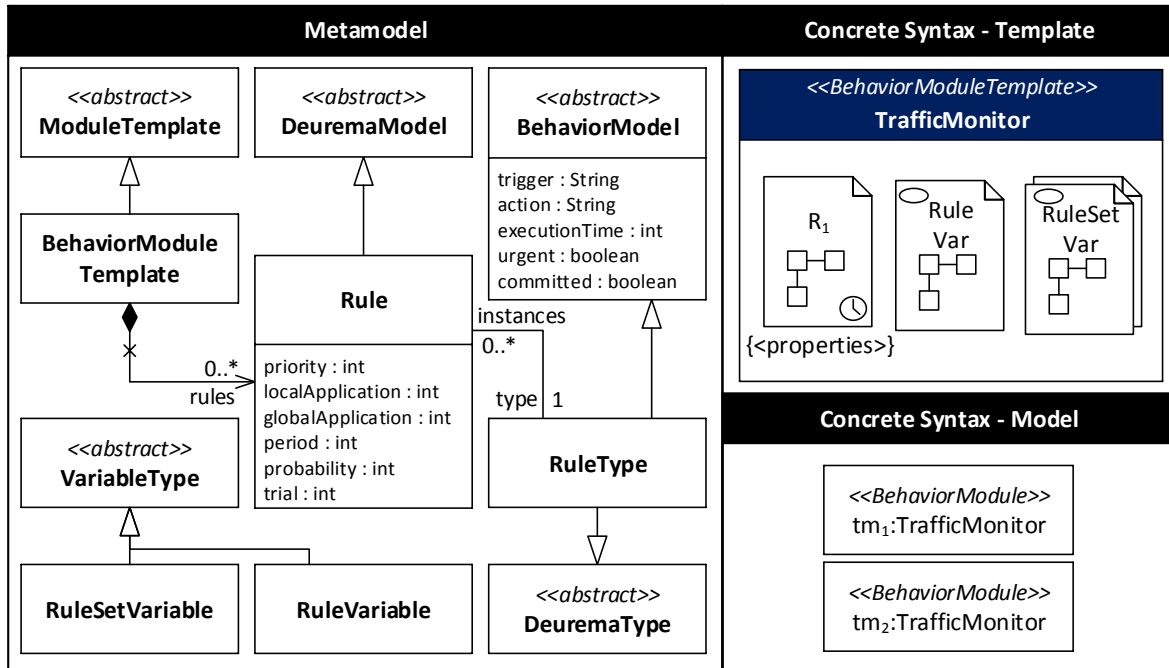


Figure 5.34: Deurema Behavior Rule Diagram (BRD)

Module instances of behavior module templates are modeled as rectangles with the stereotype «BehaviorModule», where the naming scheme follows the object diagram notation. In the example in Figure 5.34, there are two behavior modules tm_1 and tm_2 that refer to the same TrafficMonitor template.

Behavior Rule Properties

Graph transformation rules are a powerful concept of BRD that consist of a LHS and a RHS. Because the LHS of the graph transformation pattern specifies the overall applicability of the rules, the Deurema rule properties further restrict the application of a rule after a match is found. All rule properties are defined in the Rule and BehaviorModel class in the metamodel of Figure 5.34.

First, rules have a priority, whereas a higher prioritized rule is always executed before a rule with a lower priority. Priorities allow a grouping of rules. The execution order is nondeterministic between rules with the same priority. Furthermore, Deurema assumes no specific scheduling policy (e.g., round robin), which guarantees a successive execution of rules. Consequently, the same rule can be executed over and over again, although there are other rules with the same priority, as long as a match for the LHS is found. However, the scheduling of rules belongs to their execution. The Deurema interpreter enables the support of different strategies, which includes a scheduling policy that fits to the domain problem.

Second, the application count of a rule can be locally and globally restricted by the `localApplication` respectively `globalApplication` property. The maximum local application count defines how often each rule can be executed within one execution of the parent behavior module. Therefore, the local application count is reset if the corresponding behavior module becomes inactive. In contrast, the global application count of a rule defines how often the rule can be executed throughout the whole lifetime of the execution of the adaptive SoS. Consequently, the global count is not reset during execution. Local application counts can be

used to restrict the adaptation effects of rules to a desired amount of times that enables the execution of other enabled rules within the module. Whereas, global application counts may be useful for modeling one-shot or n-shot rules restricting adaptation effects over the whole system. Thus, the application count of a rule restricts the occurrence of adaptation effects, which can be fine-grained modeled for each rule.

Third, according to the need of modeling timing properties, each rule has a period that specifies the minimal amount of time a rule has to wait until its next possible execution. Furthermore, rules can have an execution time property that defines the amount of time a rule needs to perform its action as defined by the RHS. Additionally, inspired by timed automata [23, 34], Deurema supports **urgent** and **committed** rules. Urgent rules are higher prioritized as non-urgent rules. With respect to timing, during the execution of urgent rules, no time elapses in the system. Therefore, urgent rules have a zero execution time. The definition for committed rules is even stricter as for urgent rules. During the enabling and execution of a committed rule, no delay in the complete system is allowed. Thus, an execution of a committed rule elapses no time in the system and between two enabled committed rules, no other non-committed or urgent rule can be executed. Therefore, an interleaving with other actions can only be happen in the system if those actions have the committed property, too. Timing properties are important for the simulation of the adaptive SoS behavior, whereas Deurema supports different strategies. The urgent and committed property can be useful in early development stages neglecting realistic execution time, whereas a simulation gives insights into the overall interaction between modules showing that designed adaptation logic performs as expected. Of course, a zero execution time semantic is an abstraction from the real world, which implies that real execution times should be considered later in the development. As an effect, even if early simulations show that the overall adaptation logic corresponds to the goals, timing effects may raise further interaction problems or timing deadlocks that have to be resolved to guarantee proper execution.

The fourth and last property for rules is related to the probabilistic occurrence of system effects. Therefore, rules can have different application probabilities that are modeled in the **probability** and **trial** attributes of the **Rule** class. The former defines the likelihood between zero and one hundred, whereas the latter indicates the total number of trials before the rule is disabled. If the LHS for the rule is found, Deurema rolls a dice, which corresponds to the modeled likelihood. If the dice denotes an activation, the rule is executed. Otherwise, the trial count is increased and Deurema rolls the dice again, until the maximum number of trials is reached. The trial count is reset after the parent module becomes inactive and enabled again. If an additional period is specified, the time between two consecutive trials is at least the defined period. Probabilities can be used to model rare adaptation effects or to introduce errors in the adaptive system. Especially for the latter case, application count limits can be used to limit the overall number of introduced errors in the system.

Figure 5.35 shows examples for different property combinations³ of behavior rules. The rules R_1 and R_2 show the same property configuration that is a priority of one, an unlimited number of local and global applications (* multiplicity), a period of ten time units, a probability of one hundred percent with an unlimited number of trials, and finally an execution time of one time unit. All of these properties, except of the period, are the default configuration and can be omitted. The period can be directly modeled under the clock symbol as done for the rule R_2 . In the rule R_3 , the priority is increased to five and the maximal global applications are

³The timing properties **executionTime**, **urgent**, and **committed** are inherited from the **BehaviorModel** class. All other properties are defined in the **Rule** class in the metamodel in Figure 5.34.

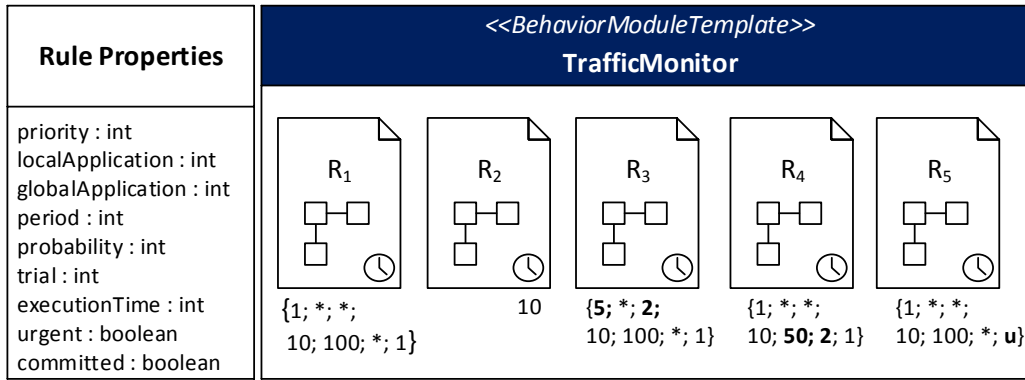


Figure 5.35: Behavior rule property combinations

restricted to two. Therefore, R_3 will be executed before R_1 and R_2 (if enabled by the LHS), but at least two times during the lifetime of the system. The rule R_4 shows the modeling of a probabilistic rule, whereas in this example, the execution probability is reduced to fifty percent by a maximum of two trials. The execution environment has to wait ten time units between both trials, which is defined by the period property. If the rule R_4 is the only rule in the template and because of the given probability and trails, a simulation should statistically execute R_4 once for each execution of the parent *TrafficMonitor* module. The rule R_5 is an example for an urgent rule, whereas a committed rule is modeled with a *c* instead of the *u*. Because of the definition of timing properties, there can be only one timing attribute modeled. Therefore, modeling an execution time, urgent, or committed property excludes the others from the specification.

In summary, the general application of a rule is defined by the LHS (the trigger), which is specified in the corresponding rule type. Additionally, the execution semantic depends on the modeled rule properties such as priority, application count, occurrence, or period. The adaptation effect (the rule action) is modeled by the RHS of the corresponding graph transformation rule. Furthermore, modeled timing properties are important for the simulation of the overall adaptive SoS behavior and must be adjusted according to the preferred simulation strategy (e. g., zero execution time or realistic timing).

Runtime Models in Behavior Module Templates

Due to the behavior module template is one of the four supported template types in Deurema, it follows the same concepts of integrating runtime models as explained for feedback loops. Behavior rules can access and manipulate runtime models, where the corresponding excerpt of the Deurema metamodel is depicted on the left in Figure 5.36. Each rule defines model operations that link to a corresponding runtime model view, which is contained in the same template definition as the rule itself. Thus, the runtime model views specify the local available amount of information of the behavior module template. Furthermore, model operations are specified by a type and knowledge queries. As in feedback loop module templates, each model query follows the black-gray-white box concept. Queries are applied on the runtime model to retrieve the desired information, which is used by the behavior rule afterwards. Therefore, the modeled trigger condition of the rule is performed on the retrieved data subset, which is defined by the model operation queries.

On the right in Figure 5.36, the concrete syntax for runtime model access via model operations is depicted. Runtime models are specified as rectangles and rules as notes as

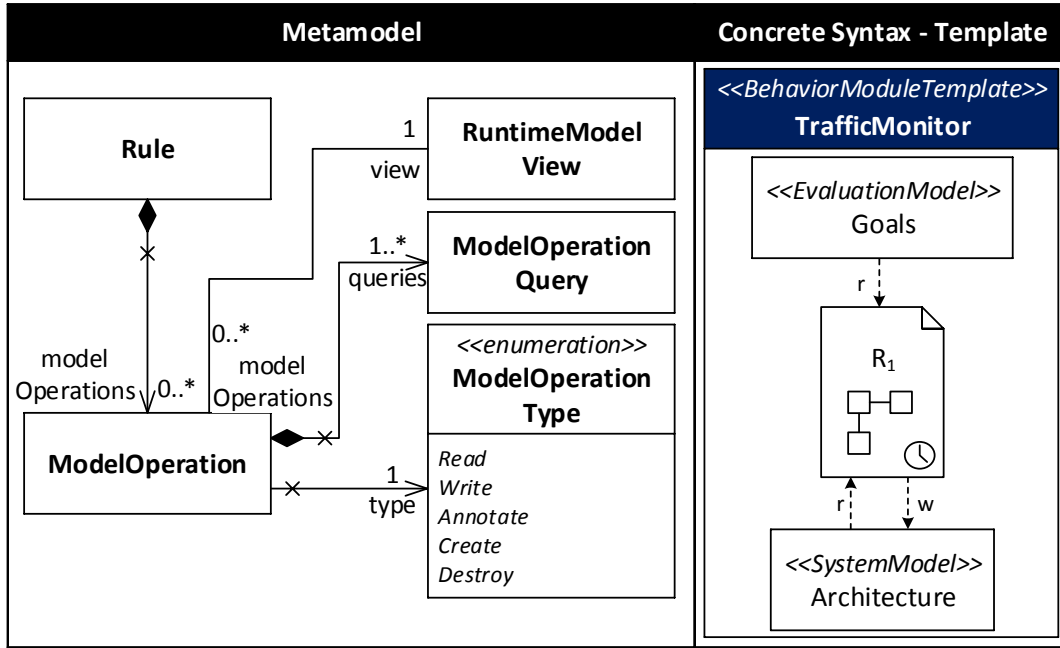


Figure 5.36: BRD rules and runtime models

already discussed before. Furthermore, model operations are modeled as dashed arrows labeled with the type. The direction of the arrow follows the data flow from source to target. For example, the read operation on top of the rule R_1 is performed from the runtime model named Goals (source) to the rule R_1 (target).

As special characteristic, from all white box behavior models and in particular for behavior rules, the model operation type can be automatically derived by Deurema. Therefore, the characteristics of the modeled graph transformation imply the corresponding operation type as follows. The combined notation for a graph transformation rule introduced in the preliminaries in Section 2.2 is used.

All five supported runtime model operation types, which can be used by a behavior rule, are enumerated in Figure 5.37. (1) The characteristic for a *read* model operation is that no side effects are recognized on the corresponding runtime model. This characteristic transferred to a graph transformation rule implies that the LHS and RHS are equal. Thus, the retrieved knowledge is used for searching the given pattern and finding a match for the LHS, whereas the RHS defines no modifications on the match. Because reading a runtime model corresponds to the existence of the LHS of the rule, it is included to all other runtime model operations except for create. Note, model operations are further defined by model queries, which specify the amount of data that is retrieved from and written to a runtime model view as discussed before (cf. Section 5.3.2). Model queries can be defined as declarative graph patterns without any side effects in Deurema. By applying the model queries, the retrieved models are aggregated and afterwards, the behavior rule is applied on the retrieved knowledge base.

(2) The most general model operation is a *write* modification. Writing a runtime model includes a read, an optional create and a delete of runtime model elements. The essential characteristic for deriving the model operation from the graph transformation rule is a missing element in the RHS of the pattern, which is equivalent to a deletion of an element. Beside a deleted element, the modification of an attribute is also considered as writing operation,

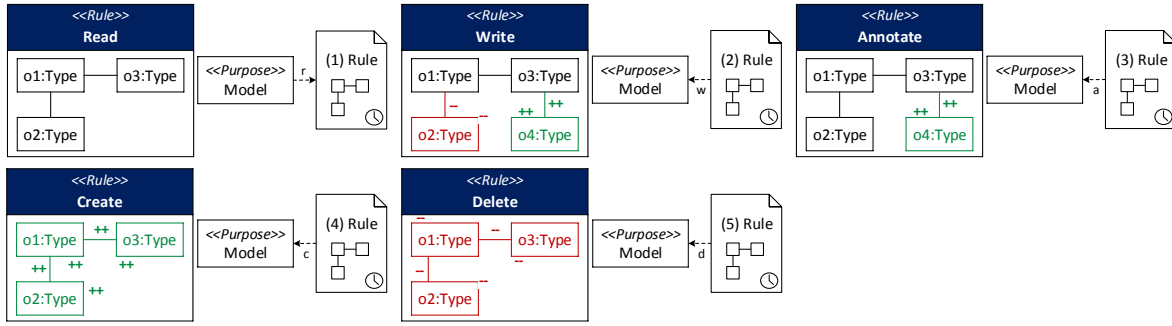


Figure 5.37: Derived runtime model operation types

because the old value of the attribute is replaced by a new one. Additional created elements in the RHS are optional for this model operation.

(3) *Annotating* a runtime model denotes the appending of information without modifying the existing knowledge in the system. Therefore, the RHS of the graph transformation rule is only allowed to have new elements. The removal of existing elements in the pattern is forbidden, because this would refer to a *write* model operation.

(4) The *create* model operation type is a special case for annotating it, where the LHS of the corresponding graph pattern is empty. Thus, all elements are created by applying the RHS of the rule.

(5) The *delete* model operation type is the contrary model operation to a create. Therefore, the RHS of the graph transformation rule must be empty, whereas the LHS denotes the amount of deleted information. Note, the delete model operation type denotes that all model elements, which are retrieved by the corresponding model queries, are deleted. This does not mean that the complete runtime model view is deleted, because this depends on the specific read queries that are applied before to retrieve the desired information.

In summary, the module operation type can be automatically derived from the corresponding rule pattern. Additionally, violations of the modeled pattern in the graph transformation rule and the specified model operation type in Deurema can be recognized and reported to the developer. The search for a match is done on the data that is retrieved by specified model queries. The existing of a LHS in the graph transformation rule corresponds to a read operation, whereas the difference in the RHS determines the type of the corresponding modifying model operation. Because for the lack of information in black box and gray box behavior models (rules), the model operation type cannot be derived during development. Deurema may monitor the modeled adaptation behavior, its access characteristics as well as the adaptation effect during simulation and suggest an appropriate runtime model type or report violations. However, the monitored characteristics and the corresponding suggestion by Deurema may differ from the real implementation for black box and gray box behavior.

Behavior Module Template Example

For closing the behavior module template concepts, Figure 5.38 refines the running example from the beginning by depicting two behavior rules R_1 and R_2 in the TrafficMonitor behavior module template. At the top, the figure shows the specification of the two rules inside the behavior module template. On the bottom, the figure visualizes the derived model operation types for the two rules. For this example, the expected execution order of the adaptation effects is first R_1 directly followed by R_2 .

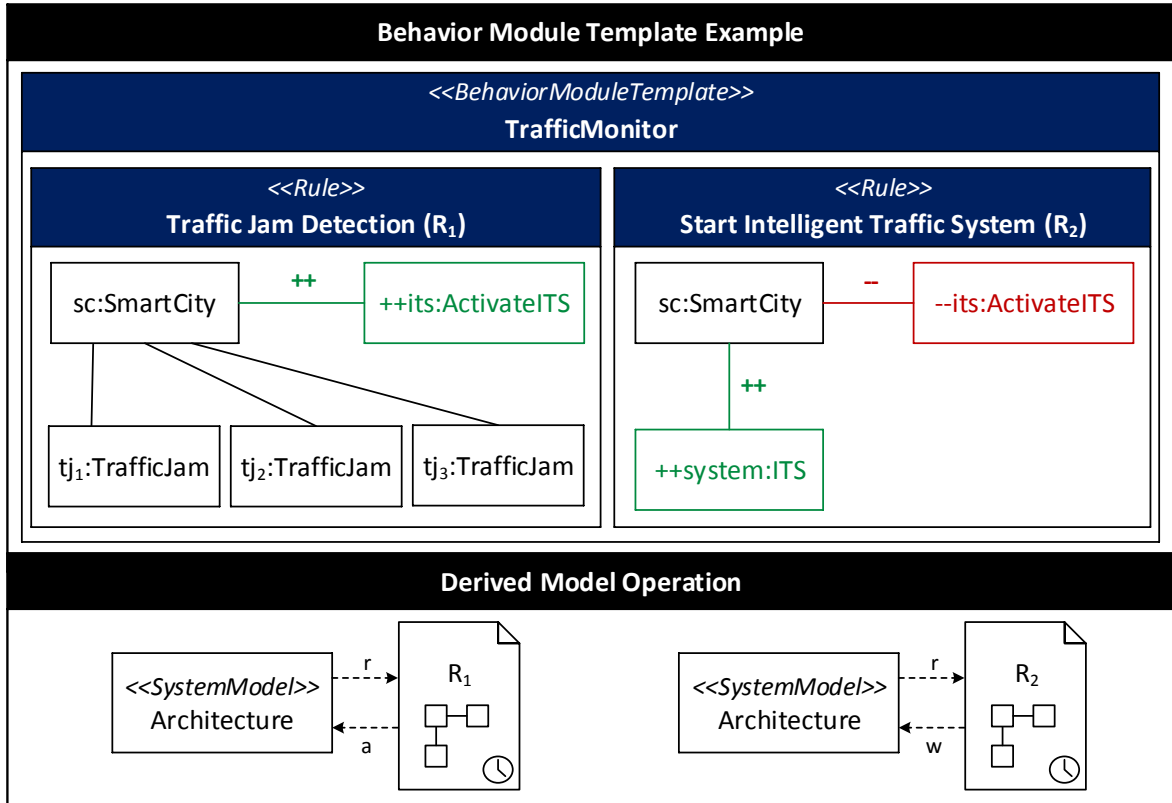


Figure 5.38: BRD rule example

The rule R_1 has the typical structure of an analysis rule, which searches for specific system situations and annotates it for further treatment. Therefore, the rule searches for a situation in the smart city, where at least three traffic jams occur. Thus, the LHS of R_1 defines a pattern of four elements, where a SmartCity object has a reference to three TrafficJam objects. Furthermore, R_1 denotes the found situation with the need for an adaptation, which is encoded in the its:ActivateITS object. In the example, the adaptation need is the activation of an intelligent traffic system for handling the traffic jams, as for example described in [71]. The types of the references in the pattern can be nonambiguous determined and thus, are omitted in this example. Of course, the type of the reference between objects must be specified in a graph transformation rule pattern modeled in Deurema as introduced in Section 2.2.5.

However, in the case of a found traffic situation described by R_1 , the rule R_2 becomes enabled. The adaptation step in this example is the removal of the adaptation need annotation (the its:ActivateITS object) and the creation of a corresponding intelligent traffic system (system:ITS object). The effect on the physical system after applying both rules is the starting of a software system that handles the traffic jam by an intelligent routing of the underlying traffic in the smart city, e.g., by changing the maximum speed limits on highways to adjust the overall traffic flow.

This example shows how the application of graph transformation rules and the corresponding manipulation of the runtime models realize the adaptation of a Deurema behavior module. In general, the declarative nature of graph patterns facilitates the specification of special situations as trigger condition in the adaptive SoS together with an appropriate reaction. That

corresponds, but is not limited, to the specification of watchdogs and monitoring modules that observe the adaptive SoS as well as report unwanted or special situations, which further support the verification of assumptions about the overall modeled SoS behavior.

5.4. Deurema Adaptive System Architecture

After discussing the Deurema core concepts, the use of runtime models and the different module templates, Deurema specifies the layered architecture of the system in Layer Diagrams (LD). The LD contains module and system instances, whereas the internals of a module or system are defined by the corresponding module template respectively system template. Beside the deployed instances, a LD comprises trigger dependencies between modules that can be used for inter-loop coordination of the adaptive behavior.

Figure 5.39 shows a sketch of the smart city system template, which contains different module and system instances. As discussed in the former sections, there is a module template description for each module instance following a specific modeling approach such as feedback loop or component-based modeling. This section refines the Deurema concepts related to the inter-loop coordination between modules by means of module trigger dependencies as highlighted in gray in the figure.

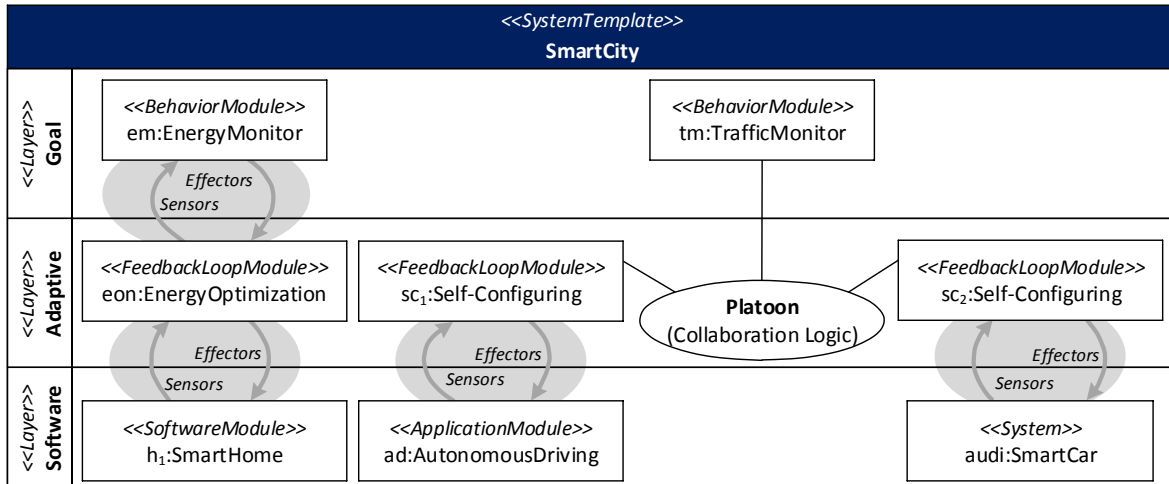


Figure 5.39: Smart city running example: Deurema module trigger dependencies

Due to the adaptive behavior is specified in module templates, modules, which are template instances, are the basic runtime entities in Deurema. In general, modules are considered as independent from other modules, with the exception of two cases. First, a module can explicitly trigger another module that defines a causal inter-loop dependency between both, where the triggering module emits an event to the triggered module. Second, modules can synchronize their behavior or exchange knowledge via collaborations, which are more complex than the use of triggers and can comprise several interactions between modules.

The excerpt of the Deurema metamodel related to the module trigger concept is depicted in Figure 5.40. Deurema distinguishes between two trigger types that are TimedTrigger and EventTrigger. A module can have maximal one timed trigger, but may emit an arbitrary number of different event triggers. Time triggers have a period that defines the minimum waiting time between two consecutively runs of a module, which refers to that time trigger. Event

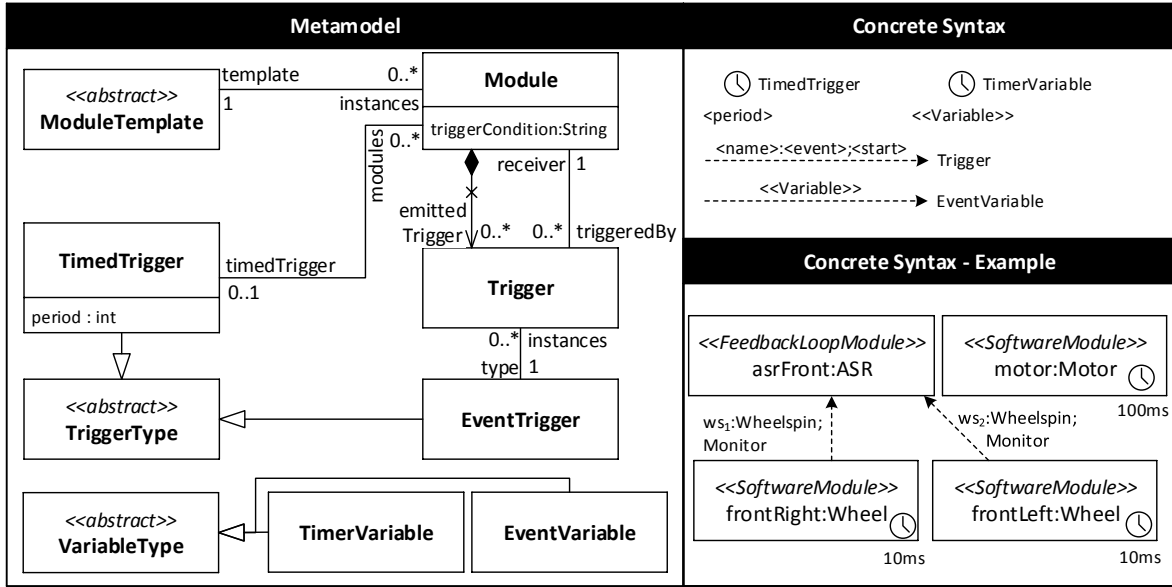


Figure 5.40: Deurema module trigger

triggers are contained by the emitting module, refer to a type that enables the specification of different kinds of event triggers, and have exactly one receiving module.

For execution purpose, each module must have at least one timed or event trigger in Deurema. Furthermore, a module is enabled for execution, if for condition (1) the elapsed time between the last run and the current time is greater or equal the period of the referenced time trigger. Condition (2) defines that at least one incoming event trigger was fired at least once. Consequently, there are the following implications of this execution semantic definition. First, if no time trigger is specified, the condition (1) is fulfilled. Second, if no event trigger is specified, the condition (2) is fulfilled. Third, if an event is thrown multiple times between two consecutive runs of a module, all incoming events are collected and processed by the next activation of the module. Fourth, the execution order of two enabled modules is nondeterministic. This is for example possible, if two modules reference the same time trigger. Fifth, time and event trigger can be combined with the effect that condition (1) and (2) must be fulfilled. Sixth and finally, for more than one incoming event trigger, the trigger condition, specified in the Module class in the metamodel in Figure 5.40, is evaluated. This trigger condition is a logic expression over the incoming event trigger that must be evaluated to true, additionally to the condition (2).

Instead of the concrete modeling of time and event trigger, Deurema supports the specification of trigger variables for both types respectively. Thereby, timer variables can be replaced by different timed trigger types that allow the modeling of a range of possible periods for one module. Furthermore, event variables allow the configuration of several event trigger types with the effect that a triggered module can be activated by different events. However, similar to the variable concept in module templates, the modeled configuration space is resolved during the deployment of the system template, where each variable must be assigned to a concrete value from the modeled configuration space. Therefore, timer and event variables do not occur at runtime in the concrete instance situation of the system.

Concrete Syntax

The concrete syntax of triggers is depicted on the right in Figure 5.40. An event trigger is modeled as dashed arrow that follows the naming scheme $\langle name \rangle : \langle event \rangle ; \langle start \rangle$. The $\langle name \rangle$ refers to the name of the event trigger instance, whereas the $\langle event \rangle$ part refers to the event trigger type. The $\langle start \rangle$ parameter is optional and refers to the initial node of a triggered feedback loop module. The parameter can be omitted, if only one initial node exists. Obviously, this parameter is not necessary for triggering software, application and behavior modules, because all three module templates do not have an initial node concept (cf. corresponding module template sections above).

A timed trigger is denoted with a clock symbol, where the period is annotated below. Both, timed and event trigger variables can be recognized by the stereotype «Variable».

An example is depicted at the bottom on the right in Figure 5.40, where the two front wheels of the smart car trigger a traction control feedback loop (ASR). Therefore, the two software modules of type Wheel emit a Wheelspin event to the feedback loop module of type ASR named *asrFront*. Both event triggers reference the Monitor initial node of the feedback loop module as starting point for its local adaptation behavior. Because there are more than one incoming event trigger for the ASR feedback loop module, the beforehand mentioned trigger condition is evaluated every time a Wheelspin event is emitted. Examples for such a trigger condition can be: (ws_1 and ws_2) as well as (ws_1 or ws_2), whereas the latter is more useful in this example. Thus, every time one of the event triggers is emitted by one of the wheels, the traction control feedback loop is triggered and ready for its execution.

Furthermore, because Deurema allows the usage of time trigger without any event trigger, modules can be annotated by a clock symbol that specifies the period of a corresponding time trigger. In the example, both wheels have a timing trigger with a period of ten milliseconds. Additionally, the Motor software module has a timing trigger with a period of one hundred milliseconds. Consequently, the traction control feedback loop is enabled, if one update events from one of the wheels *frontRight* or *frontLeft* occur, which might happen not earlier as every ten milliseconds. The motor software module can be executed independently from all other modules, but maximally with a frequency of every one hundred milliseconds.

Adaptive Layered Architecture Example

An example for an adaptive layered system architecture of a smart city and smart car is shown in Figure 5.41. Accordingly, the example comprises two system templates, where the smart car is modeled on the top and the smart city system template on the bottom of the figure. The example is used to demonstrate the discussed Deurema triggering concept above and to visualize the interplay of different module types and system instances on multiple architectural levels.

The smart car has three layers, where on the lowest layer four software modules are deployed that encapsulate the domain logic for controlling the wheels of the car. Each wheel software module has a period of ten milliseconds, runs independently from the other wheel modules and triggers via a Wheelspin event a corresponding traction control feedback loop (ASR) at a higher layer. There are two traction control feedback loops, whereas each loop is responsible for a wheel pair (front and rear). The adaptation logic of the ASR module is defined by a feedback loop template and thus, the internal specification comprises different adaptation activities that operate on runtime models as discussed in Section 5.3.2. Furthermore, the traction control feedback loop has no timing trigger and an event trigger condition in the form (ws_1 or ws_2). Thus, the feedback loop is enabled, if the first Wheelspin event from one of the wheels occurs. Each of the both traction control feedback loops on layer L-1 are supervised

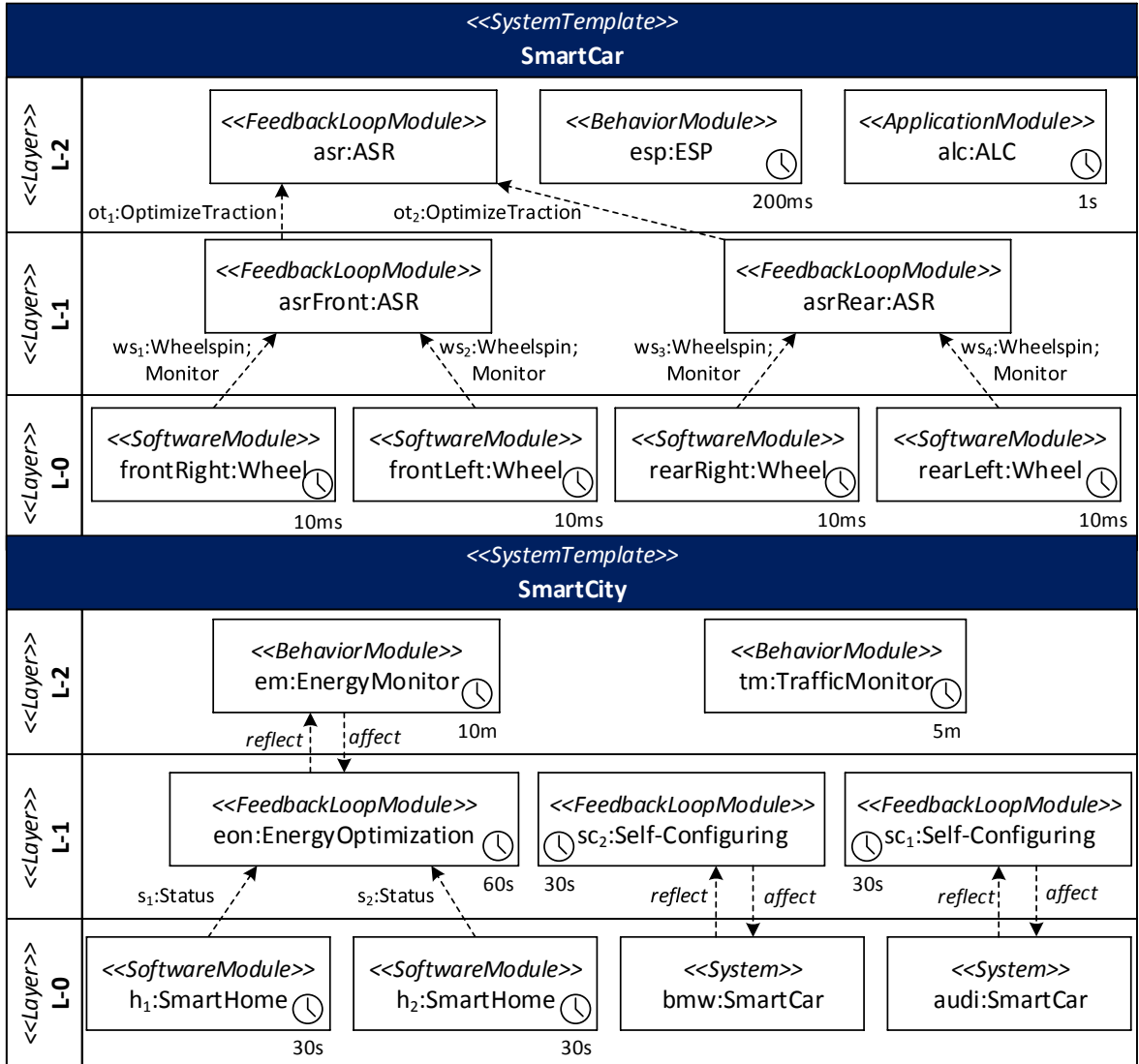


Figure 5.41: Deurema Layer Diagram (LD) example

by another traction control loop on layer L-2. Both, the `asrFront` and `asrRear` feedback loop module emit an `OptimizeTraction` event to the supervising control loop on the highest layer. Therefore, the traction of the smart car is finally optimized by the `asr` module instance in layer L-2, which corresponds to a hierarchical control scheme.

Additionally to the traction control, an ESP and ALC module are deployed at the highest layer of the smart car. The ESP module instance is a behavior module, which consists of graph transformation rules as discussed in Section 5.3.5. It has a period of two hundred milliseconds and runs independently from the traction control feedback loops. In contrast, the adaptive light control module (ALC module instance) follows the component-based development paradigm as discussed in Section 5.3.4. It has a timing trigger with a period of one second and runs independently from the other modules in the smart car architecture. Thus, the ESP and ALC module realize an additional adaptive behavior that does not interfere with

the hierarchical traction control of the smart car. In summary, the three architectural layers together with the deployed modules define the smart car system template.

The smart car system template is deployed twice in the smart city system at the lowest layer on the bottom in Figure 5.41. As shown, the smart city system in the example consists of two smart car system instances named `audi` and `bmw`. The smart cars are placed on the same layer as two additional smart homes, which are specified as software modules. The smart homes trigger an energy optimization feedback loop one layer above via a `Status` event. On the same layer L-1, two Self-Configuring feedback loop module instances supervise the each of the smart car systems the layer below. Thus, the feedback loops watch the current traffic situation and optimize the behavior of the smart cars accordingly. As an emergent effect, the overall traffic flow of the smart city is optimized, which is monitored by a `TrafficMonitor` module on the highest layer L-2. Beside the traffic situation, an energy monitor supervises the overall energy prosumption in the smart city. Both monitors at the highest layer are behavior modules and have a timing trigger with a period of ten respectively five minutes.

As depicted in the example, there is no trigger from the adaptive car systems to the self-configuring feedback loops. In Deurema, systems in an adaptive SoS are considered as independent and therefore are not allowed to trigger other systems or modules. For interactions between systems, the Deurema collaboration or reflection concept can be used. Furthermore, collaborations between modules and reflection of modules are also supported in Deurema. In general, collaborations are a powerful concept for the explicit specification of system as well as module interactions inside the adaptive SoS as introduced in the next section. Afterwards, the Deurema reflection capabilities are discussed.

5.5. Deurema Collaboration

After the discussion of refining system behavior by means of modules and corresponding template descriptions, this section focuses on the Deurema collaboration aspect as emphasized in Figure 5.42. As a consequence from the definition of the terms related to collaborations among systems in the preliminaries in Section 2.4, there are different needs for modeling and analyzing collaborations. At first, collaborations in an adaptive SoS should be explicitly defined. This comprises the *"management of individuals, their activities and resources"* [160]. Thereby, this thesis focuses on systems as individuals, adaptation activities, and runtime models as available resources. Furthermore, because *"correlation wants to happen"* [147], it imposes coordination and emergent behavior [160] in an adaptive SoS. Concerning the goals in Chapter 1, this thesis aims at the systematic modeling of interactions among multiple feedback loops with runtime models in adaptive SoS to achieve a coordinated self-adaptation.

As sketched in Figure 5.42, this section explains the Deurema collaboration concept by means of the platoon running example. In the figure, there are two smart cars, which are supervised by a self-configuring feedback loop each. The feedback loop decides about a behavioral mode switch of the underlying smart car functionality changing between manual and autonomous driving. Furthermore, an autonomous driving is only possible within a platoon. Thus, both smart cars must collaborate with each other to realize a higher functionality. Beside the both cars, there is another participant in the collaboration, who supervises the cars within the platoon with respect to the overall traffic flow optimization of the smart city.

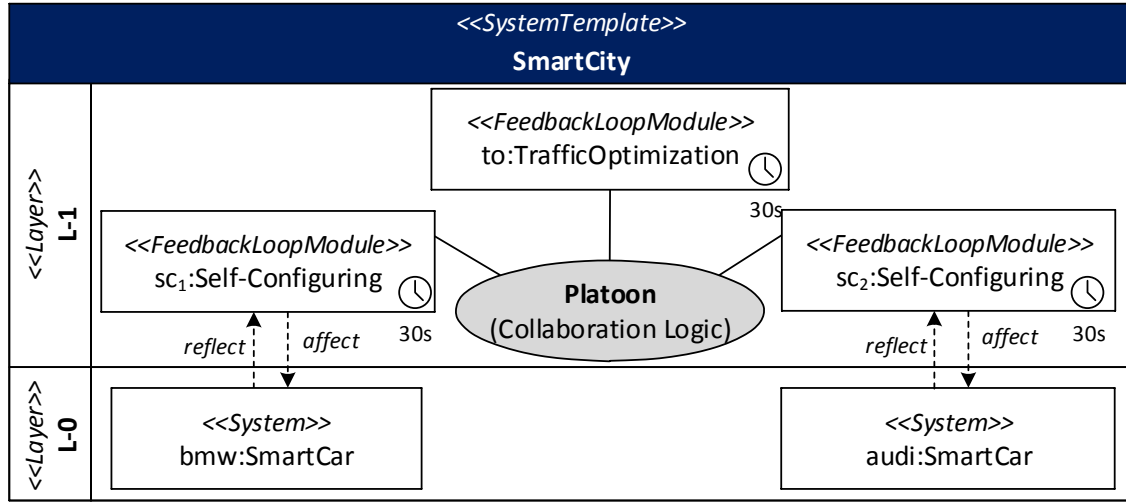


Figure 5.42: Smart city running example: Deurema collaborations

The Deurema modeling language concepts for collaborations are introduced as follows: Figure 5.43 gives an overview of the collaboration modeling dimensions that go along with the structure of this section.

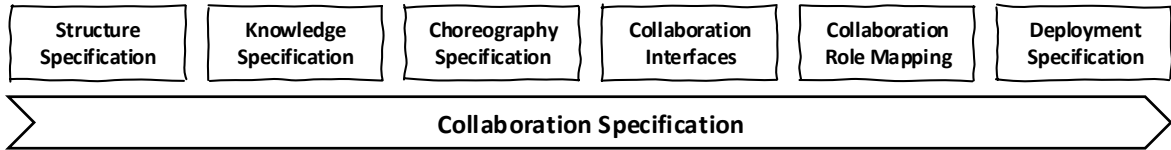


Figure 5.43: Deurema collaboration modeling dimensions

First, the collaboration structure defines, which collaboration capabilities are available in the adaptive SoS. Furthermore, it defines the possible participants of the joint interaction. Second, collaboration resources must be defined that are considered as available knowledge (runtime models) in the SoS. Third, the collaboration choreography solves the need of specifying the interactions, communication mechanisms and causal order of the joint cooperation. Therefore, the choreography considers the beforehand defined participants in the collaboration structure and available knowledge. Fourth, collaboration interfaces define the possible participation and intention of collaborative system behavior targeting the analysis of coherence and congruence of joint interactions. Fifth, the collaboration behavior is integrated into the local adaptive behavior to avoid or to identify contention. Sixth and finally, specified collaborations are deployed in the overall adaptive SoS that can be seen as application of the defined collaboration capabilities to concrete instances within the running SoS.

Each of the six modeling dimensions partially depends on each other. For example, the causal order of the communication between collaborating participants as modeled in the choreography specification can only be defined after the possible participants are identified as done in the structure specification. However, the collaboration modeling concepts must not be applied in a sequential way. The discussion of using the Deurema modeling concepts in different software development processes is not in the scope of this thesis. In the following, each concept is introduced in detail by means of the proposed order in Figure 5.43.

5.5.1. Collaboration Structure

In Deurema, available collaboration types are modeled independently from the local adaptation logic. The collaboration structure specification defines the collaboration types, the role types that participate in the collaboration, and how many instances of each role may participate in a collaboration instance. In the context of an adaptive SoS, a role is an abstract entity that must be realized by respectively integrated into the local adaptation behavior, e.g., defined by a feedback loop template. As a consequence, roles separate the collaboration behavior from the local behavior defined by the module templates.

On the left, Figure 5.44 depicts the Deurema metamodel for the collaboration structure specification. Deurema follows the same type instance paradigm for the collaboration structure as explained for systems and modules in Section 5.1. Therefore, the `CollaborationType` class defines the static structure of the collaboration on type level, whereas the `Collaboration` class represents the possible collaboration instances at runtime. The same line of argument holds for the `RoleType` and `Role` class in the metamodel. Furthermore, role types define a lower and upper bound (multiplicity) for the maximal occurrence of a role instance for each collaboration instance.

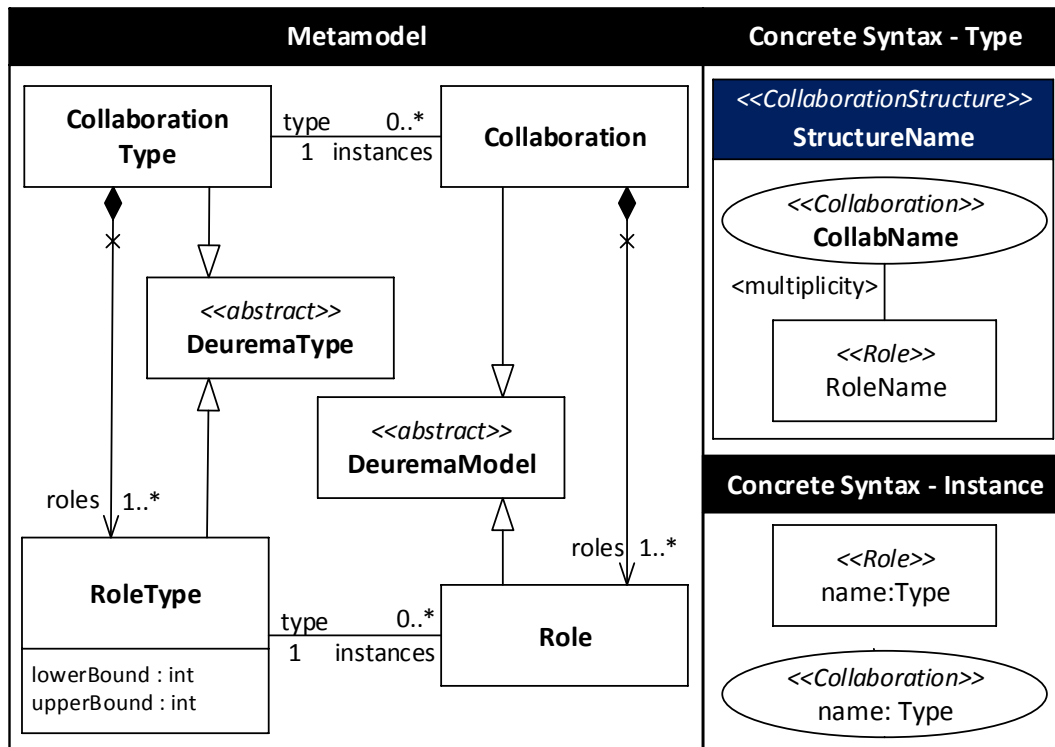


Figure 5.44: Deurema collaboration structure

Concrete Syntax

The concrete syntax for collaborations and their roles is depicted on the right in Figure 5.44. On type level, collaborations are modeled as named ellipses with the stereotype «Collaboration». Role types are modeled as rectangles and labeled with the «Role» stereotype accordingly. The multiplicity of a role is directly annotated at the edge to each role type. Collaboration and

role instances follow the Deurema instance notation. Therefore, both refer to their type, have an additional name and are labeled with the corresponding stereotype.

Collaboration Structure Example

Following the smart city example of this thesis, collaborations between adaptive vehicles are thinkable. If for example the traffic density increases or the driver wants to switch from a manual to an autonomous driving mode, the smart cars are able to communicate with each other and build platoons. Figure 5.45 depicts the structure of a **Platoon** collaboration type with its three roles **Leader**, **Follower**, and **Observer**. Deurema distinguishes between single roles (1), multi-roles (1..*), optional roles (0..1), and optional multi-roles (0..*). The leader role is an example for a single role, with the effect that each platoon collaboration must have exactly one participant that performs this role. Furthermore, a platoon can have multiple, but at least one, followers. Optional roles are not necessary to realize the collaboration but may support it in most cases. The observer role in the platoon is optional, but has additional knowledge and functionality to increase safety during the collaboration by optimizing the overall traffic flow in the smart city. For example, the leader is responsible for calculating the driving path and managing incoming or leaving platoon members. The observer collects additional information about the platoon surroundings or communicates with traffic controlling entities along the road to increase the available knowledge and support the leader by its planning activities. Thus, the observer is not indispensable for the core functionality of a platoon but may optimize the overall joint behavior. In contrast, an additional role in a collaboration may increase the communication or computation effort, which cannot be realized by all participating collaboration members and must be omitted in that case. If a mandatory role is not available, the collaboration cannot be realized.

The runtime situation on the right in Figure 5.45 shows one platoon instance named p with the required leader l_1 , an observer o , and three followers f_1 , f_2 , and f_3 . Because all mandatory roles are instantiated and in the range of the defined multiplicities, the platoon collaboration instance is valid. Beside the collaboration structure, the available knowledge plays an important role, which is discussed in the following section.

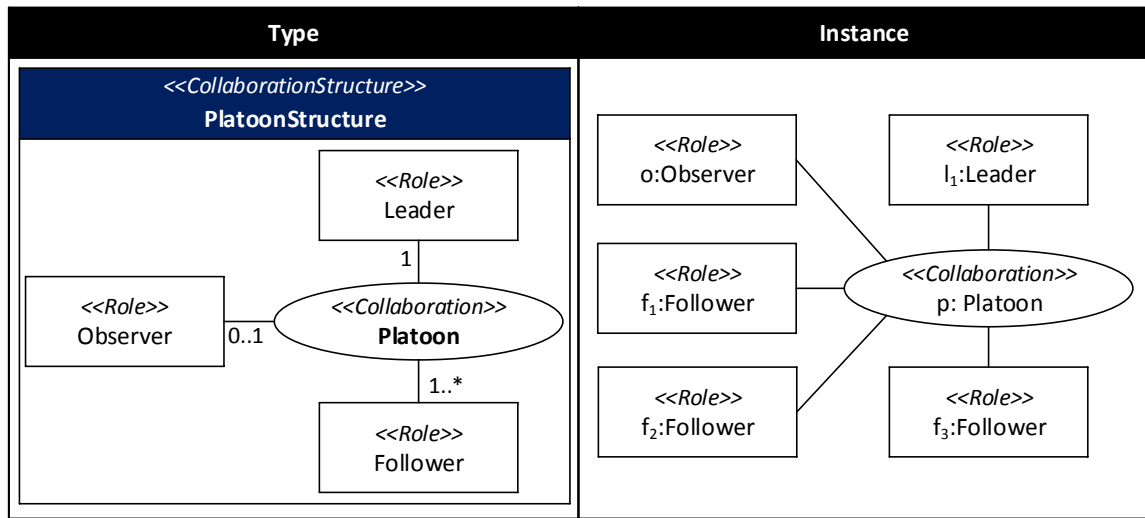


Figure 5.45: Collaboration structure example

5.5.2. Collaboration Knowledge

Defining the structure of a collaboration is a first step to identify all possible participants that interact with, and therefore influence, each other. Another important aspect is the sharing of knowledge to get the required information from other participants that is used to perform as expected within the role behavior of the corresponding collaboration as well as to improve the available information for the local adaptation behavior. However, exchanging knowledge can be broken down to the exchange of runtime models, which are modeled as first class entities in the collaboration knowledge specification. Thus, the knowledge specification explicitly determines the runtime model types and the amount of data that is involved in the collaboration defined by the local view. Deurema distinguishes three cases of runtime model treatment in a collaboration (cf. example in Figure 5.46). First, runtime models declared as *incoming* must be provided before the collaboration interaction starts. Thus, incoming runtime model information becomes visible in the local context of the collaboration. Because collaborating activities are integrated into local adaptation behavior, the local behavior must provide those runtime models, which are annotated as incoming, to the collaboration. Second, *internal* models are used in the collaboration context only and are not provided to the outside. Therefore, internal collaboration runtime models are not visible in the local adaptation context. Third, *outgoing* runtime models are provided by the collaboration and contain runtime model data, which is collected during the interaction of participants. Thus, the runtime models can be locally used after the collaboration and may contain updated information, which is retrieved during the interaction. The knowledge specification can be used for the design of the collaboration choreography, because it defines the available runtime models for the collaboration. Furthermore, the collaboration role interfaces and the collaboration mapping, which is the integration of the interaction into the local adaptation behavior, must conform to the knowledge specification.

<<CollaborationKnowledge>> Platoon			
<i>incoming</i>	<i>internal</i>		<i>outgoing</i>
<<MonitoringModel>> Monitoring Rules	<<SystemModel>> Convoy	<<ContextModel>> Environment	<<SystemModel>> Route

Figure 5.46: Collaboration knowledge specification

The example in Figure 5.46 defines four different runtime models for the platoon collaboration. The Monitoring Rules are declared as incoming and thus, must be provided for the collaboration. The rules are used by the follower role to retrieve the current environmental context as defined later in the concrete interaction specification of this role. The internal runtime models Convoy and Environment are used within the collaboration context only. The convoy model represents the current platoon and comprises the number of participants, position in the platoon, and status of the platoon member. The environment runtime model represents the retrieved context of the platoon, which may comprise several independent views from the platoon participants. Finally, the Route model contains the planned path of the platoon, which is optimized according the current convoy status and available environmental information. The route becomes visible in the local adaptation behavior and is provided/updated after each

performed collaboration interaction. Thus, the route runtime model becomes visible in the module, which performs the corresponding role as defined by the collaboration integration.

Note, the knowledge specification may evolve during the specification of the collaboration choreography. On the one hand, it is possible to derive the knowledge specification from the collaboration behavior by looking at all specified interactions. On the other hand, if the knowledge specification is modeled first, it can help the design of the behavior by restricting the interactions on basis of the available knowledge.

5.5.3. Collaboration Choreography

The choreography specification embodies the concrete interaction behavior by means of an interaction protocol. Therefore, the collaboration type, which is defined in the collaboration structure, refers to a set of interactions. In Deurema, interactions are specified by a type definition and interaction template for each role type in the collaboration as depicted in the metamodel in Figure 5.47. The interaction type can be seen as a protocol, whereas the interaction template defines the concrete steps of the protocol for each role. At runtime, interaction templates are instantiated and refer to their playing role. Thus, each usage of the protocol (interaction type) is modeled as interaction instance. The interaction template contains an arbitrary number of protocol steps, whereas the order is defined by the control flow. Furthermore, collaboration interactions are triggered by their realizing module or system. In the following the concrete syntax and an example for an interaction type are explained. Afterwards, the detailed concepts of defining the concrete interaction protocol steps in Deurema are introduced.

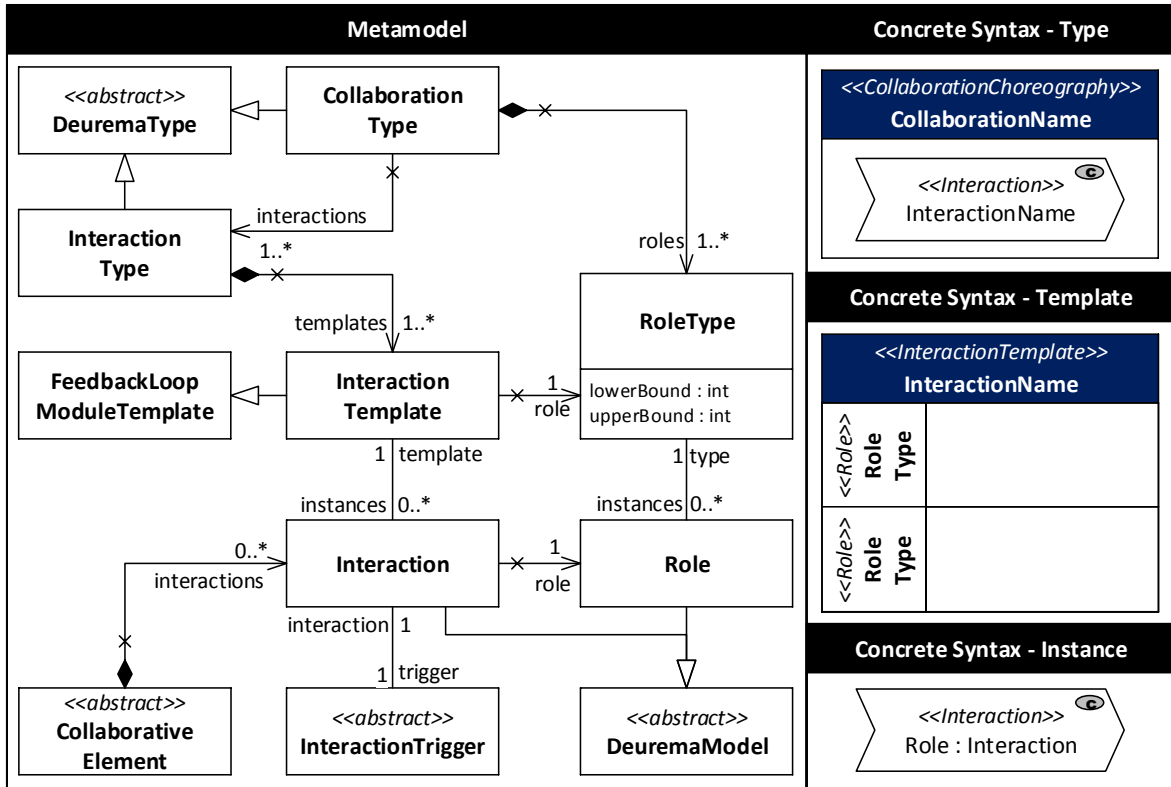


Figure 5.47: Deurema collaboration choreography

Concrete Syntax for Interactions

Because an interaction protocol comprises a set of activities (steps), it is modeled similar to activity variables, which is a labeled hexagon block arrow as shown on the right in Figure 5.47. Additionally, on type level, an interaction has the stereotype «Interaction», a name and an ellipse icon in the upper right corner of the block arrow, which is labeled with a *c* denoting the correspondence to a collaboration. The template definition of an interaction has a horizontal lane for each role type, where the interaction behavior is specified. Finally, interaction instances follow the Deurema instance notation.

Example Interaction Types

Figure 5.48 shows the interaction types (protocols) for the beforehand specified platoon collaboration. There are two interaction types HeartBeat and ShareEnvironment. The former protocol realizes a periodical notification of platoon members indicating proper following. The latter protocol defines the sharing of environmental information within the platoon, which helps the leader of planning the path of the convoy. An exemplary instantiation of the both interaction types is shown on the right in Figure 5.48.

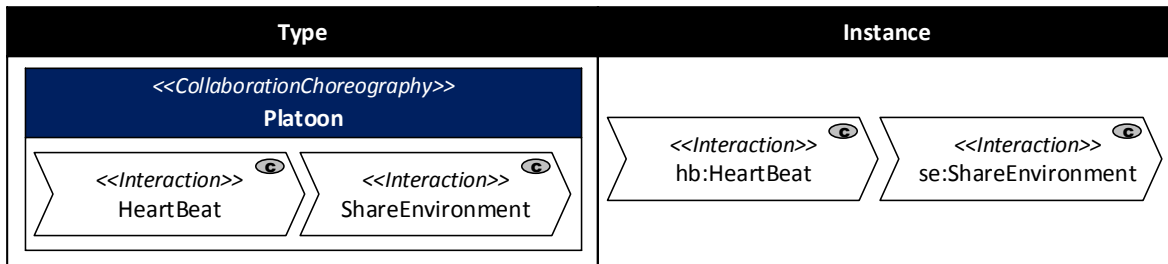


Figure 5.48: Choreography interaction example

Modeling Interaction Behavior

The detailed protocol steps are specified in interaction templates. Deurema extends the functionality of feedback loop templates as depicted in the metamodel in Figure 5.49. The `InteractionTemplate` class inherits from the `FeedbackLoopModuleTemplate` class, which facilitates the reuse of all Deurema concepts for defining feedback loops. Therefore, an interaction template can consist of adaptation activities, initial, final, and decision nodes, which allows the specification of the interaction behavior for each role. This comprises the modeling of the causal order defined by the control flow, and the usage of runtime models as discussed for feedback loops in Section 5.3.2.

For interaction templates, Deurema supports an additional message concept that allows the synchronization of interaction behavior between participants. Messages inherit directly from the `Operation` class and thus, can be seamlessly integrated into the template definition of the interaction. Deurema supports three different message types, whereas each type has a special purpose concerning interacting behavior. First, normal messages can be used for triggering and synchronization purposes of interacting activities between different roles. Second, model messages contain runtime model information and thus, are used for the exchange of knowledge. Third, services are visible, provided functionalities that can be invoked from other roles within the collaboration. Therefore, roles can offer or use services to delegate tasks among the collaboration. Due to services may realize a complex piece of functionality, they are considered as executable behavior models, whereas the realization follows the black-gray-white box semantic as comprehensively discussed for activities in Section 5.3.2.

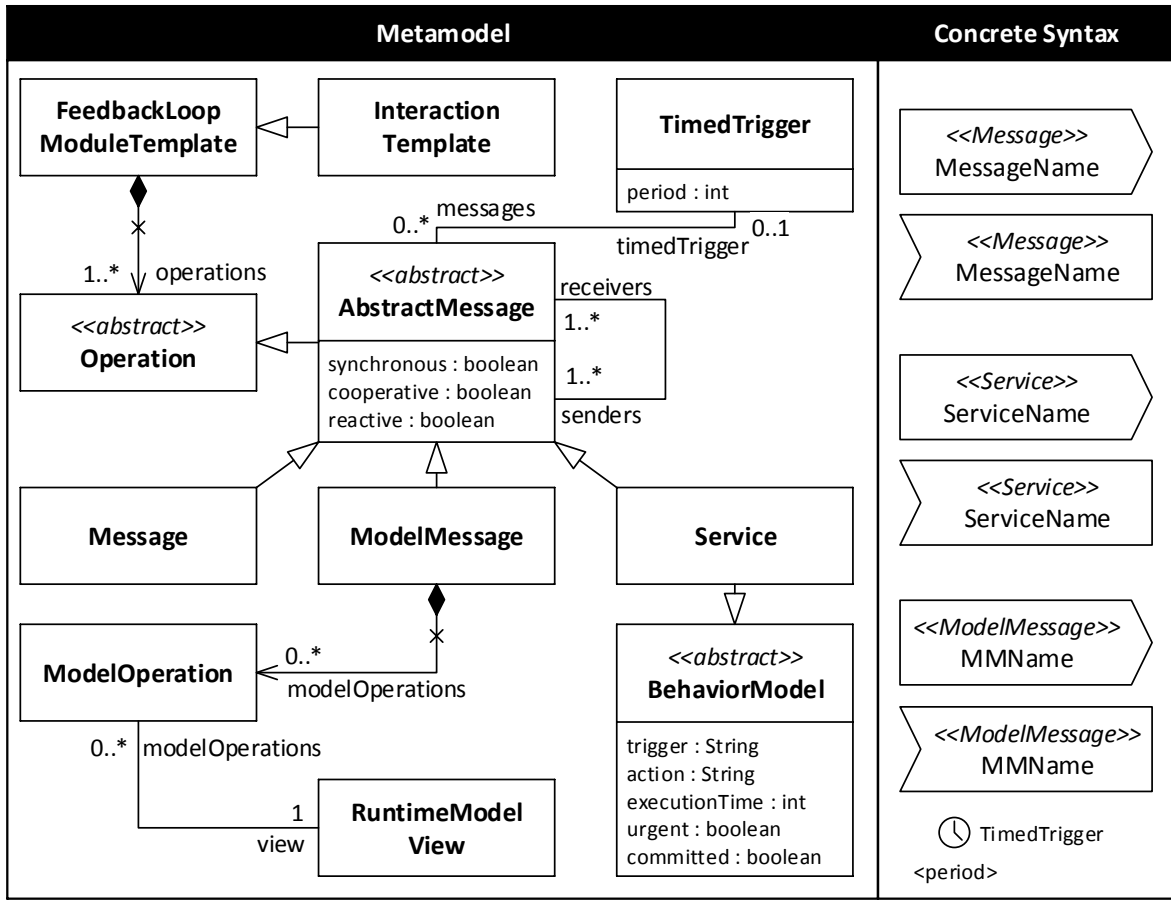


Figure 5.49: Deurema interaction templates

According to the Deurema metamodel, each message can have multiple senders and receivers, whereas additional attribute properties define different semantics of the synchronization respectively invocation behavior as discussed below. The sender and the receiver of a message must be located in separated lanes (roles) in the interaction template definition. Furthermore, messages can have a timed trigger that defines the maximal waiting time for the synchronization with other roles, which avoids waiting deadlocks in case of a missing coordination activity.

As shown at the right in Figure 5.49, messages are modeled as UML signals, where the sending and accepting of a message is indicated by the outgoing respectively incoming corner of the hexagon block. The message type is annotated as stereotype. Because there can be different messages during an interaction, the correspondence of a sent and received message is identified over the same name. The optional timed trigger is denoted by an additional clock symbol, where the waiting period is labeled under the symbol. In the following, an example for each message type is described. Afterwards, the semantic of the message properties are discussed in detail.

Interaction Template Example – Message

Figure 5.50 shows the template definition of the **HeartBeat** interaction as defined in the choreography specification in Figure 5.48. There are two lanes that specify the behavior of the follower and leader role of the platoon collaboration.⁴ Each role behaves independently from the other roles with the exception of a message exchange. In the example, the follower role sends an **Alive** message directly after starting the interaction. The leader is able to receive the **Alive** message at the beginning of its interaction. Furthermore, if the leader receives the message within a period of thirty seconds, it finishes the interaction. Otherwise, the leader performs an additional update activity on the **Convoy** system runtime model annotating the missing follower. The follower role directly finishes its interaction behavior after sending the message.

Note, the correspondence between sender and receiver is denoted over the message name. Furthermore, the message is denoted in template notation, where two different messages must have a different name. Conceptually, each role, which corresponds to a lane in the interaction template definition, can be instantiated multiple times. According to the example, the follower role can be played by more than one smart car within one platoon collaboration. Thus, each follower instance sends an **Alive** message, which is received by the leader role. If the leader waits for the alive messages from all followers before it continues its execution or if each alive message from each follower is processed individually, depends on the message properties, which are discussed below.

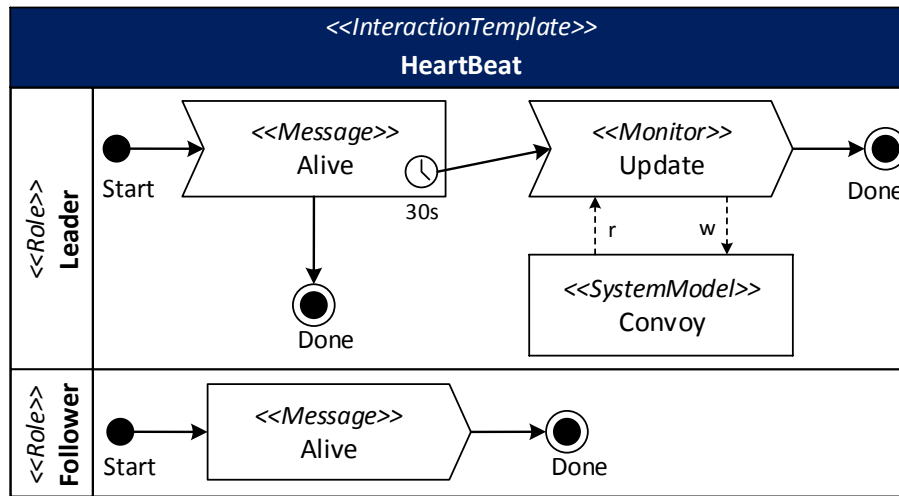


Figure 5.50: Interaction template example using a message

Interaction Template Example – Service

The usage of an **Alive** service instead of exchanging synchronization messages is shown in Figure 5.51 and looks very similar to the message example before. The interaction template has the same two lanes for the different roles. Furthermore, the execution order and timing constraints are the same. The difference is the execution semantic of a service. In contrast to a lightweight exchange of a synchronization message, which is fully controlled by Deurema, the leader provides an **alive** functionality that is invoked by the follower role. Services leave the boundaries of Deurema to application specific functionalities similar to adaptation activities.

⁴In this example, the optional observer role is omitted. Normally, an interaction template must specify the collaborative behavior of all role types.

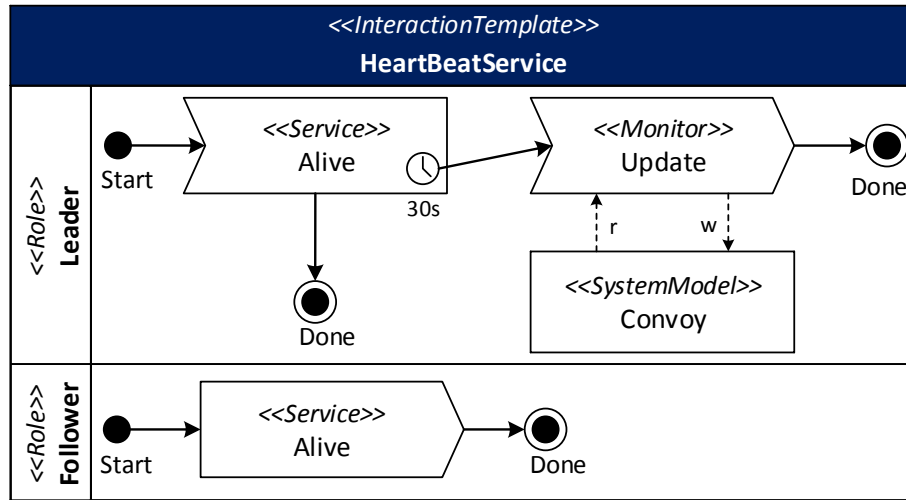


Figure 5.51: Interaction template example using a service

Therefore, the service is considered as Deurema behavior model following the black-gray-white box semantic. As a consequence of using a service, the computational load is deferred to the leader role. Moreover, the invocation of the service may cause changes in the state or runtime models of the leader. Similar to adaptation activities in feedback loops, if the service is defined as black box, the Deurema modeling language cannot predict the execution effects of the domain specific implementation, which is triggered by each service invocation. In contrast, if the internal details of the service are known (white box specification), Deurema can reason about the execution effects on the available runtime models of the service at development time.

The service and message examples show that the interaction points between roles can be modeled in different ways. It depends on the collaboration developer to choose the appropriate concept that fits best to the problem that has to be solved. Transferred to the example above, the possibilities of realizing a heartbeat protocol are first, using a lightweight synchronization message or second, a more complex service, which defines an additional application specific piece of functionality.

Interaction Template Example – ModelMessage

If roles want to share runtime model information during an interaction, they can use model messages as exemplarily depicted in Figure 5.52. First, the follower role performs an internal activity that scans the environment. Therefore, the `ScanEnvironment` activity reads a `Monitoring Rules` runtime model, which enables the access to physical sensors of the smart car. The retrieved information is written to the local `Environment` context model, which is shared via the `EnvInfo` model message afterwards. The leader waits up to thirty seconds for this information and finishes execution if the message does not arrive in that time frame. Otherwise, the leader performs an `analyze` activity, which checks the current route of the platoon, and optimizes it afterwards. Finally, the leader sends the optimized route to the observer role via the `ProvideRoute` model message. As defined in the Deurema metamodel and shown in the example, a model message must have at least one model operation to a runtime model for reading/writing the sent/received information from/to the local context of the corresponding role. Thereby, the handling of runtime models, the support of model operation types as well

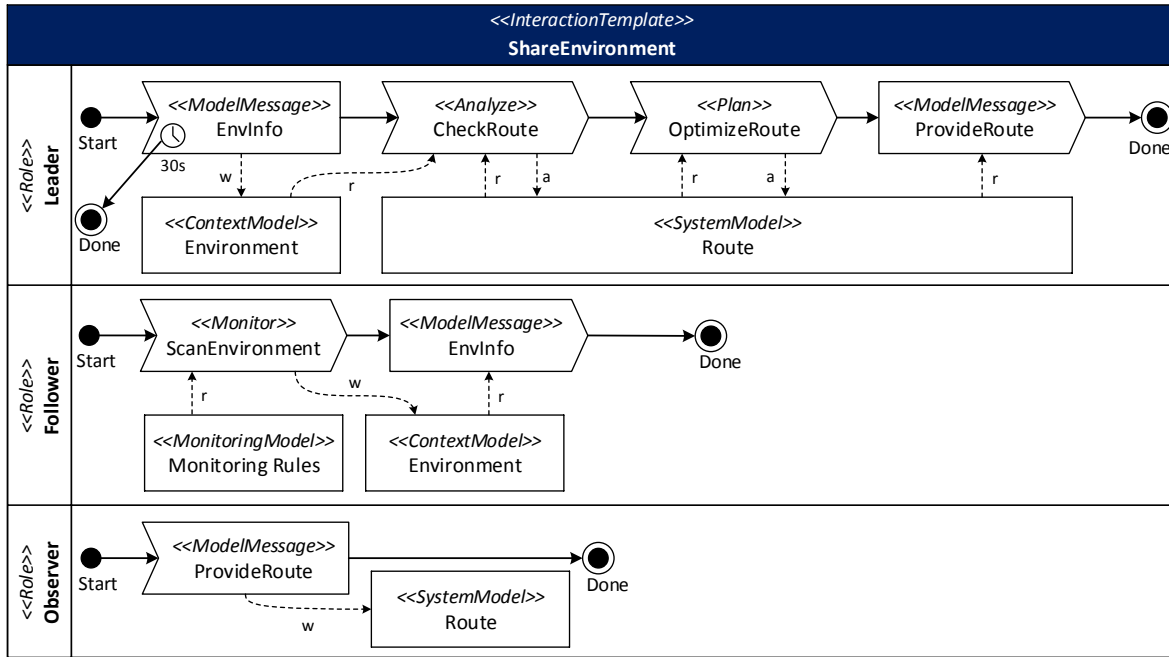


Figure 5.52: Interaction template example using a model message

as the model query concept are exactly the same as comprehensively discussed for feedback loop templates in Section 5.3.2.

In summary, all three examples show how activities, different message types and runtime models can be used to describe the interaction behavior for each role of the corresponding collaboration interaction protocol. Varying message, service, or model message properties lead to different synchronization semantics. In the following, each message property is discussed in detail.

Message Properties

As depicted in the `AbstractMessage` class in the metamodel in Figure 5.49, messages, services, and model messages are considered as abstract messages, which have three additional properties, namely *synchronous*, *cooperative*, and *reactive*. The first property refines the waiting behavior of a message sender, whereas the *cooperative* and *reactive* property refine the behavior of multiple sender respectively receiver. Therefore, the last two properties are useful to define the semantic for $1:m$ or $n:m$ message exchange. For the discussion concerning the message properties, the terms abstract message, message, model message, and service are used synonymously, because it works for all three message types exactly in the same way.

At first, a message can be synchronous or asynchronous. A sender of a synchronous message waits until the message is received by another role. In contrast, the sender of an asynchronous message directly continues its own execution without waiting for any confirmation. The Deurema execution framework ensures the buffering and consistency of the message until it is received by an appropriate role. The receiver of a message always waits until the corresponding sender emits the message, except an additional time trigger is specified as discussed in the examples above. Therefore, synchronous messages represent dedicated synchronization points within an interaction template specification. Combinations of asynchronous messages can be

used to delay the synchronization and executing additional local activities in between two asynchronous synchronization points as shown in the example in Figure 5.53.

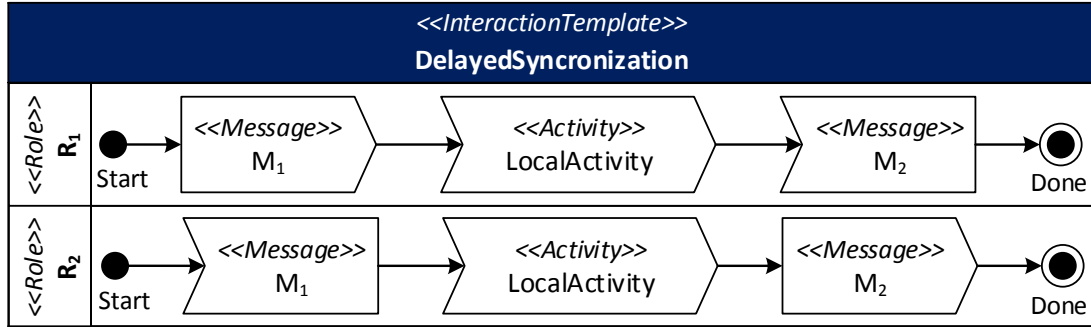


Figure 5.53: Delayed synchronization example

If the messages M_1 and M_2 in the example are synchronously sent, both local activities in each role can only be started after the message M_1 was sent and received. Furthermore, both roles synchronize their behavior again by exchanging the message M_2 . In contrast, if both messages are asynchronously sent, the role R_1 can emit the message M_1 and directly continue with the execution of its local activity afterwards. But, R_1 has to wait for the reception of M_2 . The role R_2 has the inverse behavior. First, it waits for the message M_1 , but can continue with the local activity and the sending of M_2 without waiting of any interaction of the role R_1 after the message M_1 is received. Therefore, the combination of two asynchronous messages is similar to a delayed synchronization of role R_1 , because it can continue local execution after triggering the role R_2 via M_1 and later synchronize its interaction with the other role by waiting for the reception of M_2 .

The reactive and cooperative property of a message defines the behavior of multiple sender respectively receiver and can be used orthogonal to the synchronous property. Consider the example in Figure 5.54, where four roles are depicted. There are two senders S_1 and S_2 of one message named Alive as well as two receivers R_1 and R_2 of the same message. Note, the same situation can appear if a role is instantiated multiple times as defined in the collaboration deployment specification, e. g., the follower role appears multiple times in the platoon example. For Deurema, there is no difference whether there are multiple senders/receivers in the template definition or instance situation, which send/receive the same message. All combinations are handled by the Deurema modeling language according to the following semantic.

The reactive property points to the execution semantic of the two senders and describes, whether all senders must emit the alive message so that the receiver can continue its execution or if at least one sender is enough. In the first case, each receiver must wait until all senders emit an alive message at least once (all semantic) before it can continue its execution. In the second case, it is enough if at least one sender (e. g., S_1) emits the message (first semantic), whereas the receiver can process the message and continue execution. Thus, the message is considered as reactive, if one sender is enough for message triggering. Therefore, a combination of a not reactive, synchronous message implies that the sender S_1 , S_2 emit the message and must wait until it is received by R_1 and/or R_2 .

In contrast to the reactive property, the cooperative property focuses on the receiver side. The message is cooperatively used, if all receivers must receive (handle) the beforehand emitted message (and semantic). Straight forward, the message is not cooperatively, if one receiver is enough for message processing (or semantic). Therefore, a not reactive, cooperative,

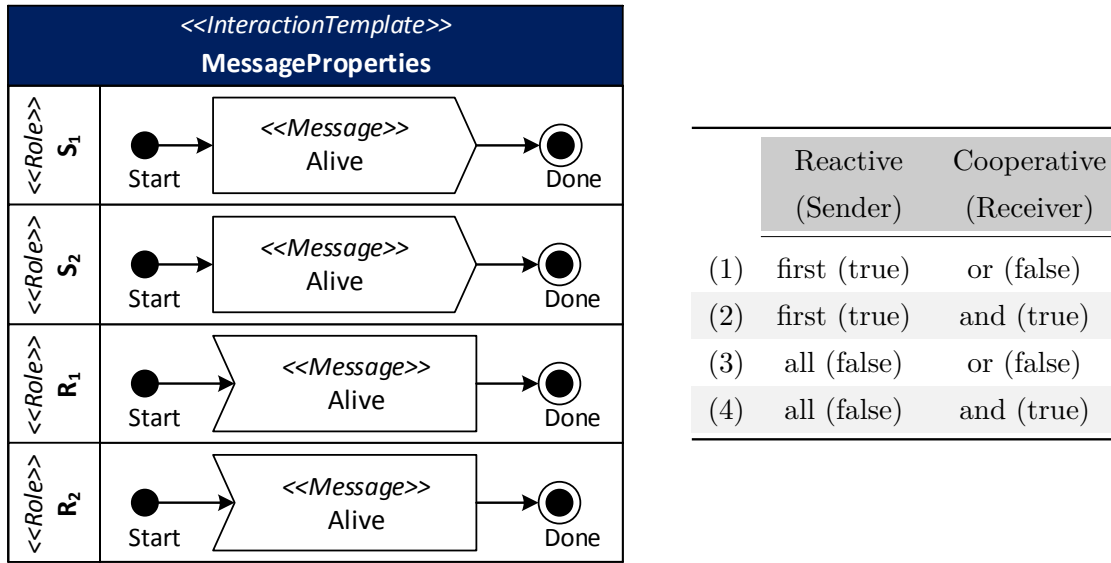


Figure 5.54: Message property example with reactive and cooperative combinations

synchronous message implies that both senders S_1 , S_2 emit the message and must wait until both receivers R_1 , R_2 process the message. Thus, this combination has the semantic of a synchronization point over the Alive message within the interaction.

The table on the right in Figure 5.54 shows the four possible combinations for the reactive and cooperative property. According to the explanation above, the corresponding semantic combinations are enumerated in the table, where the corresponding boolean values of the two properties are in brackets arranged behind. The combination (1) is similar to a XOR semantic, where one sender and one receiver are enough to exchange the message. Thus, the first sender/receiver pair can continue its execution after exchanging a synchronous message. In the asynchronous case, the sender can directly continue and the first appearing receiver will get the sent message. Note, if either senders or receivers send respectively receive the Alive message at exactly the same point in time, both are allowed to continue local execution. In the combination (2), all receivers must handle the Alive message, which can be sent by S_1 or S_2 . Straightforward, combination (3) requires the availability of both senders and at least one receiver. Finally, combination (4) can be considered as global synchronization point over all four roles, if the message is synchronously sent. Consequently, one difference between case (1) and (4) is that in the first combination two Alive messages can be handled during one execution round of the interaction, whereas in the fourth case maximal one message is exchanged.

The concrete handling of the message exchange depends on the timed execution order of the interaction role instance. For example, one sender and one receiver are enough for processing the message in the combination (1). However, due to the independent, concurrent execution of the roles, both senders/receivers may emit/process the alive message at exactly the same point in time, which is supported by the Deurema execution environment. Obviously, there is a difference in the overall runtime behavior if one receiver processes the message or both receivers get the message during the collaboration, which depends on the concrete timing situation during execution. A comprehensive overview of different execution orders considering timing aspects and the resulting effect of message handling is shown in the Appendix B.

5.5.4. Collaboration Role Interfaces

From the specification of the collaboration structure, knowledge and choreography, corresponding role interfaces can be automatically derived. Thereby, the visible interactions and their causal order are identified from the choreography specification for each role. Following the approaches of Neumann et al. [8] and Salah et al. [155], the interface description abstracts from local interaction activities that are not visible or could be hidden to the outside of the collaboration. The interface description comprises: First, the causal order of interactions defined by the control flow of the choreography specification, e.g., the message x is send before message y. Second, the shared runtime models as well as the model operations that are necessary for the interaction, e.g., reading a runtime model before sending it via a model message. Third, timing constraints that restrict the waiting time of an interaction, e.g., if a role waits at maximum x seconds for a message.

As an example, Figure 5.55 shows the three role interfaces for the Leader, Follower, and Observer for the ShareEnvironment interaction of the Platoon collaboration. Compared with the Figure 5.52, the interface hides local role activities such as checking and optimizing the route within the Leader role. Furthermore, from the internal use of runtime models is abstracted as for example the reading of the Monitoring Rules by the ScanEnvironment activity performed by the Follower role at the beginning of the corresponding interaction.

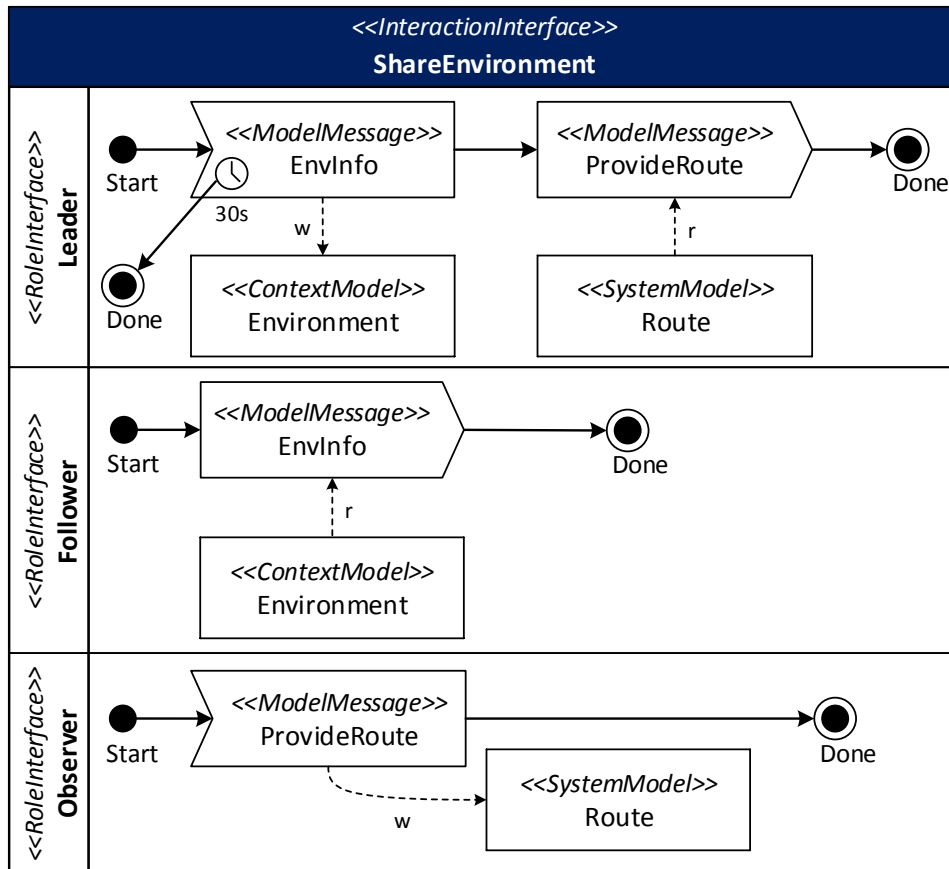


Figure 5.55: Interaction role interface

Thus, the interface of a role comprises the manipulation and exchange of runtime models, the causal order of visible interactions, and the invocation of services for the complete collaboration choreography. Automatically derived role interfaces can be used for analyzing the visible interaction behavior of the collaboration. If role interfaces are manually specified, they can be used for the modeling of the refined collaboration choreography. Thereby, the refinement of the interface can introduce local collaboration activities and knowledge handling, but must follow the overall defined interplay between roles. Because interfaces can be automatically derived from the choreography, the other way around allows the checking whether a refinement of a choreography specification still conforms to the beforehand manually created interface description or not.

5.5.5. Collaboration Role Mapping

After the separated development of the collaboration logic, the collaboration role mapping describes how collaboration interactions are integrated into the local adaptive behavior, e. g., feedback loop templates. Depending on the module template type, feedback loop operations, runnables, or rules are the corresponding behavior elements, which can trigger a collaboration interaction. Thus, interactions are woven into the module templates and appear as additional behavior. In application and behavior module templates, interactions are executed after the triggering runnable or rule. In feedback loop module templates, interactions are integrated into the control flow of the local operations, which is in fact before and after existing operations.

Figure 5.56 shows the Self-Configuring feedback loop template as introduced in Section 5.3.2 that integrates the additional interactions *ShareEnvironment* and *HeartBeat* from the platoon collaboration. In the example, the heart beat is performed directly after the local *Update* activity. Furthermore, the sharing of environmental information (*ShareEnvironment* interaction) is directly done after the heart beat (*HeartBeat*) interaction finishes. Both interactions perform model operations on local runtime model views. The model operation type conforms to the collaboration knowledge specification as outlined in Section 5.5.2. Therefore, the Monitoring Rules are read for the *ShareEnvironment* interaction and the Route runtime model is provided afterwards.

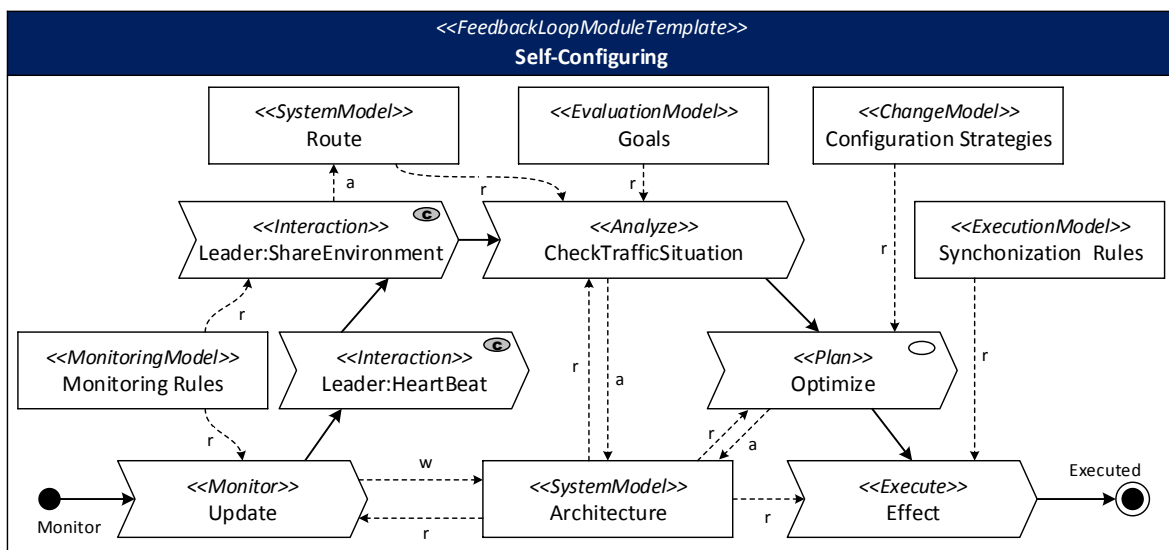


Figure 5.56: Collaboration role integration

Due to the interactions can be played in different roles, there can be individual module templates realizing each role. Thus, each role and interaction can be placed differently in separated templates. In the example, the template integrates the interactions for the **Leader** role, which is indicated by the role name for each interaction. Consequently, the leader role performs the **HeartBeat** and **ShareEnvironment** interaction directly after the monitoring step. The observer and follower role as specified by the collaboration structure definition in Section 5.5.1 is not integrated into the feedback loop module template shown in Figure 5.56.

This example shows the integration of interactions into FLD, the use of interactions in BRD and ACD follows the same integration concept. Furthermore, the access to local runtime model views must conform to the collaboration knowledge specification, where the local information becomes visible in the corresponding collaboration interactions.

Interaction Trigger

Due to interactions are integrated into the local adaptation behavior, the elements in the Deurema module templates can trigger the defined interactions at runtime. There are three module templates in Deurema that contain elements describing the adaptive behavior, namely feedback loop, behavior, and application module templates. As a consequence, operations (e. g., activities), rules, and runnables can trigger an interaction between collaborating modules as depicted in the metamodel in Figure 5.57. The triggering is realized by the inheritance of an abstract **InteractionTrigger** class, which has a reference to all interactions that have to be triggered.

In summary, the local behavior together with the integrated collaboration interaction realizes the adaptive behavior capabilities of the corresponding module template. The templates are blueprints that can be deployed (instantiated) in the layered architecture of a system template, which leads to the overall behavior specification of the adaptive SoS. The use of collaboration instances is described in the following.

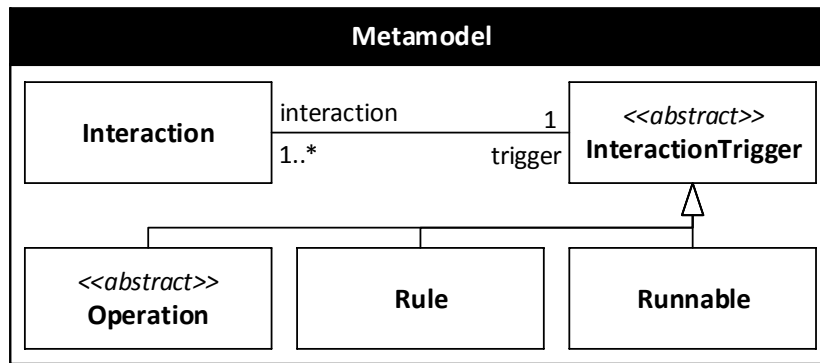


Figure 5.57: Deurema interaction trigger

5.5.6. Collaboration Deployment

The collaboration deployment defines the architectural instance situation of the system together with collaboration instances and role assignments. Therefore, Deurema extends the LD introduced in Section 5.4 by allowing module, system, and collaboration instances.

Collaboration Player

Collaboration roles must be realized by modules and integrated into the local adaptation behavior. Therefore, Deurema refers to an abstract player entity that behaves according to this role. The abstract entity is defined by the `CollaborativeElement` class in the Deurema metamodel in Figure 5.58. Each player can realize an arbitrary number of roles, whereas each role is assigned exactly to one player. Modules are the basic concept of executing the adaptation logic in Deurema. Therefore, modules are a candidate for playing roles within a collaboration. Furthermore, because Deurema considers adaptive SoS, systems can realize (play) a role. As discussed in Section 5.1, systems may contain other systems and modules. Due to this fact, Deurema supports the delegation of roles by systems to other subsystems or modules in the hierarchy.

Furthermore, Deurema supports the triggering of collaborations on different levels that are *before*, *after*, or *within (internal)* a module execution, which is defined by the `RoleTrigger` enumeration in the metamodel. The triggering of interactions before and after module execution enables black box modules, e. g., software modules as introduced in Section 5.3.3, to participate in collaborations. Of course other module types can also use the before or after trigger, if for example the integration of collaborative behavior is not wanted or disturbs the local adaptive behavior. Considering the platoon example above, if a feedback loop, which plays the leader role, uses the After role trigger, the local feedback loop behavior is executed first and afterwards the interactions as defined in the collaboration choreography specification. Straight forwards, for the Before role trigger, the collaboration behavior is executed first and afterwards the local feedback loop behavior. The internal role trigger is used, if the interactions are integrated into the module template as described in the former section for the feedback loop, which integrates the interactions of the platoon collaboration performing the leader role.

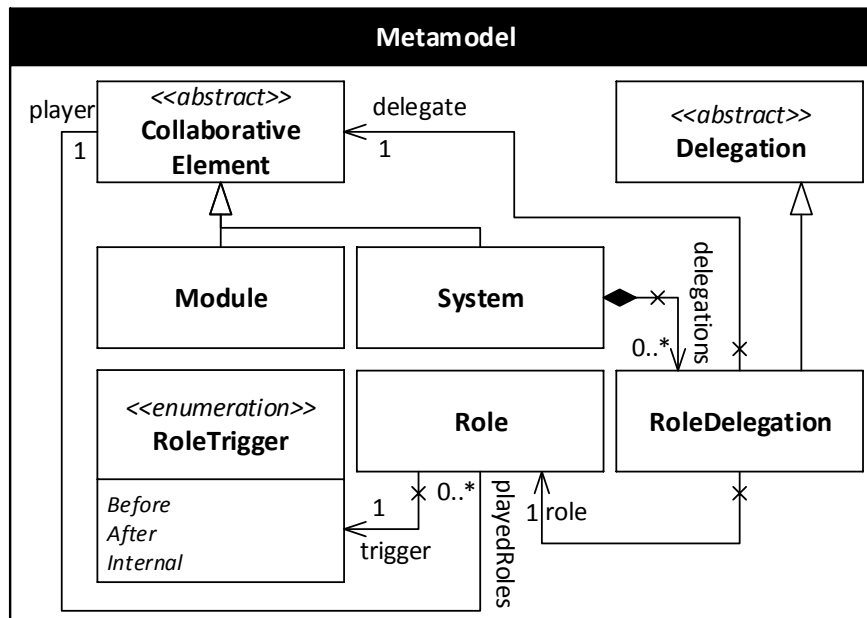


Figure 5.58: Deurema collaborative elements

Collaboration Deployment Example

Following the running example of this thesis, the LD in Figure 5.59 shows a refined system template specification of a smart city consisting of two smart cars in a platoon as introduced in the beginning of the collaboration section. Each smart car comprises a layered system architecture as comprehensively described in Section 5.4. Therefore, a smart car is shown at system instance at the lowest layer of the smart city in Figure 5.59. Both cars are supervised by an individual Self-Configuring feedback loop that integrates the collaboration interactions as depicted for the leader role in Figure 5.56.

Furthermore, the example shows one instance of the platoon collaboration with one instance of each role type. The mapping of role instances and their names are directly annotated at the corresponding module. Therefore, the feedback loop module sc_1 plays the Leader role of the platoon, whereas the feedback loop module sc_2 realizes the follower behavior. The optional observer is deployed in the TrafficOptimization feedback loop module. Depending on the integration of observer behavior, the traffic optimization feedback loop can use an appropriate role trigger that defines the playing of the observer role before, after or within the module execution. The LD shows the instances of the modules from an architectural perspective. Internals are modeled in the corresponding SMD, BRD, ACD, FLD, and collaboration interaction templates respectively.

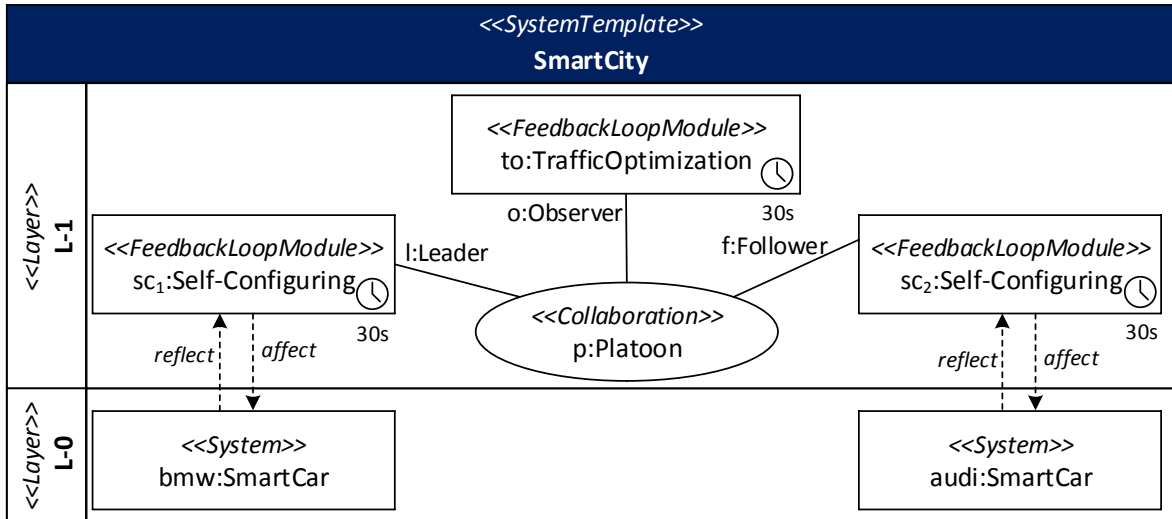


Figure 5.59: Collaboration deployment

System Role Delegation Example

Deurema considers systems as collaborative entities playing roles in a collaboration. Due to systems cannot be executed directly but rather consist of other systems and modules, they must delegate the playing role to modules or inner systems. Figure 5.60 depicts a small example, where two smart cities named potsdam and berlin interact with each other over a Traffic collaboration. For example, the cities may share the current traffic situation to predict the overall future traffic flow knowing possible incoming and outgoing vehicles. However, the collaboration instance defines that the smart city potsdam plays the Server role and the other smart city berlin acts as client. Both systems delegate their role to an internal TrafficMonitor behavior module. Consequently, the internal modules realize the specified roles of the parent system.

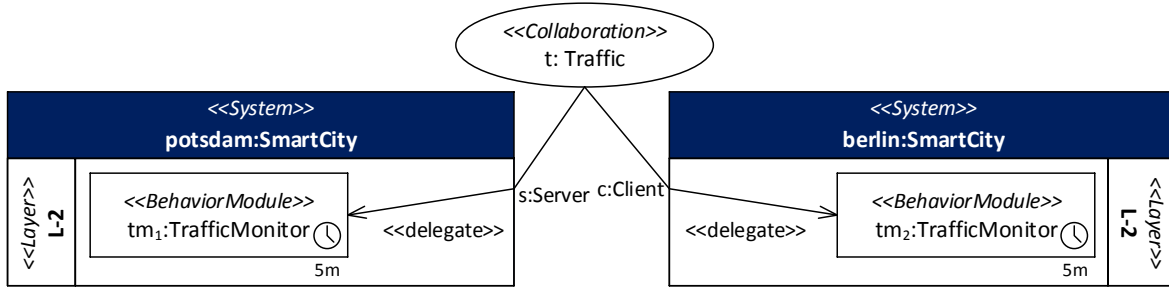


Figure 5.60: Collaboration role delegation

In general, the correct scheduling and triggering of the specified modules in systems together with their collaboration interactions is done by the Deurema execution environment. The Deurema collaboration concept can be used to define specific interaction protocols between modules and systems, which enable the exchange of knowledge, the invocation of remote services, or the synchronization of independent behavior in dedicated synchronization points. Normally, collaborations between systems and modules appear at the same layer in the corresponding system template, where the collaboration players are on the same level of abstraction. Furthermore, for retrieving information and influencing modules on different layers in the system template, the Deurema reflection mechanism can be used as discussed in the following. Additionally, the reflection mechanism is the key concept of further enabling the reconfiguration, adaptation, and meta-adaptation capabilities of Deurema.

5.6. Deurema Reflection, Reconfiguration and Adaptation

Up to this point, the modeling of the adaptive SoS architecture by means of system templates and the specification of system behavior using modules is discussed. Furthermore, interaction between systems and modules using the Deurema collaboration concept is outlined in the former section. As a consequence, almost all elements contained in the smart city running example sketched in Figure 5.61 can be modeled with the Deurema approach. Beside collaborations, which are used to model interactions on the same level of abstraction, systems and modules must reason about the underlying behavior of the system to realize envisioned self-* capabilities and thus, the adaptive behavior within the SoS. Therefore, this section introduces the Deurema reflection concept, which enables the reasoning about underlying modules and systems within the adaptive SoS architecture. Furthermore, system reflection enables the reconfiguration and adaptation of the underlying system behavior, which is also directly supported by the Deurema approach. As a consequence, module and system instances, which are placed on the layered system template, become enable to reflect runtime information from module and system instances at lower layers. On basis of this reflected information, those modules and systems are able to adapt the underlying behavior accordingly as highlighted in gray in Figure 5.61. As motivated in Chapter 3, there is a need that the adaptation engine itself must be adapted over time to cope with uncertain situations in highly dynamic system environments. Deurema supports a build in reflection mechanism to enable the reasoning about elements in the adaptation engine itself. This reflection mechanism further enables the application of system and module reconfiguration, adaptation, and the change of the adaptation logic on different layers in the adaptation engine, which is known as meta-adaptation as introduced in the preliminaries Section 2.1.1. In general, reflecting parts of the system template specification

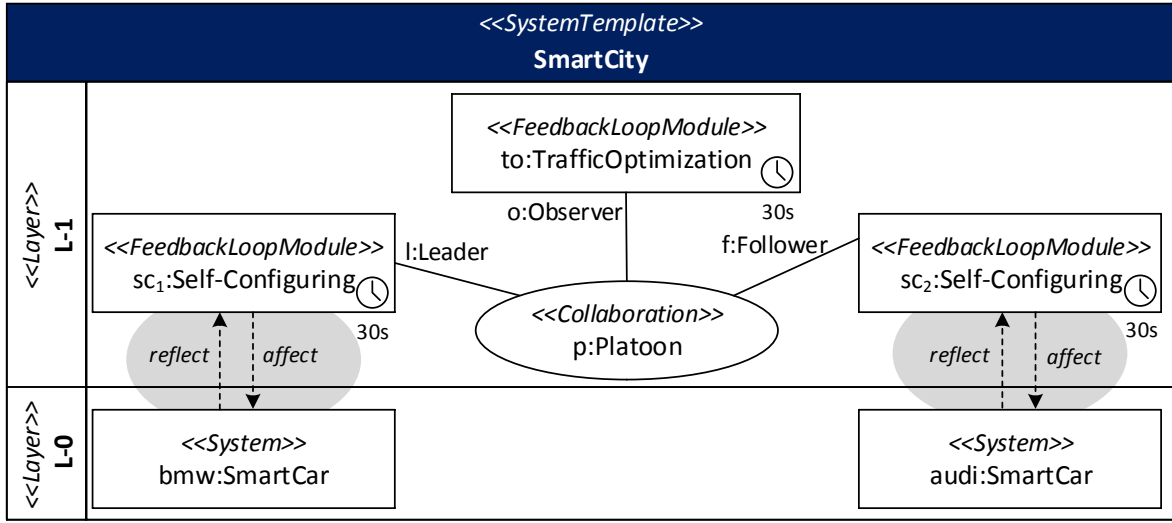


Figure 5.61: Smart city running example: Deurema reflection and adaptation

is enabled by the Deurema megamodel approach. Following the main idea from the MDE as motivated in Section 5.1, all Deurema elements are considered as model entities. Thus, the Deurema specification itself is contained and maintained by the megamodel. As a consequence, Deurema model information of interest can be retrieved (reflected) from the megamodel and represented as runtime models. Due to runtime models are available in the module template specification, the local adaptation behavior, e. g., feedback loop activities, can operate on the reflected information (the Deurema models) and reason about it.

The realization of the Deurema reflection concept is depicted in the metamodel in Figure 5.62. All Deurema elements that can appear on the layered system template specification can be reflected, which are modules, (sub)systems, and collaborations. The possibility of reflecting those elements is enabled in the metamodel by the inheritance from the abstract `ReflectiveElement` class. Furthermore, those reflective elements can perform module operations, which are specialized model operations. Module operations can be directly applied on other reflective elements. Therefore, the type of a module operation, defined by the corresponding enumeration in the metamodel, can be *Reflect* respectively *Affect*. The former denotes a module operation that retrieves (reflects) information from a reflective element, which will be locally accessible in a runtime model. The latter describes the other direction, where a manipulation of a runtime model leads to a change in the beforehand reflected element. Similar to model operations, module operations consist of model queries, which are used to specify the concrete parts that have to be reflected/affected. Model queries can be directly applied on the reflective element. Model queries are used in the same way as discussed for runtime models in module templates, cf. the description of feedback loops in Section 5.3.2.

Beside the reflection of information via module operations, the `ViewDelegation` class points to the runtime model view that holds the reflected information in the local module template. Therefore, the reflected information becomes visible as runtime model and can be used in the same way as any other runtime knowledge. Thus, the defined module operations together with the view delegation establish the causal connection, which is maintained by the Deurema execution environment. In the following, three examples are given to illustrate the use of reflecting and affecting module operations for each reflective element in the metamodel.

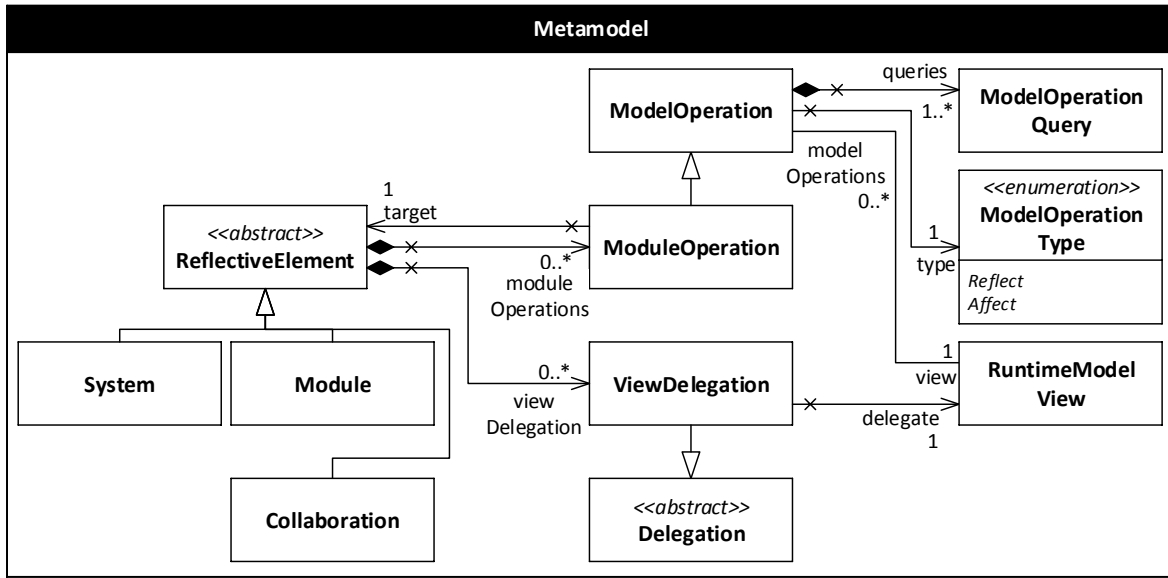


Figure 5.62: Deurema reflection metamodel

Module Reflection Example

Figure 5.63 depicts an example, where the Self-Configuring feedback loop module reflects and affects the smart car system. On the left side, the module and system instance as well as the corresponding module operations are depicted in concrete syntax. For illustrating the delegation of the reflected information, an excerpt of the Self-Configuring feedback loop module is modeled in the template notation on the right.⁵ As shown, the reflect module operation is delegated to the Architecture runtime model. Deurema ensures that every performed read model operation on the runtime model retrieves up-to-date, reflected knowledge from the architecture of the smart car system. The amount of available reflected data is specified by the defined model queries that belong to the module operation as shown in the metamodel in Figure 5.62. Furthermore, the amount of read data of the Update activity is defined by the model queries that correspond to the read model operation. Due to the causal connection, each modification of the Architecture runtime model, as done by the Optimize activity in the example, will cause corresponding changes in the smart car system. In this case, the runtime model modification is delegated to the corresponding affect module operation of the feedback loop module.

System Reflection Example

Similar to modules, Deurema systems can reflect and affect other reflective elements as well, which is shown in Figure 5.64. Additionally, systems do not perform adaptation activities directly, but rather encapsulate other modules and subsystems. Therefore, a system that reflects/affects another element must delegate the reflected/affected information to/from its contained modules or subsystems. Again, subsystems delegate the information to their contained children until a module takes the runtime information and delegates it to a runtime model view as discussed above for the example in Figure 5.63.

In the system reflection example in Figure 5.64, a smart car system reflects/affects an application module named Sensing. The reflected knowledge is delegated to an inner feedback

⁵For a comprehensive description of the complete self-configuring feedback loop see Section 5.3.2.

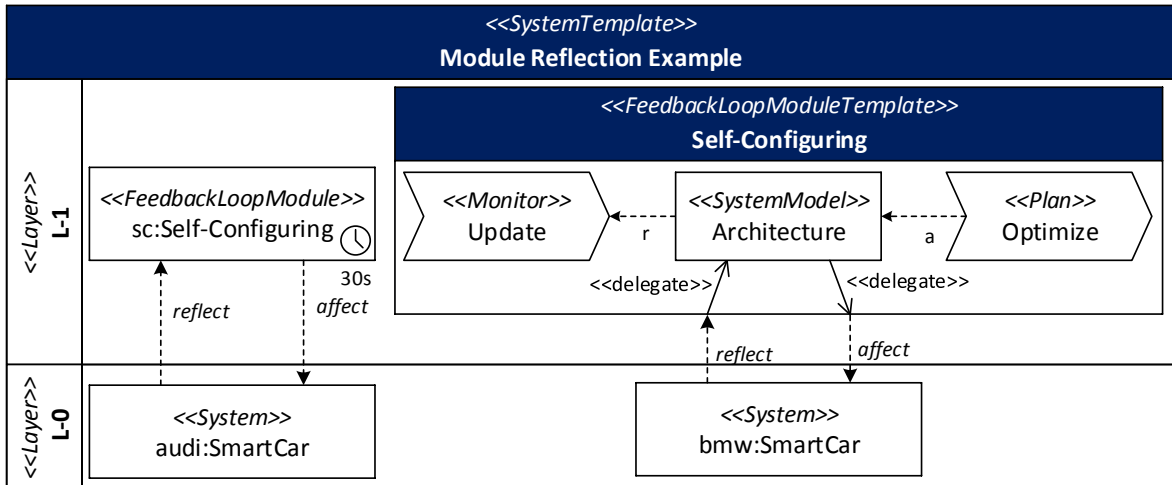


Figure 5.63: Module reflection example

loop module ABS and application component module AutonomousDriving. Thus, the reflected knowledge is duplicated or at least becomes visible multiple times in the car system, which is directly supported by Deurema. Furthermore, the ABS feedback loop module uses the reflected information as read-only, which is indicated by the missing affect delegation. In contrast, the AutonomousDriving application module reflects as well as affects the Sensing application module underneath.

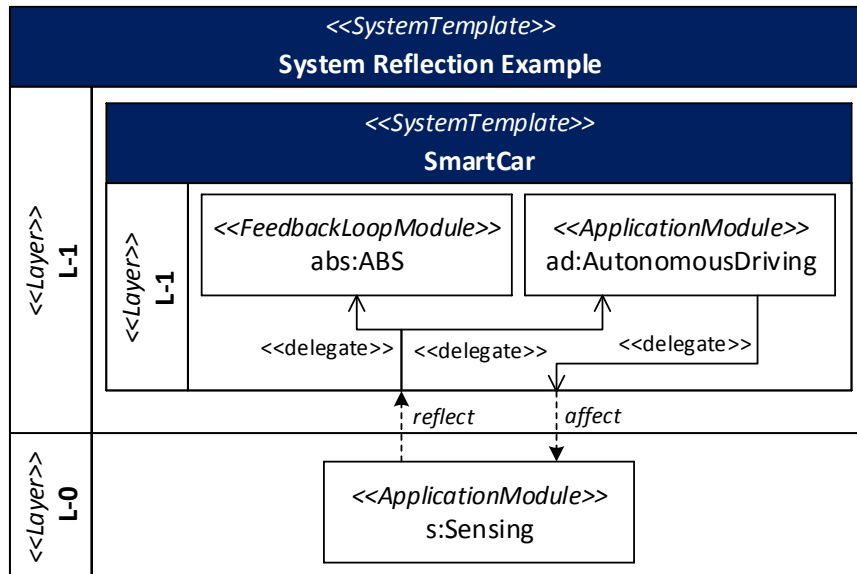


Figure 5.64: System reflection example

Collaboration Reflection example

The last example in Figure 5.65 considers a collaboration as reflective element. There are three pairs of a smart car together with their self-configuring feedback loop. Furthermore, two feedback loops interact with each other in the Platoon collaboration, whereas one feedback loop performs the leader role and the other feedback loop realizes the follower behavior. For

illustrating the reflection mechanism, an excerpt of the leader role lane is depicted in the interaction template notation directly in the platoon collaboration.

In this example, the figure describes a scenario, where a third smart car wants to join the existing platoon. In this case, it is possible for the leader role to directly reflect needed status information from the smart car system instance. The reflected information is delegated to the Convoy runtime model in the collaboration interaction. Remembering the knowledge specification of the platoon collaboration as explained in Section 5.5.2, the convoy runtime model is only used in the context of the collaboration interactions and does not become visible in the local context of the realizing feedback loop (sc_1 in this example). Therefore, the reflected information can be only used within the leader role during interaction. Straight forward, changing the reflected runtime information leads over the Deurema delegation concept to an appropriate change in the underlying smart car system. As a consequence, the reflect-affect mechanism can be used to integrate new members into the collaboration, which is in this example another smart car. Therefore, the affect operation must deploy the corresponding platoon interactions, which are appropriate to the played role, into the smart car system instance. Afterwards, the smart car can join the existing platoon as another follower.

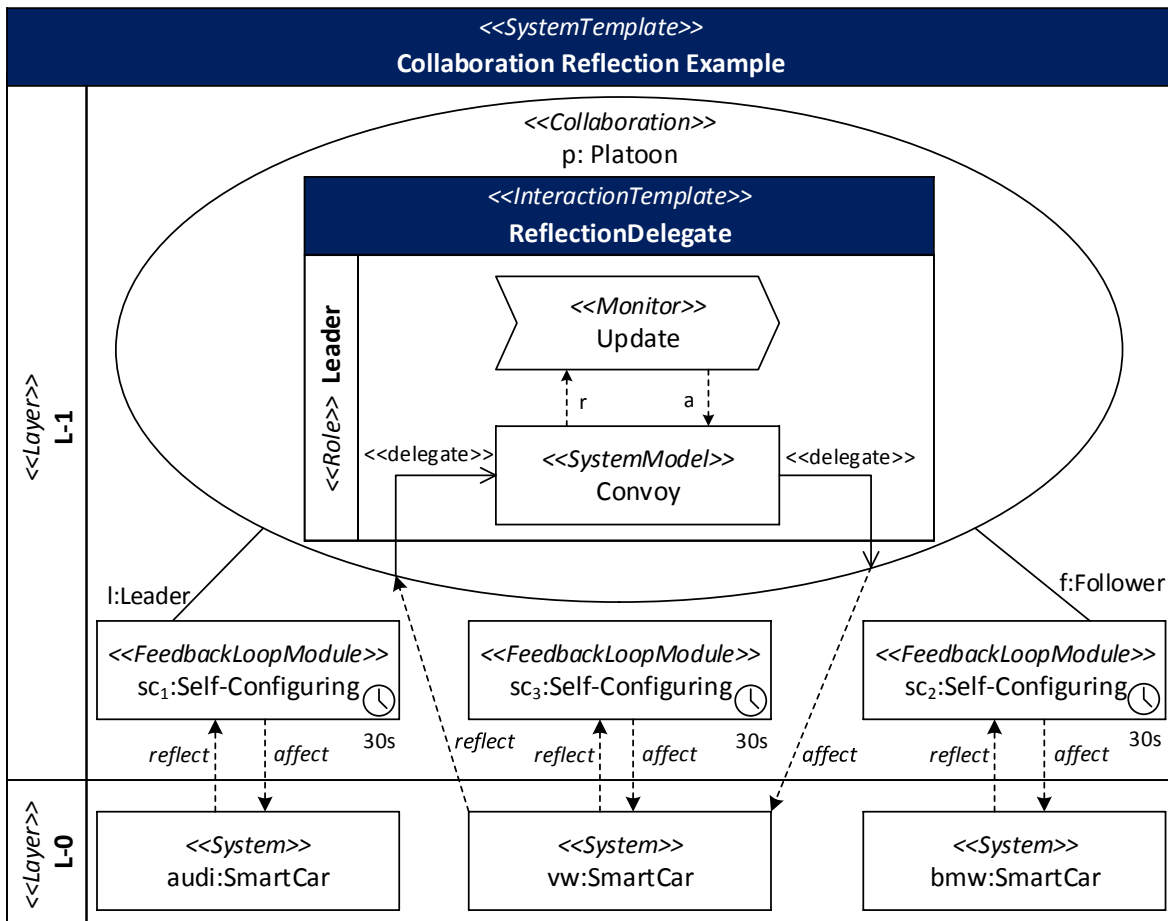


Figure 5.65: Collaboration reflection example

In summary, the reflection and affection of modules, systems, and collaborations are directly supported in Deurema. Reflected information becomes visible in form of runtime models in the local context of the reflecting entity, which enables a seamless integration. Thus, reflected information can be handled in the same way as normal runtime models across all supported template types as well as collaboration interactions in Deurema. In the following, the Deurema reconfiguration and adaptation capabilities are discussed.

5.6.1. Runtime Reconfiguration

As discussed in the preliminaries in Section 2.1.1, there are conceptually two ways for a SAS to change its behavior at runtime, which are reconfiguration and adaptation. Deurema supports both approaches, whereas for the reconfiguration approach the possible configuration space of the system must be specified upfront. This can be done in a variability runtime model as shown in the metamodel in Figure 5.66, which belongs to the corresponding adaptation runtime model categorization as discussed in Section 5.2.1. Furthermore, a variability model contains an arbitrary number of configuration types. Each configuration type refers to a variable type and corresponding possible parameters (assignments) for the variable, which span the possible configuration space. The variable types are predefined by the Deurema modeling language as already discussed for the different module templates in the former sections. Examples for variables types are activity variables, runnables variables or behavior rule variables. Each variable type can have multiple variable instances, which are contained by the corresponding module templates.

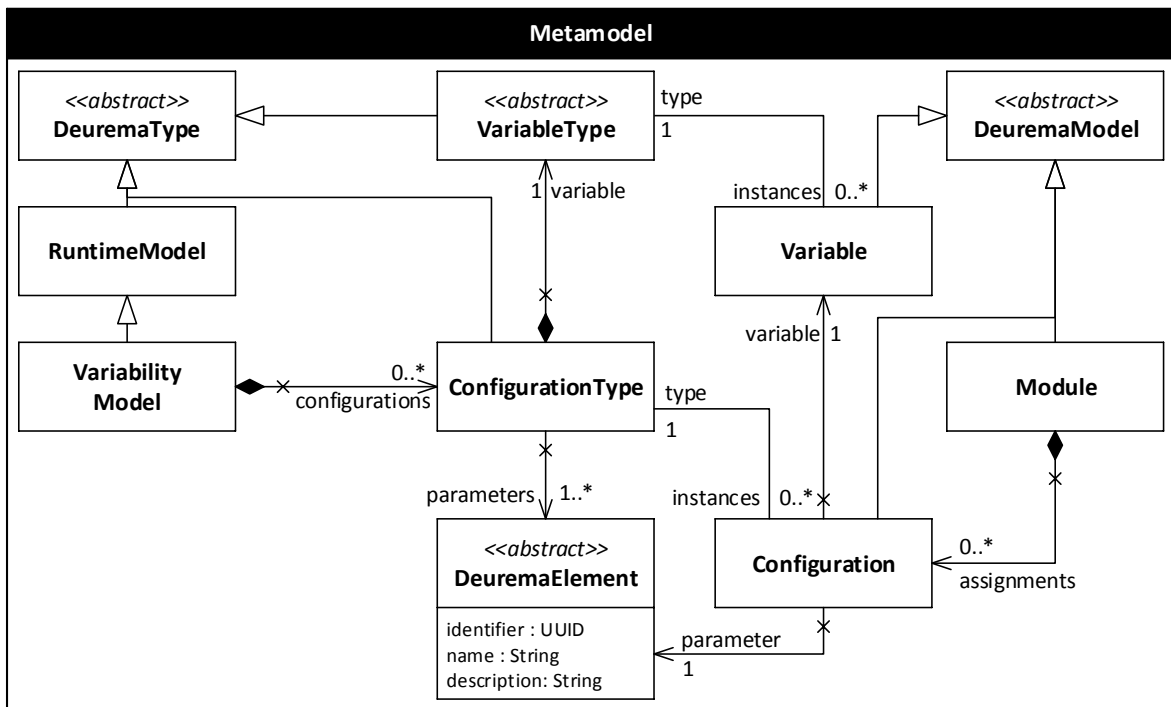


Figure 5.66: Deurema variability model

For example, an activity variable instance can be used within the specification of the feedback loop as discussed for the Self-Configuring feedback loop example in Figure 5.23, where an activity variable named *Optimize* defines a placeholder for different optimization strategies

(cf. Section 5.3.2). Additionally to the variables, modules contain configuration instances, which is the concrete deployment of variables with a parameter defined in the configuration space. Thus, a variable is defined by its type and is further assigned/resolved at runtime via a configuration from the Deurema variability model.

Figure 5.67 shows an example in the context of the smart car using all four possible variable types for an AutonomousDriving application module template. Although the example is for an application module template, the reconfiguration concepts work in the same way for behavior and feedback loop module templates. In the example, the AutonomousDriving application module template contains three sensor components, namely GPS, Distance, and Battery. The GPS sensor is able to retrieve the current position of the smart car, whereas the battery sensor can measure the power level of the battery. For the sake of simplicity, runnables that are not necessary to explain the reconfiguration concept are omitted in components, as for example in the case for the GPS and battery sensor. The distance component contains a runnable set variable, which enables the exchange of a set of runnables and is highlighted in gray in the figure. The task of this component is the detection of obstacles in the context around the car. Furthermore, all three sensor components read the monitoring model named Sensing, which enables the access to the physical sensors. Additionally, they update the Environment context model by annotating sensed information.

The SensorFusion software component uses the raw context information, transforms it and creates an environmental map representation. The software component contains a runnable variable to exchange the algorithm in different configurations. Furthermore, the example contains a component variable and a composition variable that use the map representation to optimize the behavior of the car. The former encapsulates different behavior strategies in one component and the latter allows the exchange of a complete subsystem that consists of several components. Finally, an actuator component reads the envisioned behavior and sends commands to physical wheels of the smart car.

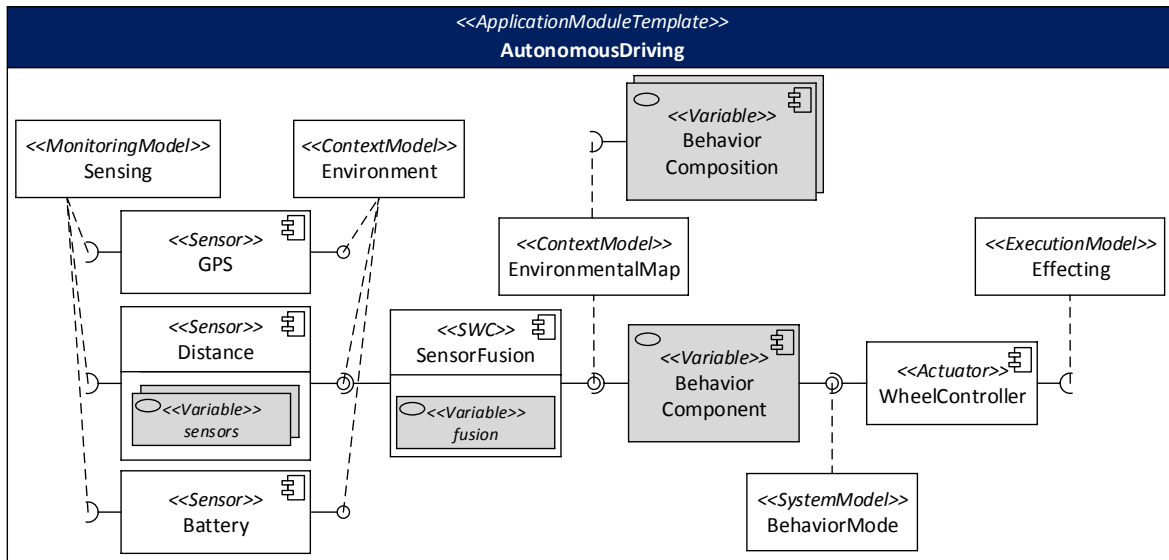


Figure 5.67: Reconfiguration example with gray highlighted variables

On basis of the application component template, Figure 5.68 shows the possible reconfiguration space of the modeled variables. The variable type, as defined by the Deurema metamodel, is depicted on the left. The concrete variable instance is shown in the middle and the possible configuration parameters (variable assignments) are depicted on the right in Figure 5.68.

The runnable variable, named *fusion*, allows three different runnable implementations realizing a *fast*, *basic*, and *advanced* fusion algorithm of the raw environment sensor data. The *sensors* set variable defines two different variants of an infrared and laser scanner runnable, which can be applied *indoor* or *outdoor*. In this example, there are two runnables each in the set definition and thus the number of runnables is equal for both sets. In the general case, this is no requirement for a set variable and Deurema supports unbalanced number of configurations, which enables a flexible modeling of the configuration space.

The component variable can be used to exchange a complete component. The configurations in the example show two variants, whereas the first can be used for adding autonomous driving functionalities in the car and the second realizes the charging of the battery cells at a power supply station. Therefore, the Driving component contains two runnables realizing a lane assistance and an adaptive brake (e.g., ABS or ESP), which requires the environmental map representation as well as the current route of the car. A necessary change in the behavior is annotated in the BehaviorMode system runtime model. In contrast, the second configuration defines a Charge software component that needs information about the battery level and provides information about the loading status of the battery cells.

Finally, the composition variable supports a configuration for car diagnosis, which can be used during car maintenance, and a configuration for normal operation, where the car calculates a route on the basis of the provided map. In the case for normal operation, the composition variable is resolved by two software components. In general, the number of components for a composition variable is not limited, which is useful for decomposing the application module template in reusable parts.

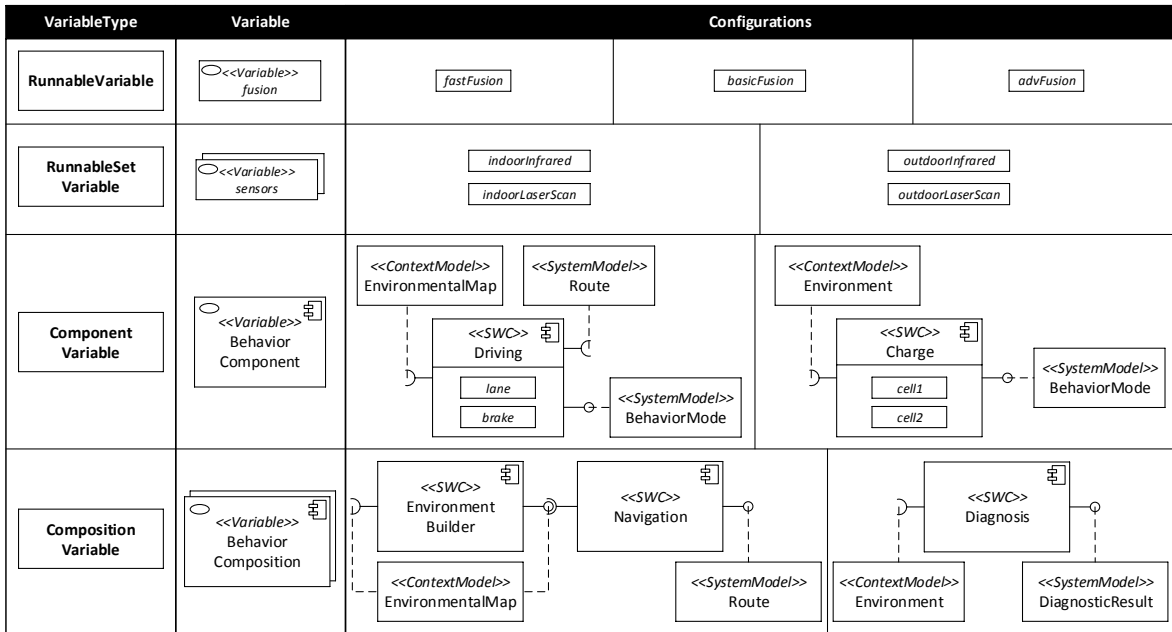


Figure 5.68: Reconfiguration space

After specifying the application template with the variables and the reconfiguration model, the modeler must specify the deployment of variable assignments for each module instance. Thereby, one configuration must be chosen to resolve all variables in the template definition. Figure 5.69 depicts one exemplary configuration (highlighted in gray) of the application module template example in Figure 5.67. The software component **SensorFusion** uses the advanced fusion functionality. The distance sensor is configured to use the infrared and laser scanner runnable for outdoor sensing. Furthermore, the behavior component variable is assigned to use the **Driving** software component, which introduces the additional runtime model **Route** into the template. Finally, the composition variable is resolved by the **Environment Builder** and **Navigation** software component and their corresponding runtime model usage. Consequently, changing the assignment of variables leads to other template configurations. Deurema supports the changing of the variable deployment at development-time and at runtime. In general, the configuration types define the available (predefined) configuration space together with its variable types. Variable instances can be used in the module template specification as placeholder for different template variants. Finally at module deployment, configuration instances assign concrete values to defined variables, whereas the available parameters are given by the configuration type in the variability model.

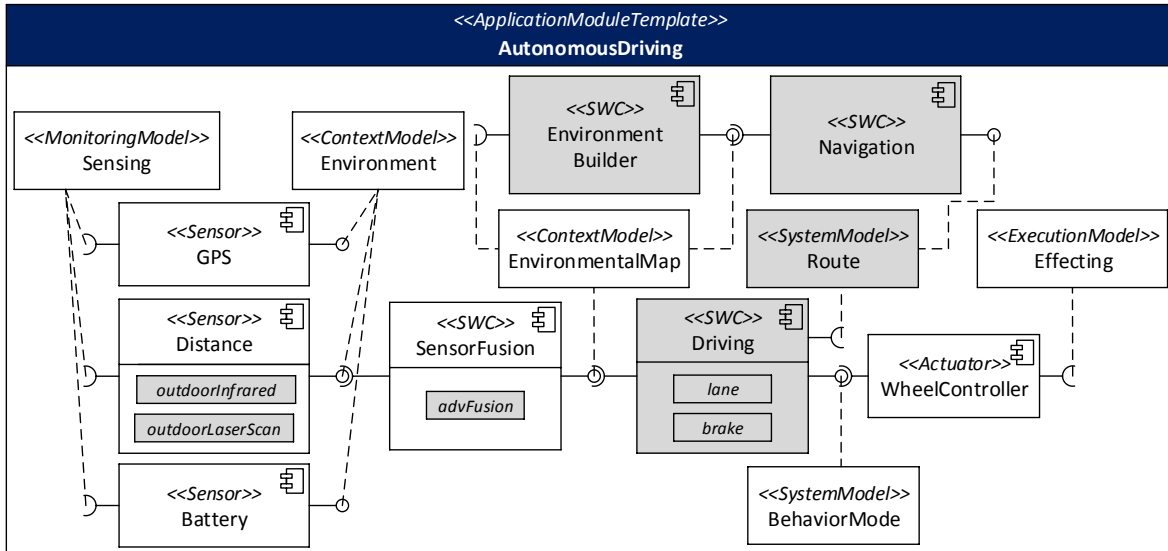


Figure 5.69: Resolved variable configuration highlighted in gray

5.6.2. Runtime Adaptation

Beside the reconfiguration capabilities as described in the former section, Deurema supports the dynamic adaptation of specified behavior. The dynamic adaptation capabilities are enabled by the Deurema reflection concept defined by the metamodel in Figure 5.62. Therefore, a feedback loop can directly manipulate an existing reflexible Deurema element, which can be for example a deployed module and its corresponding template definition. Thereby, the feedback loop must specify a corresponding reflect module operation, which specifies the concrete part that is reflected and makes this part visible as runtime model in the local feedback loop behavior. Afterwards, the feedback loop can manipulate the runtime model, which has a corresponding effect on the underlying, reflexible element.

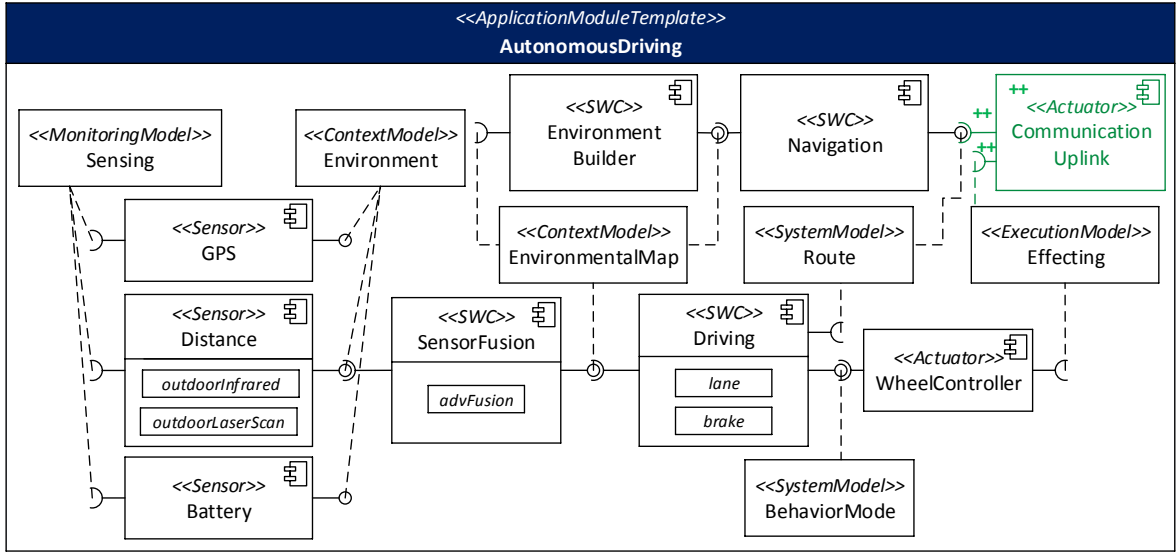


Figure 5.70: Deurema adaptation

An example for an applied adaptation effect in the **AutonomousDriving** application module template is shown in Figure 5.70. Additionally to the configuration explained in the previous section, an actuator component named **CommunicationUplink** is created, which is denoted by the ++ sign in the figure. The actuator reads the **Route** runtime model and provides it over a wireless communication channel to the outside, e.g., to a public station that collects data about the current position and route of cars. However, because the actuator component is not defined in a configuration, it is added via an adaptation activity performed by another Deurema module, e.g., the self-configuring feedback loop on top of the smart car. Furthermore, because changes enforced by an adaptation cannot be foreseen in the general case, the analysis of the adaptation effects is difficult. Consequently, the Deurema modeling language gives no guarantees for the correctness of arbitrary adaptation changes. If the adaptation operations are known, e.g., in form of graph transformation rules kept alive as runtime models, the adaptation effects are analyzable and Deurema can reason about possible adaptation effects. However, the Deurema execution environment enables the application of adaptation activities changing arbitrary Deurema elements and handles arising race conditions of changes in the underlying models. If the adaptation operation leads to a valid Deurema model, the change will be considered during the execution.

Using system reconfiguration or adaptation depends on different factors. The specified reconfiguration space and therefore, the effects of applying given configurations can be analyzed before the real reconfiguration happens at runtime. Furthermore, the application or change of a configuration is directly supported by Deurema, which ensures a correct deployment even if bigger changes have to be applied. For example, the exchange of several components as enabled by a composition variable implies the deployment of several Deurema elements such as runnables, a task mapping, ports, and corresponding runtime model views. For realizing the same effect using system adaptation, the developer has to ensure that the adaptation operations are applied in the correct order and create valid Deurema models. Furthermore, the developer must ensure that the adaptation process is not interrupted by other modules, which might cause inconsistencies. However, due to the predefined configuration space, reconfiguration may not be applicable for the need of dynamic decisions at runtime or if the

reconfiguration space is impractically large. In this case, system adaptation is more flexible. Moreover, known white box adaptation rules can be analyzed as well. Finally, the adaptation mechanism suits well, if new adaption strategies are derived at runtime and cannot be planned during the development of the adaptive system.

5.6.3. Meta-Adaptation

Meta-adaptation is a powerful concept that enables a runtime evolving of the adaptive behavior inside the adaptation engine itself. Applying the Deurema reflection mechanism on higher layers in the system template definition leads to a realization of the meta-adaptation concept. Thereby, the reflection mechanism is combined with the Deurema adaptation capabilities, whereas the basic principles are already introduced at the beginning of this section. Deurema modules, systems and collaborations are the three reflective elements defined in the metamodel. In the following, a meta-adaptation example for each of those element types is given.

Module Meta-Adaptation Example

A meta-adaptation example for a Deurema module is depicted in Figure 5.71. At the lowest layer L-0, two smart car instances are deployed that are optimized by the self-configuring feedback loop at the layer L-1 above. Up to this point, each pair of a smart car together with its feedback loop follow the Deurema reflection mechanism as explained at the beginning of this section. Therefore, the Self-Configuring feedback loop is aware of the internals of the underlying smart car, which is represented in the local runtime models. On basis of this information, the feedback loop can adapt the behavior of the smart car system underneath.

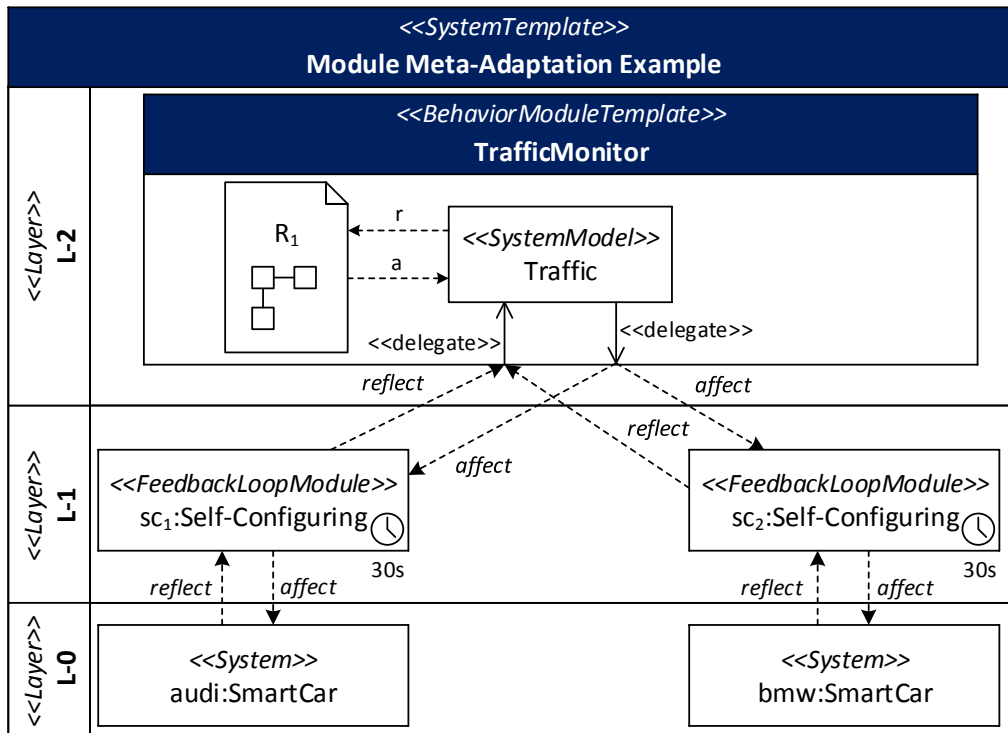


Figure 5.71: Module meta-adaptation example

Meta-adaptation is realized by the TrafficMonitor behavior module at the highest layer L-2 in this example in Figure 5.71. The behavior module reflects both feedback loops and thus, parts of the adaptation engine itself. The reflected information becomes visible in the Traffic runtime model and can be accessed as well as manipulated accordingly. For example, the modeled declarative rules in the behavior module template can monitor the Traffic system model to find violations or analyzing the current traffic situation in the smart city. Each change in the runtime model representation is delegated to the affect module operation, which synchronizes the underlying feedback loops. Thus, the behavior module can adapt the feedback loops, e.g., by exchanging an activity or the goals of the feedback loops, to optimize the overall treatment of a smart car. As emphasized by the example, the difference between adaptation and meta-adaptation depends on the layer in the system template. Conceptually, the same reflection mechanism, the representation as runtime model, and the delegation of effects are used in Deurema.

System Meta-Adaptation Example

Beside modules, meta-adaptation is also possible for systems within the adaptive SoS as shown in Figure 5.72. The example considers a smart car, which is parked in a garage of a smart home and plugged in a power supply station to load the battery. In this case, the smart home must reason about the behavior as well as adaptation capabilities of the smart car.

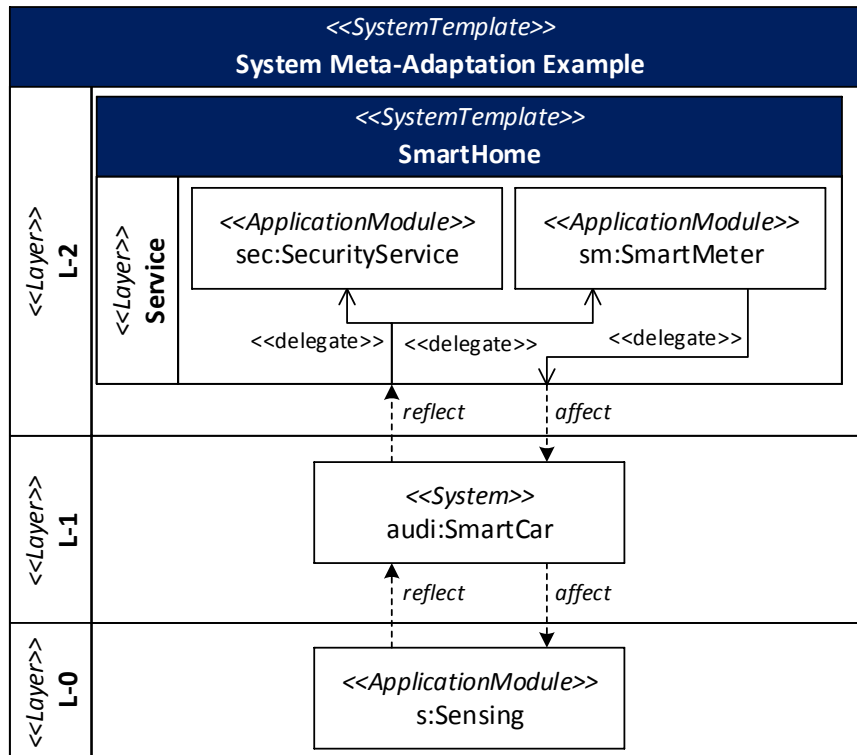


Figure 5.72: System meta-adaptation example

As shown in Figure 5.68, the smart car can reconfigure its behavior towards the charging of its batteries. If the smart home wants to change the general charging function according to the specifics of the power supply station, it must first reflect the car system to retrieve the desired information. In this case, the smart home can for example look at the variability model of the AutonomousDriving module to reason about the reconfiguration possibilities of the

smart car. In a next step, this information is transferred to an internal security service and a smart meter application within the smart home system. The security component uses the information as read-only and thus, is aware of the fact that a car is currently parked in the garage. In contrast, if the smart meter application decides that the charging capabilities of the car can be optimized, it can directly manipulate the corresponding runtime model, where the causal connection ensures the synchronization with the smart car instance below.

Collaboration Meta-Adaptation Example

The third and last meta-adaptation example in Figure 5.73 shows the use of reflected information in a collaboration. Therefore, the platoon collaboration example as described above is extended by a third layer. At this layer, a TrafficData collaboration instance is deployed together with two TrafficMonitor behavior module instances that play a Server respectively Client role. For illustration purpose, an excerpt of an interaction part of the server role is depicted inside the collaboration.

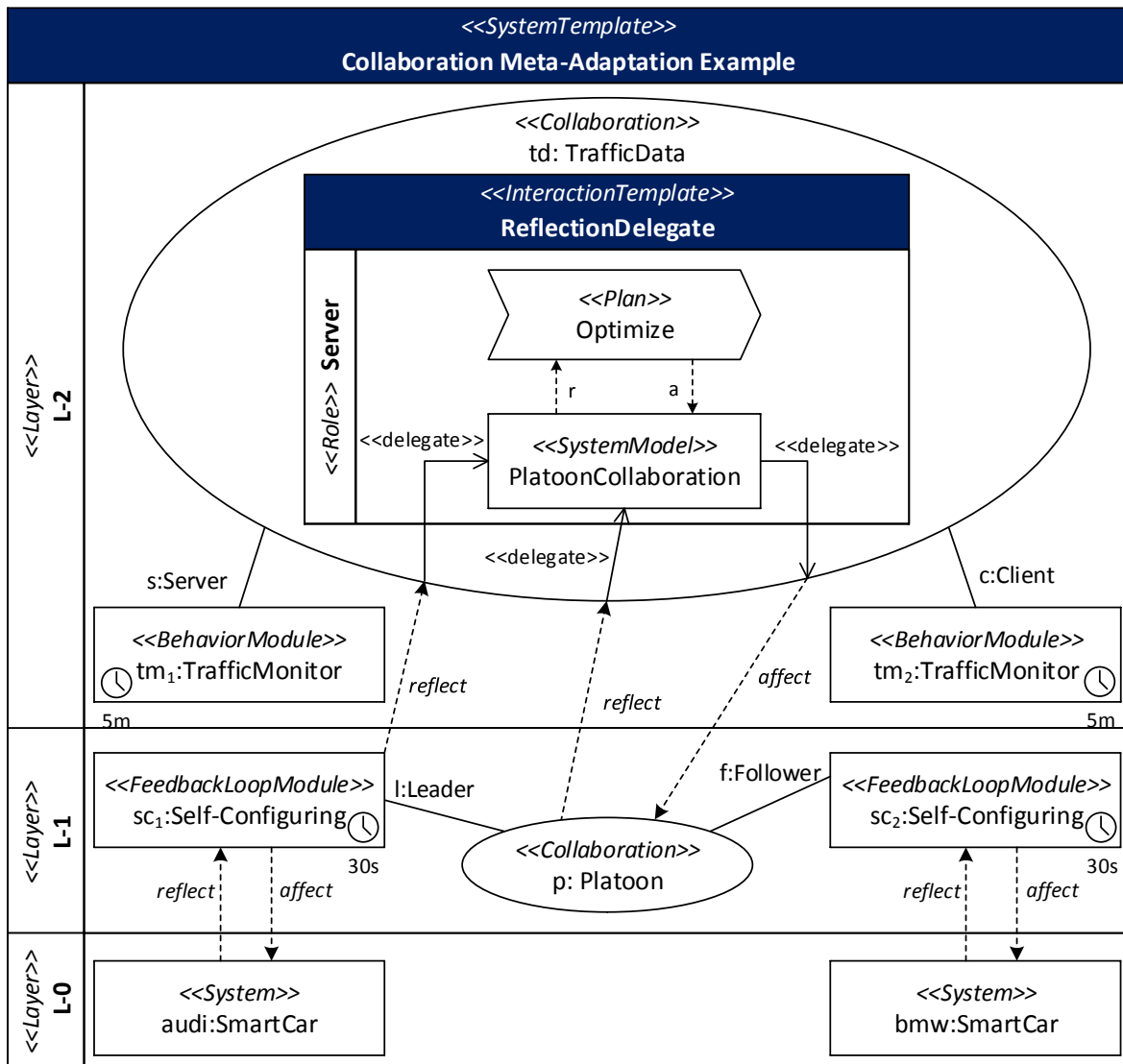


Figure 5.73: Collaboration meta-adaptation example

The reflection and affection in the collaboration example follow the same rules as explained for the system and module meta-adaptation example. Thus, the reflected Self-Configuring feedback loop module and Platoon collaboration are delegated to a runtime model view in the TrafficData collaboration. More precisely, collaboration interactions contain activities that further manipulate runtime model views. Therefore, the reflected knowledge is delegated to those collaboration interactions. In the example, there are two reflect module operations that are delegated to a combined view in the PlatoonCollaboration runtime model. In general, Deurema supports arbitrary combinations of reflect module operation and the delegation of this information to one or multiple runtime model views. Furthermore, the Optimize activity in the example affects only parts of the reflected knowledge, which is the underlying Platoon collaboration.

Reconfiguration and Adaptation Discussion

In summary, Deurema supports reconfiguration of the adaptive behavior that is defined by different variable types in the template definitions. Furthermore, variables can be configured by choosing a variant as defined in a variability model. Beside reconfiguration, Deurema facilitates system adaptation and meta-adaptation by providing reflecting and affecting module operations. The reflected knowledge becomes visible in the local template respectively collaboration interaction definition due to the delegation concept, whereas Deurema maintains the causal connection. Deurema considers reflection/affection relationships between elements as first class entities that enables further analysis of adaptation effects. Furthermore, the collaboration and reflection mechanism are two powerful concepts to enrich the knowledge about other modules or subsystems. On the one hand, collaborations should be used between modules at one and the same layer in the system, which indicates an interaction on an equal level of abstraction. On the other hand, the reflection mechanism is designed to enrich the local knowledge about modules, systems, and collaborations of the layer below, which can be manipulated afterwards. However, Deurema does not restrict the developer to use both concepts in exactly this way. Therefore, collaboration can be used between modules on arbitrary layers. Furthermore, the reflect/affect mechanism between two reflective elements on one and the same layer will work in the same way as outlined above. Using the collaboration, reconfiguration, and adaptation concepts of Deurema leads to different design decisions by modeling the adaptation engine, which may result in bad system architecture design.

5.7. Deurema Modeling Language Discussion

This section summarizes the concepts of the Deurema modeling language. At first, a comprehensive smart car and smart city example subsume the introduced concepts above to a multi-layered adaptive system. Thereby, the dependencies between modules, (sub)systems, and collaboration are explicitly modeled. Afterwards, important design decisions are discussed that influence the concepts of the Deurema modeling language approach. Beside already determined design decisions, possible extensions of the modeling language are outlined by pinpointing to the appropriate concepts. This section closes with a discussion about the coverage of modeling language requirements by Deurema as derived in Chapter 3.

5.7.1. Summary of Deurema concepts

The overall adaptive behavior of the smart car running example consists of three layers as depicted in Figure 5.74. The lowest layer L-0 represents the control software for the physical

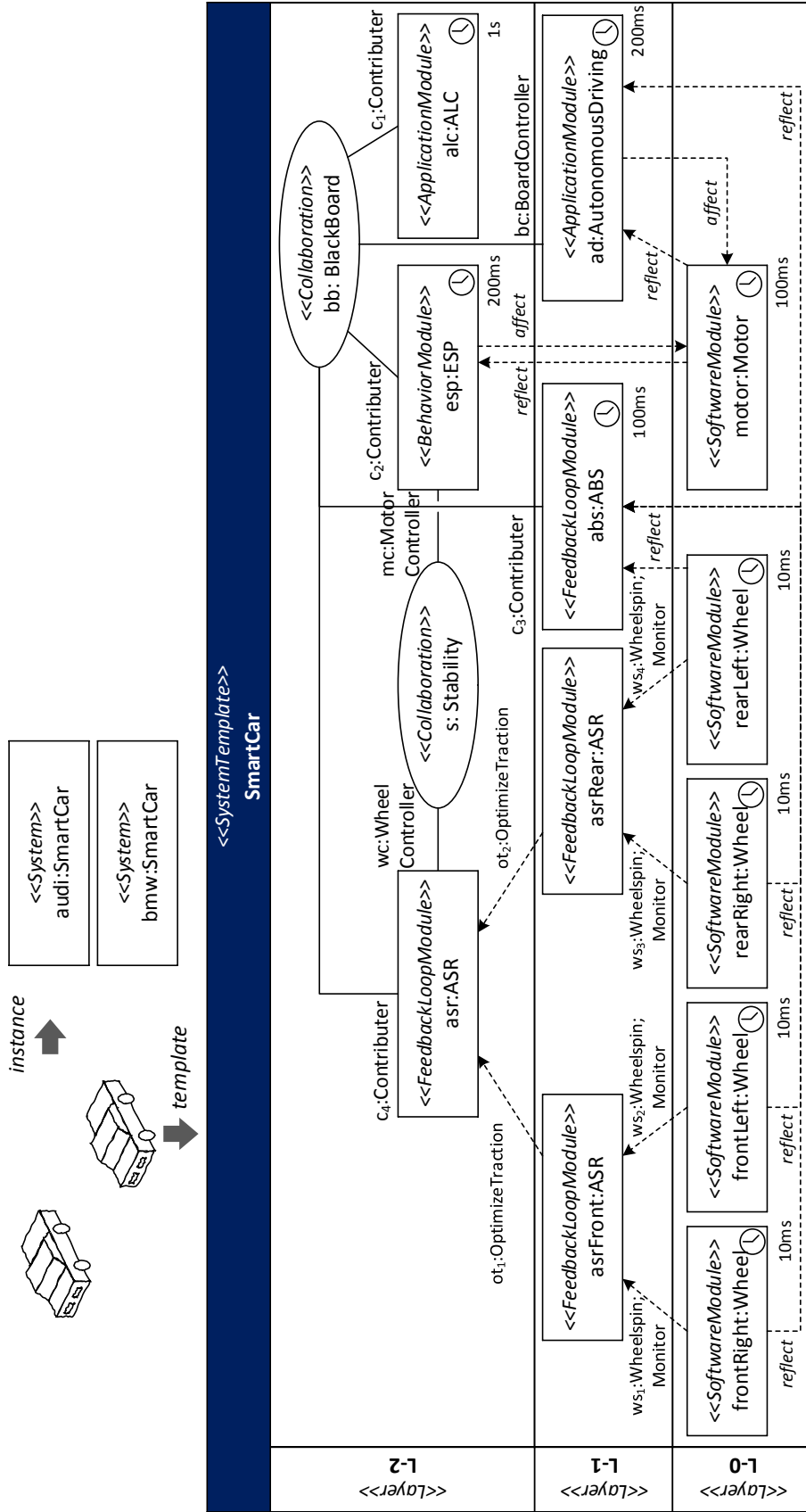


Figure 5.74: Smart car running example modeled in Deurema

parts of the smart car as black box software modules, which are the four wheels and the motor. Each wheel has a time trigger with a period of ten milliseconds, whereas the contained control software of each wheel runs independently from the other wheels. The black box software module representing the motor control has a period of one hundred milliseconds. Therefore, the physical parts of the smart car are controlled very fast as typical for embedded and cyber-physical systems.

At the middle layer in the smart car system, the traction control functionality ASR and ABS are realized by means of feedback loop modules supervising the four wheels. Furthermore, an autonomous driving functionality is encapsulated in an appropriate application component module. With respect to the traction control, there are two feedback loops controlling the front wheels and rear wheels independently from each other. Thereby, the feedback loops are triggered by a *Wheelspin* event, which points to the *Monitor* initial node of the feedback loops. In this example, one trigger event from a wheel is enough to enable the feedback loop on top (or semantic for the module trigger condition). In contrast, the ABS feedback loop module is not triggered by an event. It directly uses the Deurema reflection mechanism to retrieve the desired information from each wheel. Because each module must have at least one trigger in Deurema, the ABS feedback loop has a timing trigger with a period of one hundred milliseconds. Thus, the ABS module can reflect information from the wheels and perform its internal feedback loop behavior not faster than every one hundred milliseconds. With focus on the autonomous driving functionality, the corresponding application module reflects all software modules on the layer L-0 and has an additional time trigger with a period of two hundred milliseconds. Therefore, there are three feedback loop modules in the middle layer of the smart car, which follow the modeling approach of using adaptation activities for the specification of the adaptive internal behavior. Furthermore, there is one application module that uses the component-based development paradigm for its behavior specification.

The third layer of the smart car contains the higher functionalities of the adaptation behavior. On the left, a central ASR feedback loop is triggered by the two traction control loops at the middle layer. Therefore, the four wheels and the three traction control feedback loops, where the corresponding modules are distributed over all three layers, realize a typical hierarchical control pattern. Thereby, the control functionality is clearly separated into distinct parts (front wheels and rear wheels) and controlled by one central instance at the end of the hierarchy. Furthermore, the control loop on the higher layer performs only if necessary, which is indicated by a trigger event from the modules on the lower layer.

Another important aspect of the third layer is the use of the Deurema collaboration concept. The *Stability* collaboration enables an interaction between both main traction control functionalities within the smart car that are the ASR and ESP module. The former is responsible for the wheels, whereas the latter supervises the motor of the smart car. Therefore, the collaboration between both enables a joint interaction, whereas both modules may improve their local control functionality.

The *BlackBoard* collaboration realizes the corresponding design pattern, where a central point of knowledge is iteratively updated by different entities. Thereby, the autonomous driving module is the control entity of the blackboard, because it needs the collected information to optimize the driving of the smart car. Indicated by the role type, all other participants of this collaboration contribute information, which are for example the high level traction control module (ASR), the ABS module, and an adaptive light control module (ALC). The different specifications of both collaborations show the broad applicability of joint interaction in Deurema. Thereby, the *Stability* collaboration enables the interaction between modules on

one and the same level, whereas the BlackBoard collaboration involves multiple participants that are located on different architectural layers in the smart car system template. Both variants are supported by Deurema.

With focus on the reflection mechanism, the normal use case is that a module on a higher layer reflects/affects the module on the layer below as modeled for the combination of the Motor and AutonomousDriving module. However, Deurema does not restrict the reflection to the normal use case as indicated by the reflection of the Motor module at layer L-0 by the ESP module at layer L-2. In general, modeling such a reflection dependency over multiple layers is a design flaw, because it is a violation of the specified layered system architecture. However, in some cases those conscious violations are necessary for a fast reaction in embedded systems, where higher functionalities must directly interact with low level, physical components. Thus, those cases are supported by Deurema, too.

Beside the patterns and the used Deurema concepts in the smart car architecture, there are interesting key aspects in the layered architecture of the smart city system template as depicted in Figure 5.75. At first, the beforehand modeled smart car system template can be reused and integrated into the adaptive behavior of the smart city. In the example, there are two deployed smart car instances audi and bmw at the lowest layer L-0 in the smart city system template. The possibility of deploying (instantiating) systems within a system hierarchically is a core concept in Deurema towards the specification of the overall adaptive SoS behavior. Of course, beside system templates, module templates can be deployed multiple times and thus, reused in the layered architecture specification.

The basic concepts for module triggering and reflection within the smart city follow the same line of argument as for the smart car system and thus are not discussed in detail again. However, there is a combination of both concepts, which first appears in the smart city system template between the LocalTrafficMonitoringSystem software module at the lowest layer and the Self-Healing feedback loop module at the middle layer. At first, the software module on the lower layer triggers the feedback loop module above via the Repair event. Afterwards, the self-healing feedback loop reflects the triggering module to enhance the local knowledge and retrieve the situation that leads to the repair event (e. g., to detect the failure of the module below). This combination has the advantage that the higher feedback loop must not run periodically, which may consume unnecessary computational resources, but rather is triggered in the case of a failure. Furthermore, the feedback loop can directly reflect the information of interest at this point in time, where the failure occurs, which enables an appropriate affecting of the erroneous module below.

Another special modeled combination is the use of the Deurema reflection mechanism between modules on one and the same layer instead of between different layers of the adaptive architecture. The layer L-2 contains such a combination, where the TrafficMonitor behavior module reflects the TrafficOptimization feedback loop module. As shown in the example, the feedback loop module realizes the observer role of the platoon collaboration instance. Therefore it has additional information about the current platoon. Instead of sharing this information with the TrafficMonitor over another collaboration, the behavior module directly retrieves the desired information by reflecting the feedback loop and the contained runtime models. Although, this scenario is supported by Deurema, it is a clear design flaw and should be replaced by an appropriate collaboration. The advantage of a collaboration is that the TrafficOptimization exactly knows and controls, which local knowledge is shared over the corresponding interaction. In the reflection case, from the perspective of the reflected module,

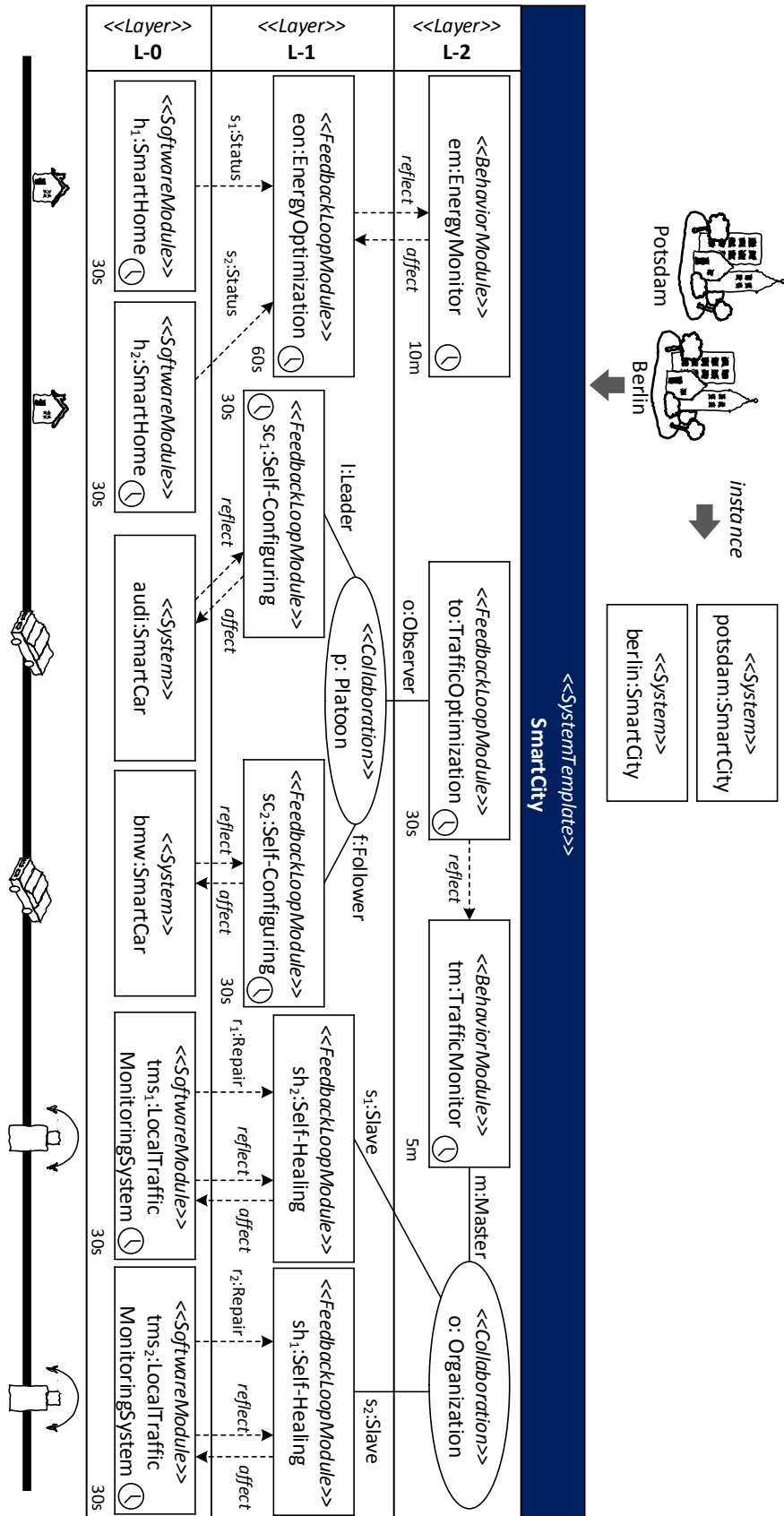


Figure 5.75: Smart city running example modeled in Deurema

it does not even know that information is retrieved by another module, which might cause further security violations by reading sensitive data.

Finally, the last typical design principle that can be observed in both template definitions of the examples in Figure 5.74 and Figure 5.75 pinpoints to the timing of modules. Typically, fast and reactive adaptation behavior is specified at the lower layers in the system architecture to cope with local problems that must be quickly solved. In contrast, higher layers are often used to place long term optimization or strategical orientations of the adaptive behavior, which runs less often. Reasons for longer periods of such strategic modules are the complexity of deriving long term adaptation strategies, which may take a long time of planning, or that the underlying adaptation effects need some time before a benefit of the overall behavior can be monitored.

In summary, the smart car and smart city example show a seamless integration of different module types, which directly corresponds to different development approaches of specifying adaptive behavior, into the layered, adaptive architecture of the system template. Furthermore, system templates can be deployed, which enables hierarchies of systems towards a specification of the overall, emergent adaptive SoS behavior. Thereby, interactions between modules and systems are considered as first class concept in form of collaboration. Finally, the triggering and reflection mechanisms explicitly determine the dependencies between modules and system instances in Deurema.

5.7.2. Design Decisions

The modeling language design decisions are motivated by the characteristics of adaptive SoS and the derived requirements in Chapter 3. First of all, the explicit modeling of the adaptive behavior in a SoS (R-01) is a major aim of this thesis. Following the MDE paradigm, Deurema applies the generic megamodel concept. A megamodel refers to a model that contains other models and relationships between these models [37, 97]. Therefore, Deurema considers models as first class entities, whereas the Deurema megamodel maintains all model elements and the relationships between those. On the architectural level, modules encapsulate the adaptive behavior and can be deployed in the layered architecture of a system. Beside modules, a system may contain other systems, which allows the hierarchical definition of the overall SoS architecture. Thereby, Deurema strictly follows the MDE approach and considers modules and systems as models contained in a megamodel. Furthermore, the relationships between modules and systems, such as triggering or reflection, are explicitly modeled and maintained in the megamodel, too. Thus, the megamodel concept is the basis for reasoning about the specified Deurema models.

On module level, there are two dimensions for the specification of the adaptation behavior that are the concrete adaptation steps and the available knowledge, where the adaptation logic can be applied on. Concerning the adaptation steps, Deurema provides adaptation activities that form a feedback loop following the MAPE blueprint from Kephart et al. [110]. This feedback loop concept clearly targets the explicit modeling of the adaptation logic in self-adaptive systems, which is specified in Deurema Feedback Loop Diagrams (FLD). Furthermore, a SoS consists of several system types (C-1.1) from different domains, such as SAS, embedded systems, or CPS. Beside the feedback loop mechanism, Deurema provides two additional concepts for describing and integrating the adaptation aspects from these different system types (R-20). First, Application Component Diagrams (ACD) follow the component-based development approach as widely adopted in the embedded, robotic and automotive domain. Second, Behavior Rule Diagrams (BRD) offer the powerful concept of

using graph transformation rules for the declarative definition of the adaptation effects by means of trigger-action conditions, which perfectly corresponds to the "everything is a model" design decision above and the graph representation of models. Therefore, the developer can choose an appropriate modeling concept that is encapsulated in a corresponding module template type, which facilitates the integration of different domains by defining the adaptive SoS behavior. On the one hand, each module template clearly separates the different domain concepts on module level. Therefore, the concepts cannot be mixed up by the concrete modeling of the adaptation steps. On the other hand, arbitrary module instances from the three different Deurema modeling concepts can be deployed on the layered system architecture, which allows a reuse of the template definition and a mixture of domain specific concepts at the architectural system level. However, Deurema supports exactly the three concepts of feedback loop, component-based and graph transformation rule modeling for a wide adoption of the modeling language in the domain specific systems and problems within the overall SoS. Additional concepts, which are not captured by these three approaches, must be modeled in Software Module Diagrams (SMD) and thus, integrated into Deurema as black box behavior. If other development paradigms should be integrated into Deurema, such as the agent-based development approach, a new module template type must be defined. Afterwards, such module instances can be integrated and mixed in the system architecture as done for the other template types in Deurema.

Concerning the knowledge specification in module templates, Deurema is designed to use runtime models (R-09) as introduced by [32, 38]. Each module template integrates runtime information in the same way in form of a partial local view that refers to a runtime model. Adaptation activities can access and manipulate runtime model views. Furthermore, each runtime model has a purpose, which must follow the presented categorization. The contained information in the runtime model is domain specific and defined by a metamodel. The purpose and the metamodel of each runtime model contribute to the semantic (R-10) of the runtime model, which enables its manipulation by well-defined Deurema model operations. Thereby, the Deurema megamodel maintains the runtime model and the relationships that define the corresponding views in the module templates.

An adaptive SoS consists of independent systems (C-4), which join or leave the overall SoS at beforehand unknown points in time (C-1). Due to these characteristics, systems are always considered as independent in Deurema. As a consequence, a system cannot directly trigger or influence another system entity. The design of the Deurema modeling language considers two possibilities of defining the inter-system/module dependencies. First, the Deurema reflection mechanism (R-16, R-17) is designed to retrieve information from systems/modules contained in a lower layer and manipulate (affect) the inner structure of those as needed by the current goals of the SoS. Second, Deurema considers system and module collaboration as first class entities (R-07, R-08), whereas the interaction concept is designed for systems/modules that are placed on the same architectural layer. For systems, the Deurema role and view delegation concept transfer the corresponding responsibilities and runtime information to modules contained in the inner system architecture.

As design decision, the modeling of the collaboration related behavior is separated from the modeling of the local adaptation behavior in module templates to foster separation of concerns (R-07) and enable the parallel development of the adaptive SoS. Later, the collaboration specific behavior must be integrated into the local module template definition in form of interactions that hide details of the concrete role protocol behavior. The advantage of this

approach is that there is no break in the abstraction level, whereas Deurema can clearly distinguish between local adaptation activities and collaboration interactions.

Finally, Deurema uses a variable concept for the specification of possible reconfiguration points within the module template (R-15). Thereby, different variable types are defined by the modeling language that corresponds to the specifics of the development paradigm in the supported module templates. Furthermore, a variability runtime model determines the possible configuration space of the adaptive SoS, whereas the concrete configuration is assigned during module instantiation and can be changed during the lifetime of the system. However, the predefined variable types focus on the existing Deurema module templates and can be generically applied for system reconfiguration. On the one hand, if additional variation points are necessary, e. g., by introducing another module template, the existing variable types must be extended accordingly. On the other hand, the Deurema reconfiguration mechanism must not be changed because it operates on the abstract variable concept and not on module template specific extensions.

5.7.3. Coverage of Requirements

As outlined above, the modeling language requirements are derived from the adaptive SoS characteristics in Chapter 3. This section compares the Deurema modeling concepts with the derived model language requirements as follows. The adaptation logic is explicitly determined (R-01) in Deurema Feedback Loop Diagram (FLD), Application Component Diagram (ACD), and Behavior Rule Diagram (BRD). Thereby, the intra-loop coordination (R-02) of adaptation activities within a feedback loop can be modeled. Multiple feedback loops (R-03) are considered by the Deurema module concept. Furthermore, modules are independent (R-05) or can be triggered (R-04), whereas Deurema supports modeling concepts from different domains (R-20). The delegation of tasks (R-06) is supported by the Deurema collaboration concepts, which further separates the interaction behavior (R-08) by means of an abstract role concept (R-07). The knowledge inside the adaptive SoS is captured by runtime models (R-09), whereas a corresponding purpose (R-10) defines the intention of the contained information. Furthermore, Deurema explicitly supports partial knowledge (R-11) realized by the runtime model view concept. Additionally, runtime models can be exchanged (R-12) during system collaboration, whereas Deurema supports different interaction messages (R-13) and synchronization mechanisms (R-14). With focus on the adaptive capabilities, Deurema supports system reconfiguration (R-15) as first class concept. Furthermore, the introduced reflection mechanism facilitates the adaptation (R-16) and meta-adaptation (R-17) of systems as well as modules.

In summary, Deurema covers all of the requirements except of the missing discussion about the coexistence of offline and online adaptation (R-18). With respect to the goals and the focus of this thesis, the coexistence of adaptation activities is not discussed in detail. However, a comprehensive discussion can be found for the predecessor Eurema modeling language in [175], whereas the general concepts are adopted by Deurema. Nevertheless, Deurema covers all major system characteristics, which are reflected by the derived requirements, by offering modeling concepts for the open, dynamic and collaborative nature of the adaptive SoS.

After the introduction of the modeling language concepts in this chapter, the Deurema analysis capabilities on basis of a modeled SoS architecture are investigated in the next chapter.

6. Analysis

This chapter discusses the analysis capabilities of an adaptive SoS modeled with the Deurema approach. As sketched in Figure 6.1, the basic idea is the investigation of a Deurema model by means of static analysis rules during the development of the adaptive SoS. Thereby, the analysis supports the investigation of all Deurema entities as introduced in the last chapter. In the example sketch in Figure 6.1, the smart city system template is analyzed. Furthermore, the system template description comprises all contained subsystems, module instances together with their template description, collaboration instances together with the underlying templates related to the collaboration as for example the structure and choreography specification, and the relationships between systems, modules, and collaborations. Thus, all parts of the adaptive SoS can be individually analyzed during the development by applying static rules on the Deurema models. In general, the Deurema analysis helps the system developer to understand the modeled SoS architecture by pinpointing to different metrics in the corresponding system or module template. This enables the detection of system properties such as *-awareness or self-* characteristics of the adaptation logic. Furthermore, typical design patterns such as feedback loops, which are designed according to the MAPE approach, are retrieved from the modeled templates. In this context, the analysis of the availability of knowledge by means of local views in the module templates gives insides into the knowledge distribution in the system template and thus, of the overall SoS. Finally, looking at collaborations, distributed feedback loops, the knowledge propagation, and the influence between subsystems becomes visible by applying the Deurema analysis.

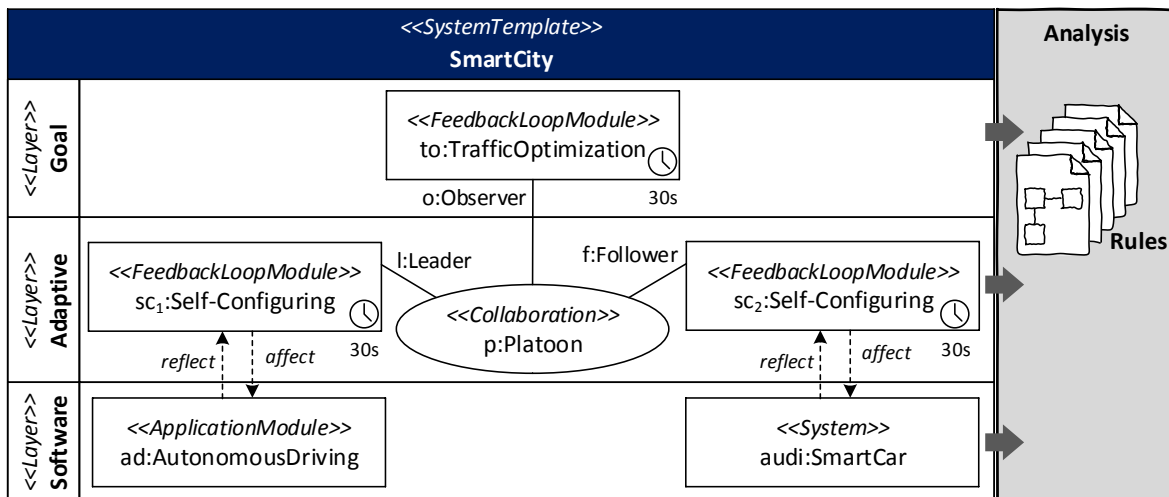


Figure 6.1: Smart city running example: Deurema analysis

After the introduction of the Deurema analysis concept, basic metrics concerning the analysis of the causality, knowledge, and adaptation purpose are discussed in Section 6.1. On basis of the fine-grain analysis rules and the corresponding basic metrics, advanced analysis rules are

introduced in Section 6.2, which enables the reasoning about complex system dependencies. At the highest level of abstraction, architectural patterns and design flaws in the modeled adaptive SoS are discussed in Section 6.3. Finally, this chapter closes with a discussion about the implementation of the analysis concepts and used underlying technology in Section 6.4.

In the following, the Deurema analysis concept is introduced as depicted in the overview in Figure 6.2. In the context of this thesis, an adaptive SoS architecture modeled with Deurema is analyzed by means of declarative analysis rules. Therefore, this thesis focuses on a static analysis technique, where the corresponding analysis rules are evaluated on the specified Deurema models. Furthermore, analysis rules are defined by graph patterns that can be directly executed, which is enabled due to the formal definition of the modeling language concepts by the Deurema metamodel. Each analysis rule defines a key point of interest, as for example a structural design pattern within the layered architecture of the SoS. In general, this key point of interest (analysis rule) is a view on the complete Deurema model describing the analysis aspect. The Deurema execution environment applies the analysis rules, which corresponds to search for a match of the LHS of the defined graph pattern in the Deurema model (cf. introduction of graph pattern matching in the preliminaries in Section 2.2.5). Thus, the defined analysis aspect is searched in the modeled SoS. A positive match of the analysis rule is directly annotated in the Deurema model. Therefore, each match of an analysis rule is called *annotation* for the rest of this thesis. Furthermore, the described analysis aspect belongs to an *annotation type*. As a consequence, annotation types define the overall available analysis aspects, whereas the corresponding annotation (instance) marks the occurrence of the analysis aspect in the Deurema model and thus, directly pinpoints to the occurrence of the key point of interest in the adaptive SoS architecture.

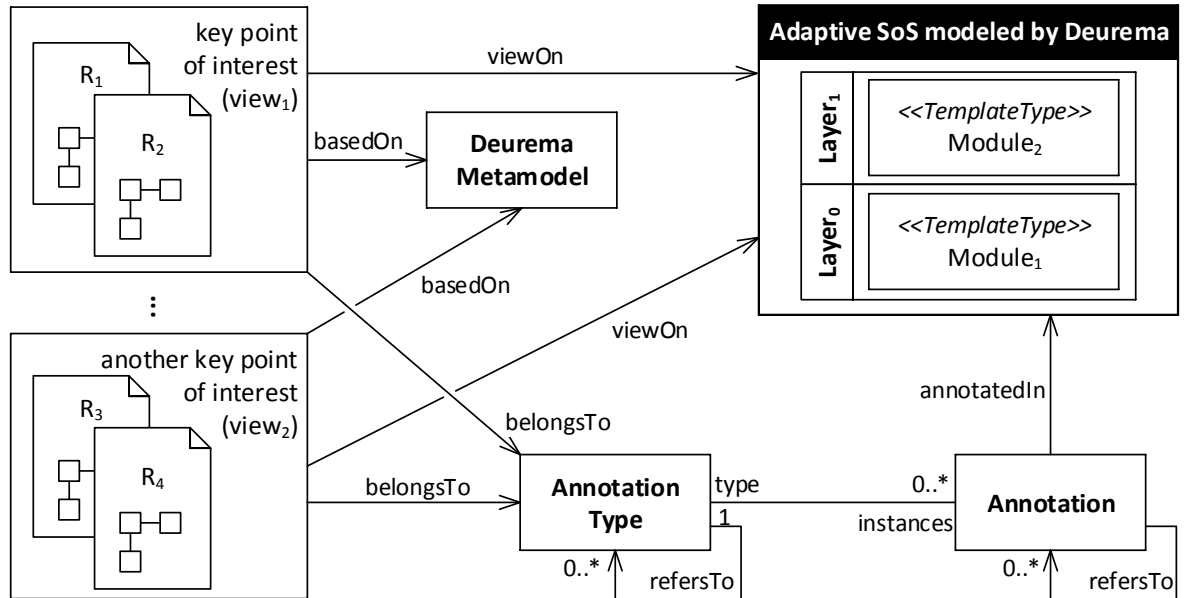


Figure 6.2: Deurema analysis overview

Of course, multiple analysis rules can correspond to one annotation type, if for example one and the same analysis aspect is represented by different variants of a graph pattern. For example, a design pattern can have different variants in the concrete realization and thus, must be represented by different analysis rules, where all rules belong to the same design

pattern (analysis aspect). Finally, low level analysis aspects can be combined to more complex aspects, whereas the latter depends on the occurrence of the former. Therefore, the complex annotation types refer to the low level annotation types as depicted by the `refersTo` reference in the `AnnotationType` class in Figure 6.2. As a consequence, the Deurema analysis directly supports the retrieval of hierarchical assembled analysis rules.

It has to be noted that the following discussed analysis patterns of this section are simplified for explanation purpose. The concrete realization and concrete syntax of the analysis rules is more complex by using the corresponding software tool as discussed in Section 6.4. However, the complete hierarchy of annotation types together with the concrete realized analysis rules can be found in the Appendix C.

6.1. Basic Metrics

Starting with the basic analysis rules, Deurema considers three different aspects. At first, the analysis of causal dependencies is determined, e. g., one feedback loop triggers another one. Second, Deurema provides rules to investigate the knowledge distribution in the adaptive SoS. Third, the adaptation purpose of Deurema elements can be analyzed, for example if a feedback loop realizes a full MAPE cycle or follows other adaptation patterns. In the following, the analysis rules for each of the three basic aspects are discussed. These basic metrics can be used to derive more complex dependencies as well as architectural patterns or design flaws afterwards.

6.1.1. Causality

In the context of this thesis, a Deurema element (e. g., an adaptation activity) can be seen as *cause* for the *effect* of another element, if the former is (partly) responsible for triggering the effect of the latter. Thus, a causal dependency between both Deurema elements arises. Investigating causal dependencies in the adaptive SoS is important for understanding the order of adaptation effects. If for example the emerged adaptation effect contradicts the expected modeled behavior, a causal dependency analysis may pinpoint to the root element, whereas the adaptive behavior deviates from the expected specification. The analysis of the causality at the level of a module template comprises the intra-loop coordination of feedback loop modules, the causal order of runnables in application component modules, as well as causal dependencies between graph transformation rules in behavior modules. Furthermore, on system level, causal dependencies between modules and collaborations are investigated by looking at the Deurema inter-loop coordination mechanisms. In the following, causal dependency analysis rules are introduced starting at fine-grain, simple rules in single module template types and increase the abstraction level towards a causal dependency analysis between Deurema modules and systems.

Feedback Loop Diagram

Deurema feedback loop diagrams define the intra-loop coordination between adaptation operations by means of the control flow as shown in the metamodel in Figure 5.20. On basis of the control flow, two kinds of causal dependencies can be defined that are a direct causal dependency between two successive operations and an indirect causal dependency looking at the transitive closure of direct causal dependencies. Additionally to the transitive closure, paths of adaptation activities, which corresponds to the causal execution sequence of the feedback loop, can be denoted.

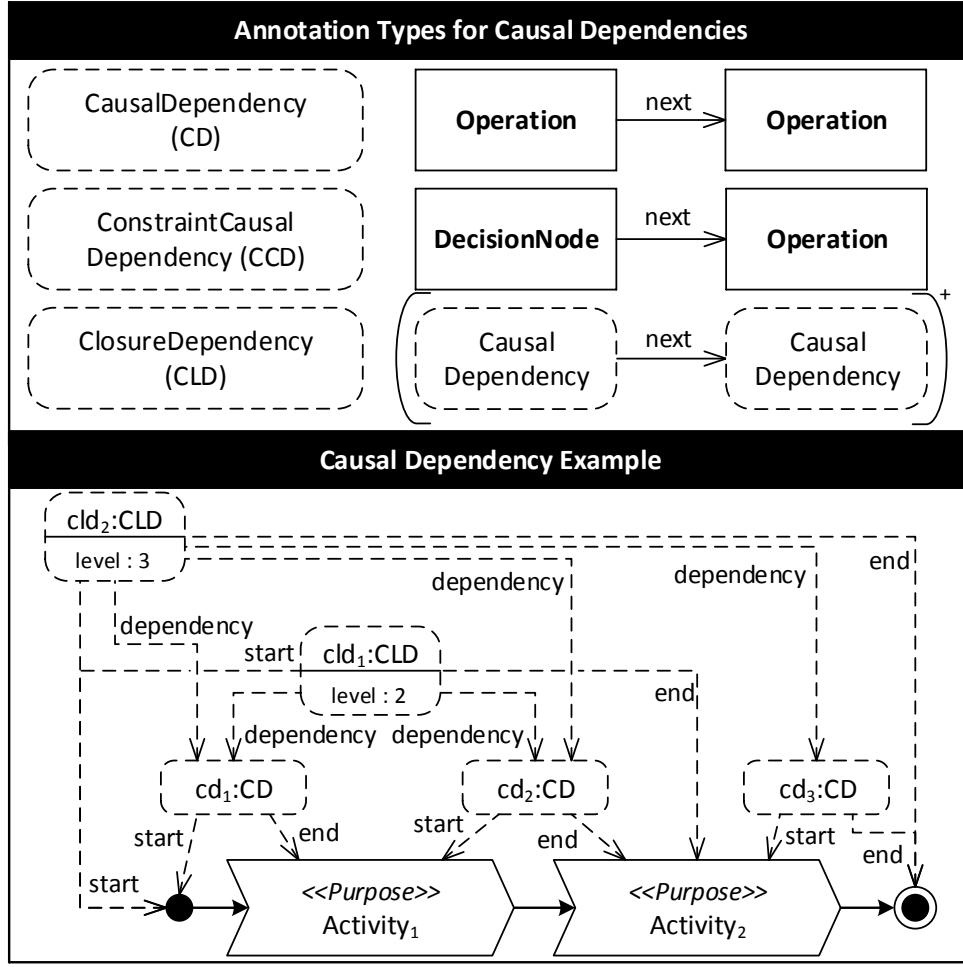


Figure 6.3: Causal dependencies in a FLD

At the top in Figure 6.3, the pattern for the simple causal dependency between two operations in a feedback loop is shown, where the first operation refers to the subsequent operation via the next reference defined in the Deurema metamodel. As discussed above, each time the pattern is found in the Deurema model, the inferred knowledge is marked with a CausalDependency annotation. For the rest of this thesis, the concrete syntax for an annotation type is a rounded rectangle with dashed border as shown in the upper left in Figure 6.3. Furthermore, annotation instances follow the UML object annotation, which refers to the corresponding annotation type. The result of an application of the simple causal dependency pattern on a Deurema example model is shown at the bottom in Figure 6.3. There are three inferred direct causal dependencies cd_1 , cd_2 , and cd_3 in the example.

A specialized type of a direct causal dependency between two operations is the Constraint-CausalDependency, where the first operation is a decision node followed by an arbitrary operation of the feedback loop. Decision nodes branch the control flow in several paths depending on the specified branch condition, which leads to multiple possible adaptation paths inside one feedback loop.

The direct causal dependencies are the basis to compute the transitive closure of the overall causality between feedback loop operations. Thereby, the size of the closure depends on the

amount of operations that follow a given starting point operation. Furthermore, a causal ClosureDependency is defined by at least two subsequent simple causal dependencies. The defined closure dependency pattern in Figure 6.3 is an example for a complex annotation type, because it aggregates pairs of the beforehand defined causal dependency annotation type. Therefore, the occurrence of a closure dependency depends on the existence of normal causal dependencies in the feedback loop. Thus, the defined closure dependency pattern leads to the retrieval of all partial closures for all operations in the feedback loop. The example in the figure shows all causal closure dependencies for the initial node of the modeled feedback loop. For the sake of visibility, closure dependencies for all other operations in the feedback loop example are omitted, but are of course retrieved during the analysis by the defined pattern. However, the initial node is the starting point for two closure dependencies cld_1 and cld_2 . The size of the causal closure is annotated by the level attribute in the inferred dependency annotation. Consequently, cld_1 denotes the partial closure beginning from the starting node to the $Activity_2$ operation in the feedback loop. It references the corresponding direct causal dependency annotations, where the amount of direct causal dependencies corresponds to the size of the causal closure (value of the level attribute in the annotation). Straight forward, the cld_2 annotation in the example denotes the complete transitive causal closure beginning at the initial node and ending at the final node of the feedback loop, which comprises three direct causal dependencies.

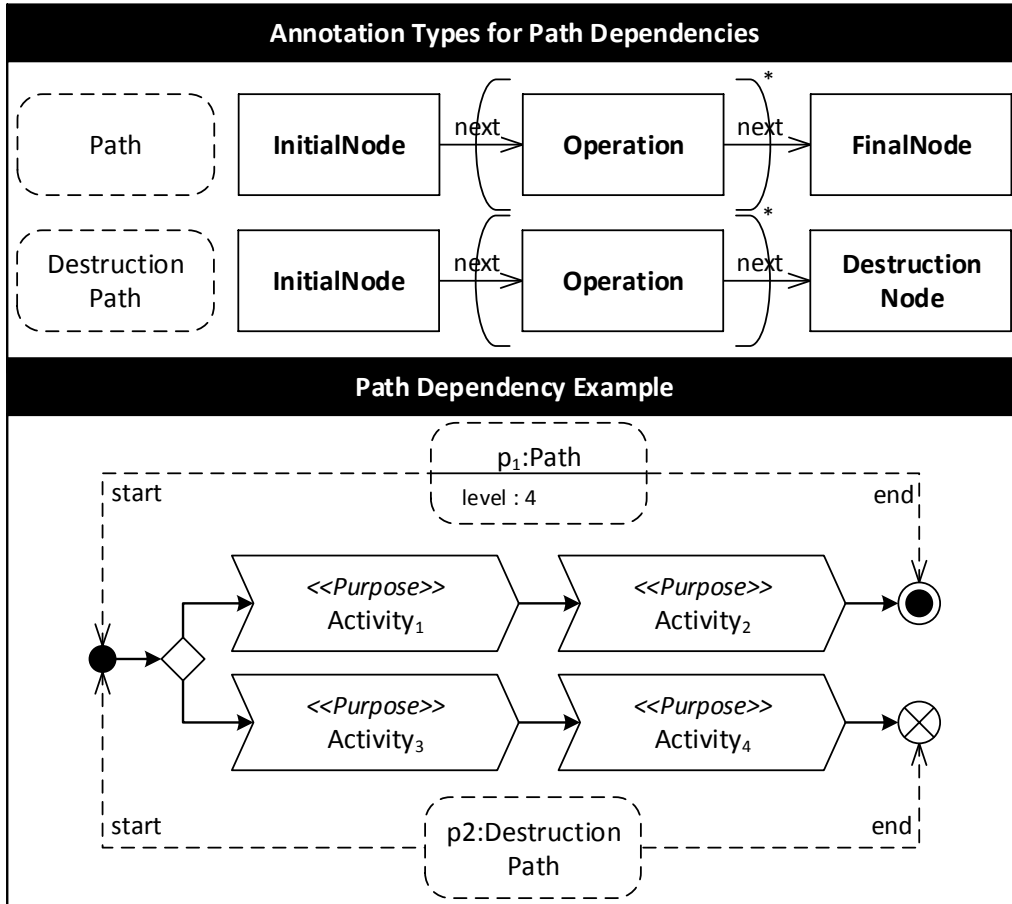


Figure 6.4: Causal dependency paths in FLD

Where the direct and closure causal dependencies are basic, but imprecise metrics, a path annotation denotes one possible trace of executing a feedback loop beginning at an initial node and ending in a final node. Therefore, each path annotation refers to one modeled adaptation behavior of the modeled feedback loop, which can be used to detect unintended adaptation effects (paths). The simplified pattern of a Path dependency together with an example are shown in Figure 6.4. The pattern defines that each path must start with an initial node, can have arbitrary number of subsequently executed operations, and ends with a final node.

A special path is the *DestructionPath*, where the end operation is a destruction node. An executed feedback loop, which reaches a destruction node, is destroyed afterwards and thus, can be used to model one-shot adaptation behavior as comprehensively discussed in Section 5.3.2. However, due to the possibility of modeling different adaptation branches within one feedback loop using decision nodes, the retrieving of possible paths as well as one-shot adaptation behavior helps identifying and understanding the overall adaptation capabilities of the FLD. The example in Figure 6.4 comprises two paths. The first path refers to the initial node, the subsequently executed activities *Activity*₁ and *Activity*₂ as well as a final node, which indicates the end of the feedback loop. The second path denotes to an occurrence of the destruction path pattern, which comprises the activities *Activity*₃ and *Activity*₄ as well as a corresponding destruction node at the end of the feedback loop.

Application Component Diagram

Determine the causal dependencies within an ACD is trivial. Because runnables are the basic entities of adaptation behavior in an ACD and they are mapped to executable tasks, the order of assigned runnables within the boundary of a task dictates the causal execution order of the ACD (cf. Section 5.3.4). Thereby, finding direct and closure causal dependencies follows the same line of argument as explained above for FLD. Furthermore, because of the missing possibility of branching the behavior, a path dependency in ACD is much more trivial as in feedback loops and can be denoted between the first and last runnable in one and the same task.

Behavior Rule Diagram

In general, there is no explicit concept in Deurema to describe a control flow between graph transformation rules in BRD. Nevertheless, a static analysis highlights causal dependencies between those declarative rules by investigating the LHS and RHS. As depicted in Figure 6.5, a direct causal dependency between two rules R_1 and R_2 can be determined, if the RHS of R_1 is equal to the LHS of R_2 . Thus, whenever the adaptation effect of the first rule is applied, the second rule will subsequently find a match according to its defined graph pattern. There is another causal influencing possibility of the two rules R_1 and R_2 , if the application of the RHS of R_1 creates/deletes a node/edge, where the type of that node/edge is part of the LHS of R_2 . Thus the adaptation effect of R_1 is suspicious to influence the match of R_2 . Whether the application of the RHS of R_1 has an effect on the match of R_2 or not, can only be determined at runtime of the adaptive SoS, because it depends on the concrete found matches of both rules. Therefore, the real runtime effects of this dependency type are uncertain at development time and are named *WeakCausalDependency*. Following the same line of argument as for FLD and ACD, the transitive closure can be analyzed for both, weak and normal direct causal dependencies, between graph transformation rules inside the BRD as sketched at the bottom in Figure 6.5. Because there is no explicit starting and end point in BRD, an explicit path between rules cannot be determined.

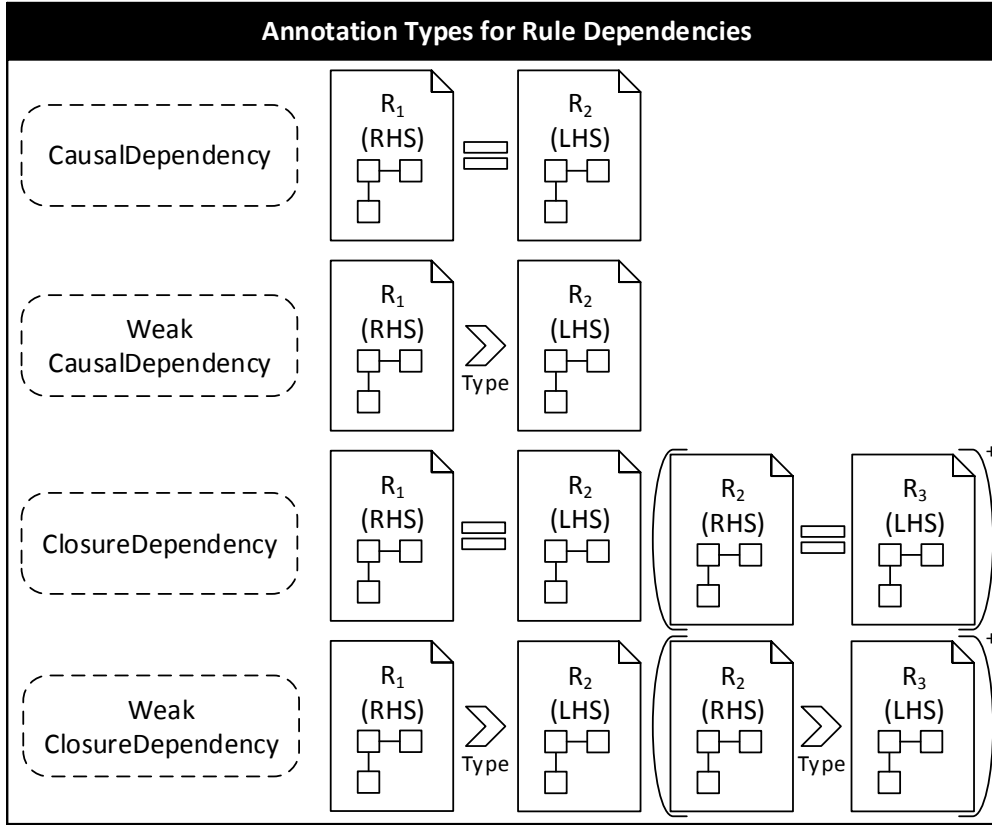


Figure 6.5: Causal dependencies in BRD

Module Triggering

Leaving the context of a single module template and investigating the triggering between modules leads to the (Layered)TriggerDependency as shown in Figure 6.6. If a module triggers another module, the former module is the cause for the adaptation effect of the latter module. Thus, from a causality perspective, all adaptation activities of the triggering module will happen before the triggered module is executed. Conceptually, the analysis rules distinguish two different trigger dependencies. On the one hand, the triggering between two modules can happen on the same layer, which is determined in the TriggerDependency annotation type. On the other hand, the LayeredTriggerDependency considers modules, where the triggering crosses the boundaries of one layer within the system template definition as shown for the $Module_1$ and $Module_2$ at the bottom in Figure 6.6. Of course, trigger dependencies can be transitively analyzed to investigate the causal closure of module triggering.

Collaboration Role Trigger

Finally, the interplay of modules and collaborations is determined concerning the causality analysis. As explained in Section 5.5.6 and shown in the Deurema metamodel in Figure 5.58, modules are the Deurema model elements that realize collaborations by playing the specified roles defined in the collaboration deployment. Furthermore, a role trigger defines the possible execution of the collaboration activities that can be *before*, *after*, or *within (internal)* the execution of the corresponding player module. The first two role triggers define that the collaboration activities do not interfere with local adaptation behavior of the module. Thus,

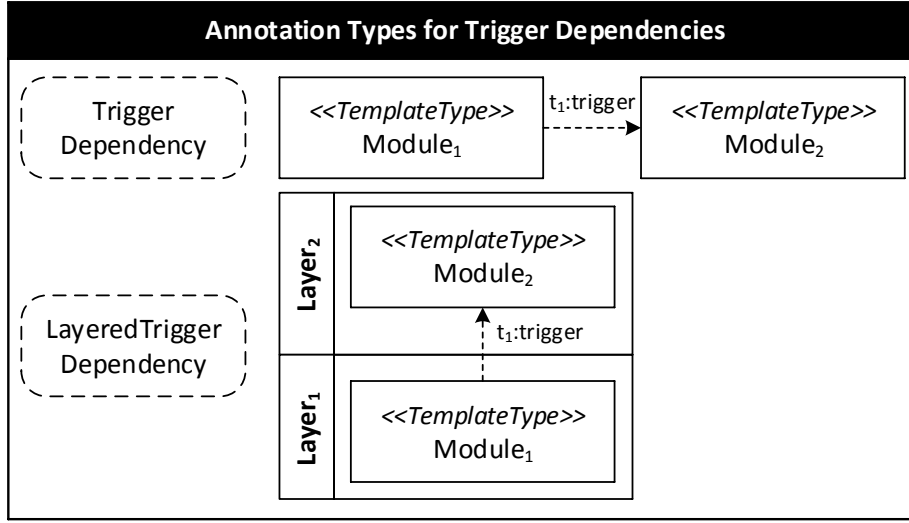


Figure 6.6: Module trigger dependency

the collaboration interactions are directly executed before/after the adaptive behavior of the module takes effect. In contrast, the internal role trigger specifies that the collaboration related behavior is integrated into the local adaptation behavior of the module as comprehensively discussed in Section 5.5.5.

Due to the possible combinations of role trigger between modules, there can be several causal collaboration trigger dependencies derived as summarized in Figure 6.7. The example in the figure consists of one collaboration together with two modules realizing a corresponding role from the collaboration. Although the example considers only two modules, the following dependencies can be easily extended for more modules following the same line of argument. In combination (1), both collaboration role trigger denote the execution of the collaboration after module execution, which is captured in the *deferred collaboration trigger dependency*. According to the example, both modules M_1 and M_2 are independent and can run in parallel. After both modules finish their local activities, the collaboration effects can happen afterwards. In contrast, for combination (2) the collaboration happens *in advance* and the local effects of the modules successively happen, if both role trigger are defined as *Before*. Straight forwards, for combination (3), if one role trigger specifies the execution of the collaboration before and the other role trigger after the local module execution, there is a causal sequence consisting of the first module M_1 , all collaboration activities afterwards and finally, the effects of module M_2 . Thus, the collaboration behavior happens *in between* the local adaptation activities.

The next three dependency combinations (4)–(6) consider the triggering of collaborations with an internal role trigger. Therefore, the collaboration behavior is woven into the local module behavior, which is denoted with the term $M_2 + C_{M_2}$ (for module M_2) in Figure 6.7. The collaboration trigger dependency for combination (4) is called *deferred interleaving*, if the collaboration specific part of one module (in the example M_1) is executed after the module itself, indicated by the term $M_1 \rightarrow C_{M_1}$. The other module, together with its collaborating activities, is independent and thus, in parallel to $M_1 \rightarrow C_{M_1}$. Furthermore, the dependency combination (5) differs in the sense that the role trigger for M_1 is defined as *Before*. Therefore, the collaboration specific behavior is executed before the module M_1 ($C_{M_1} \rightarrow M_1$). Finally, the weakest trigger dependency combination (6) is called *interleaving*, because the collaboration

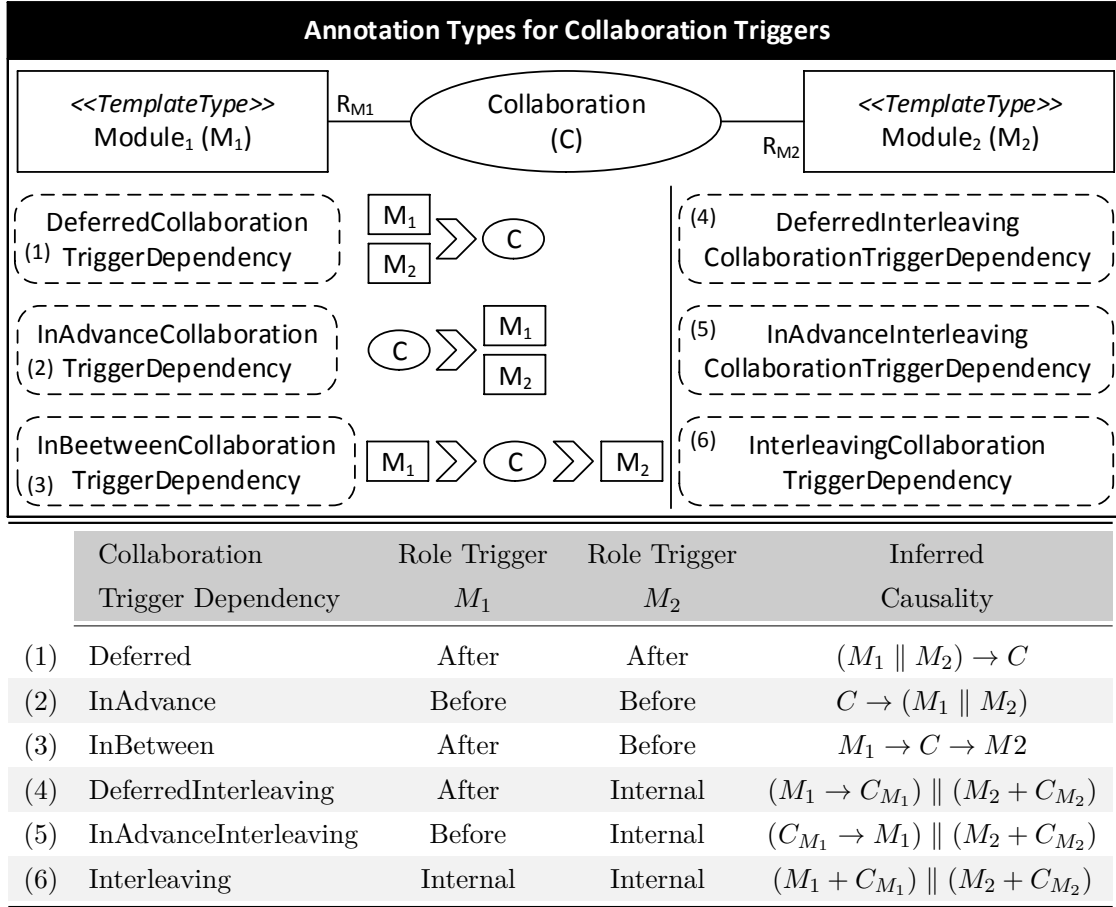


Figure 6.7: Collaboration trigger dependencies

$(M_1 \parallel M_2)$: M_1 is causally in parallel with M_2 ; $M_1 \rightarrow M_2$: M_1 is causally before M_2 ;
 C_{M_1} : The collaboration role part realized by M_1 , whereas $C = (C_{M_1} + C_{M_2})$.

behavior is woven into both modules and thus, the internals of the dependency are not known, without looking into the specific collaboration mapping in the corresponding module template.

Looking at the collaboration interaction specification, each interaction can be analyzed with respect to the causality as explained for feedback loops. Conceptually, interactions are feedback loop templates (cf. metamodel in Figure 5.47), which extends the available pool of feedback loop operations by the Deurema message concept as shown in the metamodel in Figure 5.49. Therefore, the considerations about the causal dependencies and their transitive closure are applicable for interactions as explained above for feedback loop templates.

Furthermore, the communication between roles, which comprises the exchange of a synchronization message, a model message, as well as the invocation of a service, imply a causal order between the sender and the receiver of the communication within a collaboration interaction. This causality of exchanging messages and service invocation can be analyzed and leads to a direction of the communication flow as shown in Figure 6.8. An interaction has an unidirectional communication, if a dedicated role sends one or more messages to other roles without receiving a message. This kind of communication is considered as *unidirectional interaction message dependency* as shown at the top in Figure 6.8. Unidirectional message

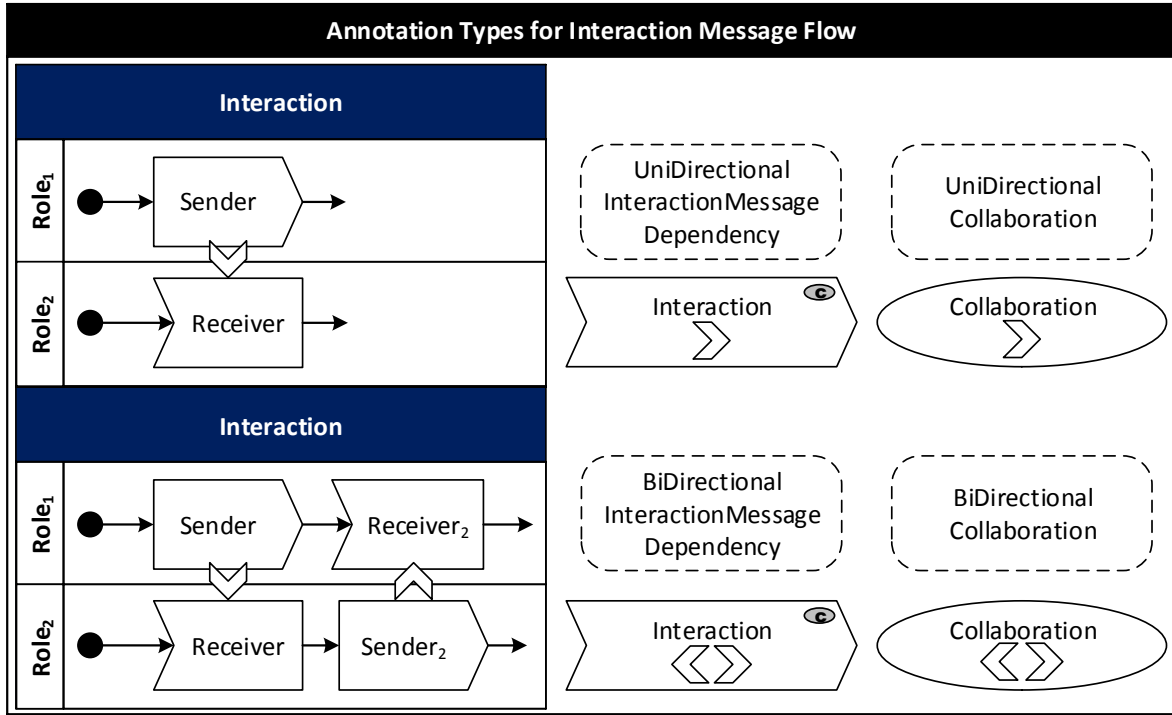


Figure 6.8: Interaction message dependency

communication is often used to trigger or inform other participants of the collaboration as done in the HeartBeat interaction of the platoon example, which is explained in Section 5.5.3.

Additionally, if one and the same role in all interactions of the corresponding collaboration uses only unidirectional message communications, the whole collaboration is considered as *unidirectional*. An unidirectional collaboration is a very strong dependency because it implies that only one dedicated role distributes information to all other participants in the collaboration. Thus, the role realizing module of the collaboration can be considered as source of knowledge in the adaptive SoS.

The more common case is a bidirectional communication within one interaction, which is shown at the bottom in Figure 6.8. The interaction is considered as bidirectional, if there is an unidirectional communication and at least one additional message flow, where another role is the sender. Straight forwards, if at least one interaction in the collaboration specifies a bidirectional message exchange, the whole collaboration is considered as bidirectional. An example of a bidirectional interaction is the ShareEnvironment interaction discussed in Section 5.5.3.

Note, the absence of any message exchange within an interaction or role, which means that a certain role exchanges any information to another role, is a typical anti-pattern that pinpoints to a very likely design error of the collaboration. Furthermore, the described message exchange dependencies combined with the causal dependencies of feedback loop operations lead to an extended causal closure, which leaves the border of a module and spreads over the collaboration to other participating modules. However, this extended causal closure follows the same argumentation as described above, but has to consider all kinds of module templates, because each collaborating module may have another template type.

6.1.2. Knowledge

Focusing on the occurrence of runtime models as local available knowledge in module templates, the Deurema analysis retrieves the corresponding runtime model purposes and reasons about basic characteristics of the module template that contains this knowledge. The analysis of the availability of different runtime model types enables insights into the knowledge distribution throughout the adaptive SoS. As first step, the occurrence of a specific runtime model purpose within the module template, which implies that a corresponding module instance at least knows about the availability of the corresponding information, is enough to derive the hereafter called **-representative* characteristics. An overview about the **-representative* module characteristics is given in Figure 6.9. Thereby, the available runtime model purposes are defined in the Deurema metamodel in Figure 5.14, which corresponds to the presented runtime model categorization of this thesis (cf. Section 5.2). For example, if a module template contains at least one runtime model with the purpose `SystemModel`, it is denoted as *self-representative*, which implies a possible access to that runtime information. All other **-representative* annotation types are enumerated in Figure 6.9. Of course, each instance of the module template, which is a Deurema module, follows the template description and therefore, will have the same characteristics.

Annotation Types for Runtime Model Purpose (*-Representative)				
Reflective	Self Representative	<<SystemModel>> RuntimeModel		
	Context Representative	<<ContextModel>> RuntimeModel		
Changeable	Requirements Representative	<<EvaluationModel>> RuntimeModel	<<RequirementModel>> RuntimeModel	<<AssumptionModel>> RuntimeModel
	Change Representative	<<ChangeModel>> RuntimeModel	<<ModificationModel>> RuntimeModel	<<VariabilityModel>> RuntimeModel
Causal Connected	Sensor Representative	<<MonitoringModel>> RuntimeModel		
	Effector Representative	<<ExecutionModel>> RuntimeModel		

Figure 6.9: Analysis of the knowledge purpose

Beside the fine-grain **-representative* characteristics, module templates are denoted as *reflective*, *changeable*, and/or *causal connected*, if combinations of runtime model purposes in the template description occur. For example, the runtime model purposes `SystemModel` and `ContextModel` belong to the higher category of so-called reflection models. Therefore, if runtime models with these both purposes occur in the module template, it gets the *reflective* characteristic as shown on the left in Figure 6.9. The same lines of arguments hold for adaptation and causal connection runtime models. Thus, appropriate analysis rules are available for each characteristic with the focus of a module template respectively module. Because Deurema uniformly integrates runtime models in the same way for all supported

template types (e.g., FLD, ACD), the analysis works for all Deurema templates as well as interactions within collaborations.

With respect to the runtime model purpose, it is possible to reason about system characteristics by transferring the found module template properties to the system, which contains the module instance. As further analysis, it is possible to identify parts of the system, for example by looking at each architectural layer, which focuses on specific runtime model information. For example, parts of the system may focus on context information, whereas other parts keep track on the given requirements. Afterwards, the interplay between those modules within one system can be interesting, which shows the exchange of this information and thus, gives insides to the data flow of the system related to the usage of runtime model information.

Beside the purpose of a runtime model, there are analysis rules that detect reflected information with respect to the Deurema reflection mechanism as introduced in Section 5.6. In Deurema, reflected information becomes locally available in a runtime model in the corresponding module template, where the defined adaptation logic can access and manipulate the information. Due to the causal connection, changes in the runtime model are synchronized with the underlying reflected system or module.

Figure 6.10 depicts the determined knowledge dependencies with respect to the Deurema reflection mechanism. Thereby, the analysis considers the direction and distinguishes between the *reflection* and *affection* of modules respectively systems. Therefore, a *reflective knowledge dependency* between two modules occurs if one module reflects another module as shown in the example between *Module₁* and *Module₂* in the figure. Due to the Deurema reflection concept, there must always be a corresponding runtime model in *Module₂*, which contains the reflected information. Straight forward, an *affecting knowledge dependency* refers to the affect concept in Deurema. Of course, beside modules, systems can be reflected/affected.

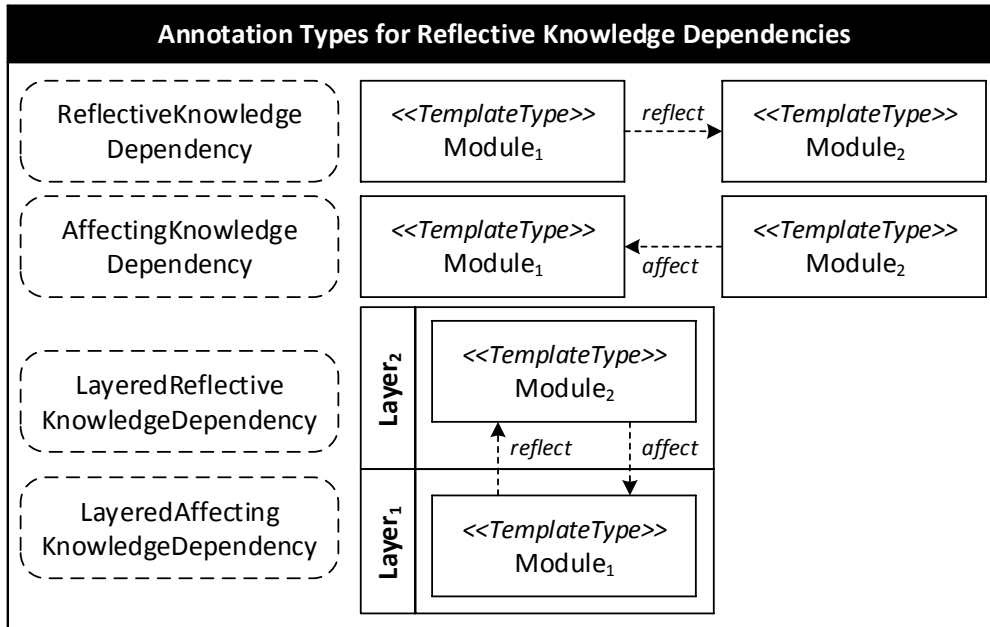


Figure 6.10: Analysis of reflective knowledge dependencies

An additional constraint is the usage of the reflection mechanism between modules and systems in different layers, which is determined by appropriate annotation types in Figure 6.10. This kind of using the reflection mechanism is preferred, but not required, in Deurema. Thus, both kinds are retrieved by corresponding analysis rules.

6.1.3. Adaptation Purpose

Beside the causality and the occurrence of runtime model purposes, investigating the purpose of the adaptation activities is the third basic information that can be retrieved from the Deurema module templates. For feedback loop activities, the Deurema metamodel supports the Monitor, Analyze, Plan, and Execute activity type as discussed in Section 5.3.2. Therefore, the activities and their types can contribute to different variants of a complete MAPE feedback loop, where meaningful examples are depicted in Figure 6.11.

The first example corresponds to a full MAPE cycle, which directly follows the proposed reference architecture from [110]. The second example shows a degenerated feedback loop that focuses on the retrieval of information, which is named *Collector*. A collecting feedback loop is characterized by a monitoring activity and an optional execute activity, where the analysis and planning step is omitted. Although the focus of a collector is the sensing of information, an execution step is reasonable to adjust the underlying hardware, e. g., configuring of sensors, for the next information retrieval round.

Straight forward, the feedback loop is called *Analyzer*, if the focus is on the analysis of sensed data. Such analyzing feedback loops are often used, if the analysis is very complex or computational intensive. In such cases, other loops, e. g., collectors, may contribute knowledge from different sources, which are aggregated and analyzed in one complex step. The optional monitoring and executing step of the analyzer are important for aggregating the information from other loops, e. g., over collaborations, and thus can be used to prepare the available data for analysis. Furthermore, analyzer can be used to monitor key points of interest during the lifetime of the adaptive SoS and report violations to the current goals. For example, watchdogs are a typical implementation of analyzer feedback loops.

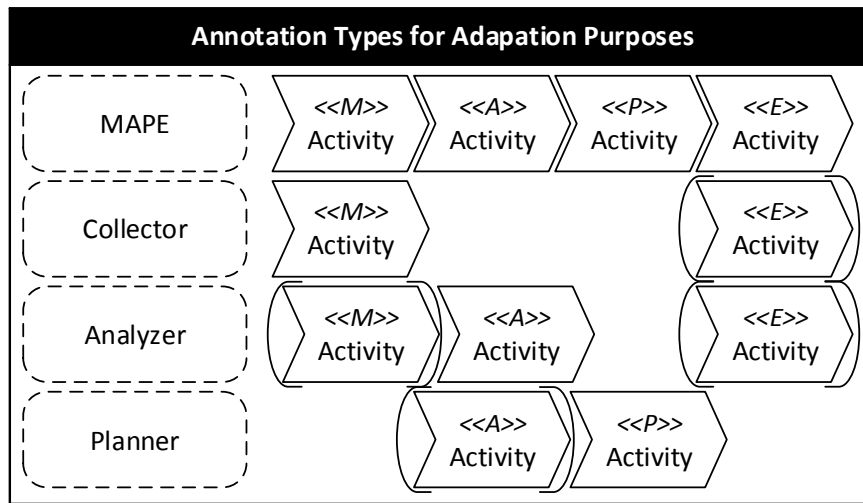


Figure 6.11: Adaptation purpose in FLD

Furthermore, a pure *Planner* feedback loop can be used for deriving long term adaptation plans, which are sometimes combined with analysis activities. Planner loops are useful, if they run independently from the short term adaptation loops to decouple necessary fast reaction of the SoS from strategical derivations of the behavior towards long term goals. They usually do not have monitoring or executing activities to prevent interleaving of other, more reactive, feedback loops. However, if monitoring and executing capabilities are necessary, it is covered by the full MAPE feedback loop as discussed above. Additionally, the occurrence of a single executing activity, without the retrieval, analysis, or planning of information, may indicate a design error of the feedback loop.

Concerning the adaptation purpose, the same line of argument can be transferred to ACD by looking at the component types inside the module template. As discussed in Section 5.3.4, Deurema supports the component types Sensor, Actuator, and software component (SWC). The occurrence of all three types corresponds to the *sense-compute-act* paradigm of developing embedded control loops. Therefore, this case is considered by the *sensor-software-actuator* (SWA) annotation in Figure 6.12.

Furthermore, the Deurema analysis rules consider two more cases, which split the complete paradigm above in the *compute* and *sense-effect* part. Consequently, software intensive parts can be independently detected from components that realize the physical interaction with the underlying system. Because the *compute* part focuses on the detection of software components, it can be compared with the *analyzer* respectively *planner* combination as discussed above for feedback loops. Furthermore, the *sense-effect* part focuses on the sensors and effectors within an ACD, which is similar to the *collector* in FLD.

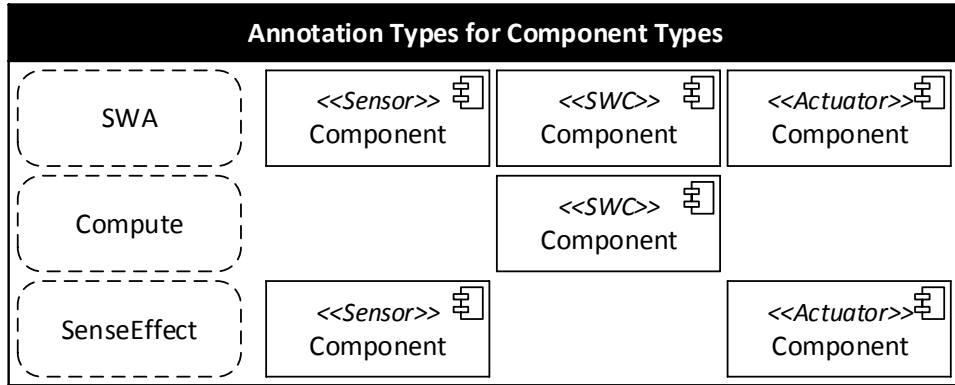


Figure 6.12: Adaptation purpose in BRD

The rule concept in BRD is a declarative description of the adaptation logic, which does not support specifying the adaptation purpose as first class entity. The declarative character of rules can be used to define trigger-action conditions, which detect the occurrence of specific situations in the adaptive SoS and describe an appropriate adaptation step. Consequently, each rule has a specific purpose related to the defined situation, which can potentially occur at any point in time in the SoS. Thus, the specification of a predefined adaptation purpose, which is related to a concrete adaptation step as for example in a MAPE feedback loop, is not suitable for rules in a BRD. However, if the rule adaptation purpose is important, characteristic access patterns can be determined for retrieving the intended adaptation purpose. For example, if a rule reads and writes a monitoring runtime model, it can be denoted as *monitoring rule*. Distinguishing analysis and planning rules is much more complex, because

the access to different runtime model types is not restricted. As a consequence, the rule purpose is often domain or problem specific and thus, must be investigated with respect to this domain or problem.

In summary, investigating the causal dependencies, occurrence of knowledge, and adaptation purpose in specified Deurema models are independent, basic metrics towards an understanding of the modeled adaptation logic of the SoS. These three basic metrics can be combined to determine more complex adaptation structures, the impact of knowledge as well as the distribution of runtime model information through the system as discussed in the following.

6.2. Complex Metrics

This section introduce meaningful examples for complex analysis metrics for each possible combination of causality, knowledge, and adaptation purpose. Combining the three basic metrics to more powerful analysis rules enables the investigation of complex dependencies between systems, modules, and collaborations within the adaptive SoS.

6.2.1. Combining Causality and Adaptation Purpose

Whereas the analysis of the occurrence of specific adaptation activities does not determine the causal order of their execution, a combined analysis can detect typical forms of feedback loops in FLD as well as meaningful task definitions in ACD.

At first, following the reference architecture from Kephart et al. [110], a typical pattern is a MAPE feedback loop, where each adaptation activity occurs once and is subsequently executed. As a next step, this pattern can be extended by allowing an arbitrary occurrence of each adaptation purpose, where the execution order of the adaptation purposes is still protected. For example, the feedback loop may have two monitoring steps instead of one, but after the last monitoring activity follows the analysis step as expected by the MAPE pattern. In contrast, each violation of the standard pattern, e. g., AMEP can be reported, which gives hints to the developer of possible design messes.

Furthermore, domain or problem specific patterns for the typical design of the feedback loop can be provided and investigated. For example, if an analyze activity decides that no adaptation steps are necessary for the current situation, the execution of the feedback loop can be stopped without executing the plan and execute activity as shown in Figure 6.13. Thus, the analysis activity in this design pattern decides on the early cancellation of the feedback loop. On the one hand, the planning and execution step are potentially omitted, which reduces the computational overhead of the adaptation logic in the optimal case. On the other hand, long term planning activities are skipped as long as the analysis does not detect violations of the current goals. Therefore, the use of this design pattern introduces advantages as well as known issues, which can be collected in a pattern catalog. The applicability of this pattern depends on the specific problem that has to be solved. However, those patterns support the developer by modeling the adaptive SoS capabilities and pinpoint to possible violations.

In the context of an ACD, the detection of causality-adaptation-purpose patterns is more complex. Due to the architectural specification of the adaptation logic following the component-based approach, the detection of the design patterns cannot be specified by looking at the component structure alone. In an ACD, the runnable is the behavioral entity, where the execution order depends on the corresponding task mapping. As a consequence, the analysis of

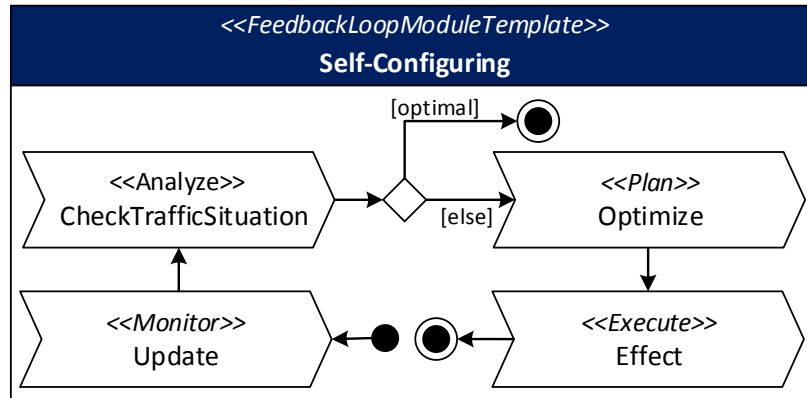


Figure 6.13: Exit on analyze pattern

design patterns considers, if the mapping of runnables corresponds to the *sense-compute-effect* paradigm as discussed above. For example, if runnables from different sensor components are mapped to a single task, where no other runnables from other component types are mapped, a *sensing task* pattern is detected. This argumentation can be transferred to computational tasks (only runnables from SWC components) as well as effecting tasks (only runnables from Actuator components). Furthermore, each of the sensing/computing/effecting related behavior can be split on several tasks, which is analyzed and detected, too. Another pattern is the detection of component-centric behavior, where runnables from one component are mapped to a single task. In this case, the execution of the behavior goes hand in hand with the structured encapsulation defined by the component architecture.

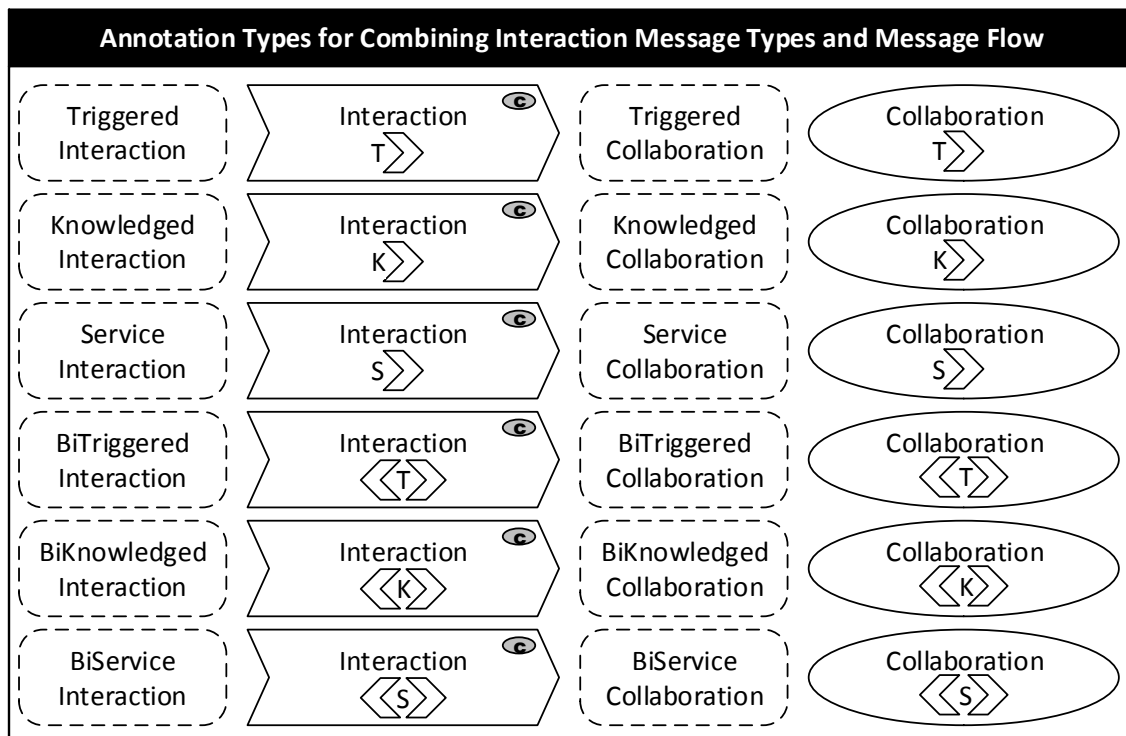


Figure 6.14: Combining interaction message types and message flow

Focusing on collaborations, the communication direction can be combined with the analysis of the message type. The Deurema metamodel defines three different message types (cf. Section 5.5.3) that are normal messages for synchronization and triggering, model messages, and services. Furthermore, the analysis rules investigate the communication flow in defined system interactions as discussed above and distinguish between unidirectional and bidirectional communication. The combinations of these two aspects together with the corresponding annotation types are depicted in Figure 6.14. For example, an unidirectional communication between two roles using a model message is marked in the Deurema model with a `KnownInteraction` annotation. Furthermore, if all interactions within the complete collaboration specification are restricted to unidirectional communication using only model messages, the analysis denotes this aspect with a corresponding `KnownCollaboration` annotation. The same line of argument holds for trigger messages and services.

6.2.2. Combining Knowledge and Adaptation Purpose

Combining the available runtime model purposes in a module template together with the adaptation purposes leads to typical access patterns, which are subsumed in Table 6.1. The Deurema analysis distinguishes a read and modifying access to a runtime model, whereas the latter subsumes writing, creating, deleting, and annotating runtime models. A modifying access is denoted with a square, whereas a read-only access is depicted with a circle. Another dimension is the commonness of a runtime model access by a specific adaptation activity. Therefore, the table distinguishes between an advised access, where an activity *should* read/modify a runtime model, and an optional (uncommon) access, where an activity *can* read/modify a runtime model. The second dimension is encoded by the color, whereas an advised access is denoted by a green, filled rectangle (■) respectively circle (●) and the optional access is depicted by an orange, non-filled rectangle (□) respectively circle (○). The case that an adaptation activity should not access a specific runtime model type is denoted by a red cross (✗).

Table 6.1: Access patterns for combining knowledge and adaptation purposes

		Runtime Model Purpose					
		System	Context	Evaluation	Change	Monitoring	Execution
Adaptation Purpose	Monitor	■	■	○	○	●	✗
	Analyze	□	●	●	○	✗	✗
	Plan	□	●	○	●	✗	✗
	Execute	●	○	✗	✗	✗	●
Component Type	Sensor	■	■	○	○	●	✗
	SWC	□	●	○	○	✗	✗
	Actuator	●	○	✗	✗	✗	●

The adaptation activity/component type ■: should modify; □: can modify;
 ●: should read; ○: can read; ✗: should not access the runtime model type

The first observation from Table 6.1 is that the monitor activity in a FLD as well as the sensor component in an ACD are the both entities, which keep the runtime models concerning the system and context representation up-to-date. Therefore, both entities are responsible for maintaining the causal connection from the underlying system to their runtime model representation, which is represented by the same access patterns in the table. Furthermore, the execute activity respectively the actuator component forces changes from the system runtime model back to the real system, which is indicated by an advised read on the corresponding system as well as execution runtime model. Finally, following the purpose of the analyze and plan activity leads to the access patterns on the corresponding evaluation and change runtime models. In ACD, the planning and analysis of adaptation activities is consolidated in the software component type. Thus, a software component combines the access patterns from both activity types.

For performing the Deurema analysis, the access patterns and their anti-patterns are encoded in rules that detect the corresponding violations. However, the depicted access patterns in Table 6.1 are a proposal based on the presented runtime model category and the discussed purpose of an adaptation activity respectively component in the context of this thesis. Such access patterns support the developer of specifying appropriate model operations for feedback loop activities and runnables by following the defined modeling guidelines. Furthermore, violations are detected that indicate a possible design flow of the template specification, which contributes to the improvement of the overall quality of the modeled adaptive SoS. However, the access patterns in Table 6.1 can vary depending on the domain or concrete problem. Therefore, the existing Deurema analysis rules must be adapted, if other guidelines or violations of runtime model accesses have to be detected.

6.2.3. Combining Knowledge and Causality

The last possible combination of the discussed basic metrics is the investigation of the causality together with the use of runtime models. Investigating this combination leads to an analysis of the knowledge distribution within a module as well as between modules by considering collaborations. Figure 6.15 gives an overview of the analysis rules together with an example situation by means of adaptation activities and runtime models as used in a FLD. Of course, the presented analysis rules are additionally applied on ACD and BRD on basis of the causality and knowledge *-representative property as discussed for FLD.

The first analysis rule in Figure 6.15 refines the *-representative property of a module, if an additional read model operation on the corresponding runtime model can be found. The *-representative property denotes the availability of information, where the additional read access denotes the real usage of that information. As a consequence, the *-representative property turns into a **-awareness* characteristic following the same naming scheme as discussed for Figure 6.9. For example, a self-representative feedback loop module becomes self-aware, if at least one adaptation activity performs a read operation on a system runtime model. Straight forward, the same feedback loop is denoted as context-aware, if at least one read operation on a context runtime model is found. The line of argument holds for the other runtime model purposes accordingly. Thus, the analysis rules of this thesis correspond to the proposed *-awareness properties introduced in [166] and comprehensively discussed in Section 2.1.1.

Furthermore, the Deurema analysis detects a *knowledge modification*, if the knowledge awareness property is fulfilled and an additional modification on the beforehand read runtime model is performed by an adaptation activity. If the reading and modifying model operations

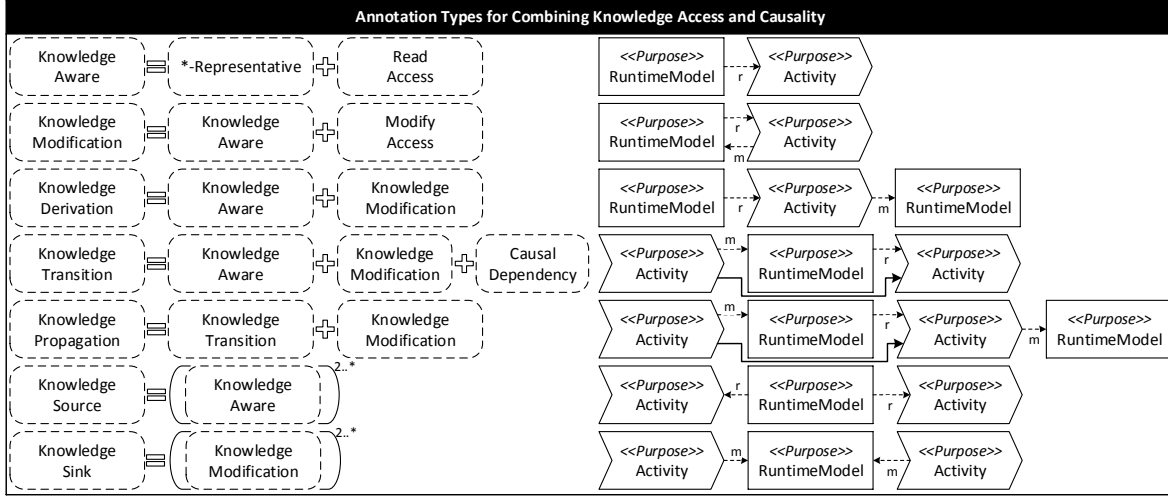


Figure 6.15: Analysis rules for combining knowledge and causality

are subsequently performed on different runtime models, the modification in the runtime model is determined as *derived* on basis of the read information and the performed adaptation step. Therefore, a knowledge derivation is detected in the Deurema model, which is marked with the KnowledgeDerivation annotation shown in Figure 6.15.

Additionally, a *knowledge transition* occurs between two causal dependent adaptation steps, where the former modifies the runtime model and the latter reads the updated information. The propagation of knowledge is investigated by looking at the transitive closure of each found knowledge transition. Thus, a knowledge transition is extended by another causal dependency, which denotes an additional adaptation step.

The last two analysis rules in Figure 6.15 investigate the start and end points of knowledge distribution, which is denoted as *knowledge source* respectively *knowledge sink*. A knowledge source is characterized by at least two distinct read operations from different adaptation activities. In contrast, a knowledge sink needs at least two modifying model operations. In summary, investigating the source, propagation and sink of knowledge completes the picture of knowledge distribution in the corresponding module template. A concrete example for such a knowledge distribution is illustrated between two software components in Figure 5.31 by the discussion of the Deurema ACD in Section 5.3.4.

Leaving the context of a module template, the knowledge propagation rule can be extended with respect to collaborations, which enables the distribution of knowledge across the boundaries of a single module. Reading a runtime model in combination with sending the read information via a model message pinpoints to the overall shared knowledge in the adaptive SoS. As a consequence, Deurema can distinguish between runtime information that is locally used within the context of a module and those runtime models that become visible for other modules due to their sharing in the interaction. This might imply further influence possibilities between modules. If for example a local system model is shared via a model message from a module M_1 to another module M_2 , the receiving module M_2 gets direct access to the runtime model. Due to the causal connection of the runtime model to the underlying reflected system, the module M_2 can indirectly influence the controlled system of M_1 . Thus, the local system model becomes a shared system model, which is explicitly supported by the runtime model categorization as discussed in Section 5.2. Furthermore, the direct (local)

and indirect (shared) influence between modules become visible for the Deurema analysis by looking at the used communication mechanisms and the runtime model purpose. The analysis rules of knowledge propagation between modules are extended to the boundaries of a system. Thus, the distribution aspects become visible for the complete adaptive SoS architecture.

6.2.4. Complex Analysis Rule Combination

Beside the pairwise combination of the analysis metrics concerning the causality, knowledge and adaptation purpose, further complex analysis rules considering all three metrics are thinkable. As outlined above, the Deurema analysis supports the reuse of found annotations and their aggregation to meaningful patterns. For example, the detection of a collecting feedback loop (cf. Collector annotation in Figure 6.11) can be combined with knowledge usage. An additional found annotation that denotes the context-aware property for the same feedback loop leads to an overall combined *context-aware collector*. Another meaningful combination is a *requirements-aware planner*, because the planning of further adaptation steps of the system behavior should always consider the current goals and constraints. Thus, the Deurema analysis can be used to verify the modeled SoS according to development assumptions or design guidelines, where the absence or violations pinpoint to the corresponding parts in the Deurema models. Following this line of argument, it depends on the domain and the concrete problem, which further combinations support the analysis of the overall adaptive SoS. The existing implementation of the Deurema analysis enables the specification of new rules. On the one hand, rules can reuse existing annotation types to aggregate them to new analysis aspects. On the other hand, complete new analysis patterns and their corresponding annotation types can be defined that are useful in the concrete context of the problem domain. In the following, complex architectural patterns as well as design flaws, where a subset can be detected in the running example of this thesis, are discussed.

6.3. Architectural Patterns and Design Smells

The Deurema language distinguishes between hierarchical control and layered adaptation, which is determined by using different Deurema intra-loop coordination concepts. Therefore, both patterns can be detected by corresponding analysis rules. Figure 6.16 shows on the left an example for a hierarchical control architecture, where modules on the lower layer trigger the execution of a module at a higher layer. This pattern occurs in the smart car running example in Figure 5.74 as discussed in Section 5.7. In general, hierarchies allow the decomposition of adaptation concerns in different abstraction levels. Usually, the adaptation logic on the lower layers enables a fast reaction of changing situations, whereas the focus is often specialized to one, locally restricted adaptation concern. Furthermore, if necessary, the local adaptation logic hands over the responsibility of finding an appropriate adaptation strategy by triggering a module at a higher layer. Possible reasons for such a transfer of responsibilities are for example, if the module cannot handle the current situation locally or if an overall optimization of the system behavior becomes important, which exits the focus of the restricted local view. The advantage of this pattern is the fast reaction on lower layers and because of the decomposition of adaptation concerns, the reduced complexity of the low level adaptation logic. In long hierarchies, a disadvantage is the possible deferring of decision making by traversing the hierarchy upwards and propagating the desired behavior into the local modules afterwards. This might cause long timing delays before an adaptation effect of

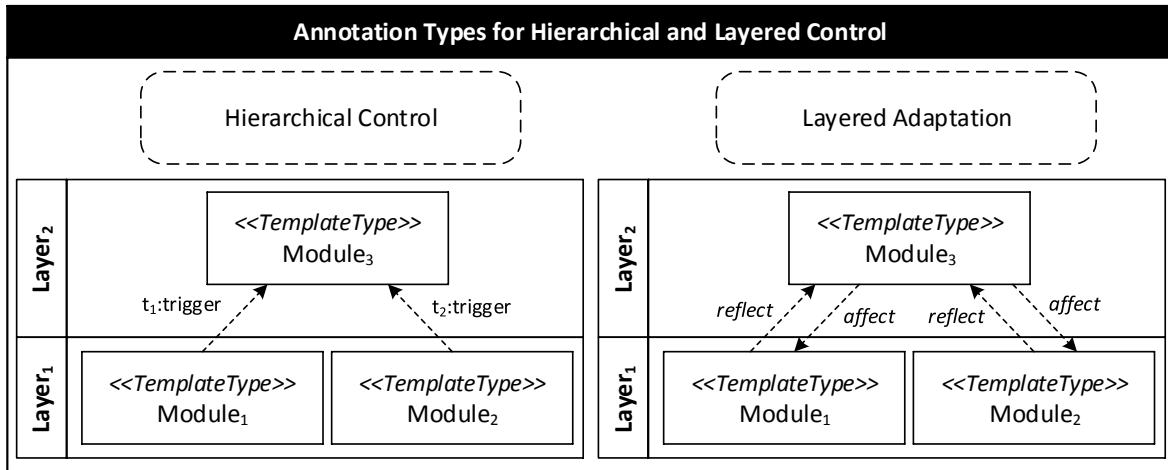


Figure 6.16: Analysis of hierarchical and layered control

the system can be recognized. Thus, this architectural pattern is appropriate for long term adaptation in modules at the highest layer and fast but more simple adaptation decisions on the bottom. However, the hierarchy is designed upfront, whereas the developer cares about a proper decomposition of adaptation concerns. Furthermore, the modules on the lower layer have a notion of responsibilities in the realized hierarchy and know their supervisor at the higher layer.

In contrast, the layered adaptation pattern does not use an explicit triggering but rather the Deurema reflection mechanism as shown on the right in Figure 6.16. In the example, the module at the highest layer reflects both modules on the layer below. Similar to the hierarchical control pattern, the modules at the lowest layer can be designed for adapting to local situations, which allows a decomposition of adaptation concerns and shows similar advantages as discussed above. The difference is that the low level modules are not designed to interact with the modules on the higher layer. In some cases, the lower layer modules even do know nothing about the adaptation logic at the higher layers. Therefore, the adaptation logic at the top can perform its functionality completely independently from the modules at the bottom. On the one hand, this could be useful to decouple long term adaptation planning from short term adaptation that requires a fast reaction. On the other hand, because there is no dedicated triggering (interface) of the adaptation loop on top, it must ensure that the affecting of the underlying adaptive behavior does not lead to inconsistencies in the overall adaptation effects. However, in Deurema, the concepts of hierarchical control by triggering modules and layered adaptation by using the Deurema reflection mechanism can be combined.

Beside patterns, the Deurema analysis detects design flaws in the modeled system. For example, Figure 6.17 shows a trigger dependency between two modules, where the module at the lowest layer triggers a module that is located two layers above. The intention of a layered architecture is that modules placed on a specific layer reason about and influence modules on one layer below as well as may additionally trigger modules one layer above. Thus, the layers in a system define a structural separation of the available adaptation logic. The triggering between the modules in Figure 6.17, which is a causal dependency as discussed above, ignores the modeled layered architecture of the system. As a consequence, the trigger dependency in this example structurally bypasses the adaptation logic in *Module₃* placed at *Layer₁*. In general, design flaws are no modeling errors and thus, can occur during the development by

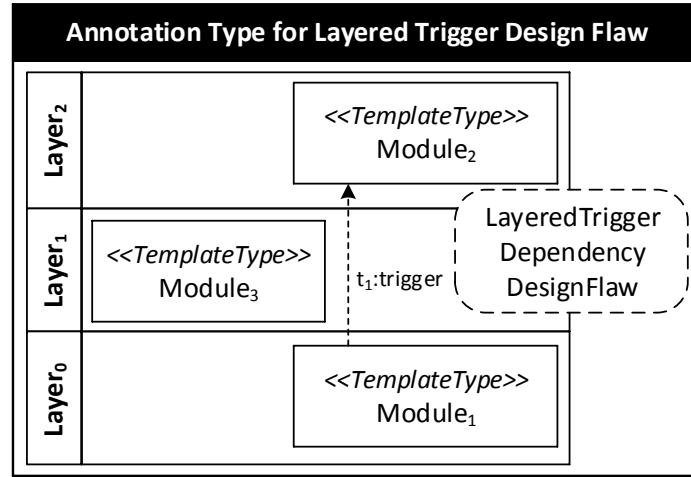


Figure 6.17: Layered trigger design flaw

modeling the adaptive SoS with the Deurema approach. However, design flaws are error prone, where the resulting adaptation effect of the SoS may differ from the modeled expectations or can cause further inconsistencies with other modules. Therefore, modeling guidelines and patterns help to improve the overall modeled architecture, which leads to more robust system designs or at least to a better understanding of the modeled adaptation functionality. As a consequence, the design flaw in Figure 6.17 is detected by the Deurema analysis by means of a found annotation in the Deurema model. If the developer checks the analysis result, he can improve the modeled SoS solution by removing such design flaws afterwards. However, design flaws are no modeling errors. Therefore, the developer may decide to ignore the found design flaw annotations and interprets those as warnings, which is also supported by Deurema.

Another example for a design flaw is the indirect coupling of adaptation activities within a feedback loop over the available knowledge, which may contradict the modeled causal order of the adaptation effects. Figure 6.18 shows an example for such a causality knowledge design flaw. There are two distinct paths of the adaptation behavior within the feedback loop, where the first path comprises the activities *Activity*₁ as well as *Activity*₂ and the second path consists of *Activity*₃ and *Activity*₄. According to the causality discussion between activities above, there is a causal dependency between the activities in each path. Furthermore, there is no causal dependency between the activities located in different paths. However, there is a potential additional coupling between *Activity*₁ and *Activity*₃ over the runtime model, where the former activity modifies the common knowledge base and the latter activity reads the modified information. Therefore, the potential influence between these two adaptation activities over the runtime model may contradict the specified control flow of the feedback loop of the distinct paths. Whether this indirect coupling between *Activity*₁ and *Activity*₃ introduces undesired adaptation effects or not depends on the concrete situation at runtime and intention of the stored data in the runtime model. For example, the feedback loop is usually executed periodically. Thereby, the *Activity*₁ may collect data over time in the runtime model, which is later analyzed by *Activity*₃. In this case, the indirect coupling is explicitly desired behavior and the detected design flaw can be interpreted as warning and thus, ignored. However, the Deurema analysis can pinpoint to such design flaws, whereas the model developer must decide whether the modeled solution conforms to original intentions or not. Especially,

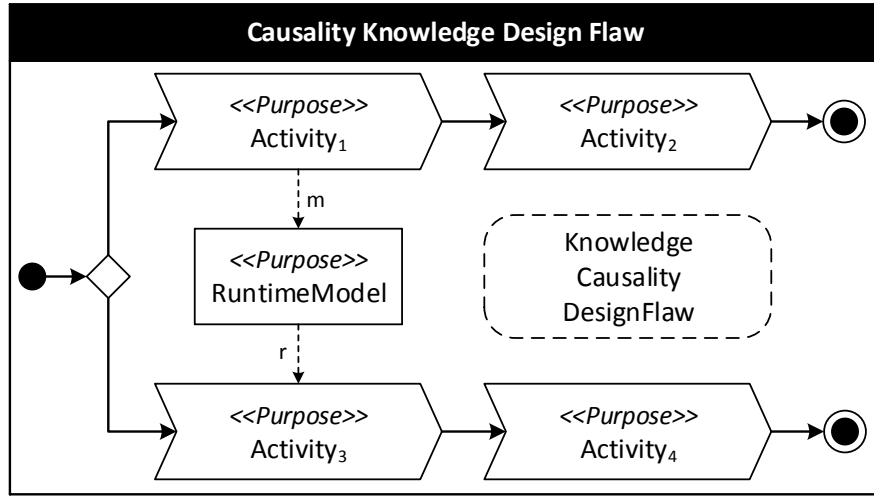


Figure 6.18: Analysis of causality knowledge design flaw

in feedback loops comprising different paths, several runtime models, and multiple adaptation activities, a static analysis helps keeping track of the overall modeled adaptive functionality.

A much more hidden indirect coupling of adaptation effects is shown in the architectural design flaw in Figure 6.19. Two independent behavior modules, which are further contained on different architectural layers in the system, comprise an adaptation rule, where the RHS of R_1 is causally connected to the LHS of R_2 (cf. discussion about causality in BRD in Section 6.1.1). Because there is no obvious trigger dependency between these two modules, this coupling cannot be detected by looking at the Deurema LD alone, where the internals of the module instances are hidden.¹ Furthermore, the corresponding module templates can be independently specified from different developers. In this example, the causal dependencies between the rules leave the border of the parent module and even further, violate the layered architecture of the system. The Deurema analysis detects such hidden dependencies and therefore, helps the developer improving the overall modeled SoS architecture.

In summary, the Deurema analysis detects such violations in the modeled system and hidden dependencies become visible for further investigation. Thereby, the concrete design patterns and modeling guidelines depend on the underlying problem domain, where individual analysis rules can be added to the existing Deurema verification framework. Deurema already provides analysis rules for the three basic metrics causality, knowledge, and adaptation purpose. Furthermore, meaningful combinations of these metrics are discussed as well as high level architectural patterns and design flaws are exemplary introduced. All analysis rules are statically applied on the modeled adaptive SoS during the development, where detected design flaws pinpoint to possible problems.

¹For the sake of this example, the internals of the module instances are indicated in Figure 6.19. However, modeling the system architecture in Deurema does not pinpoint to the internals of the corresponding module template as discussed in Section 5.4.

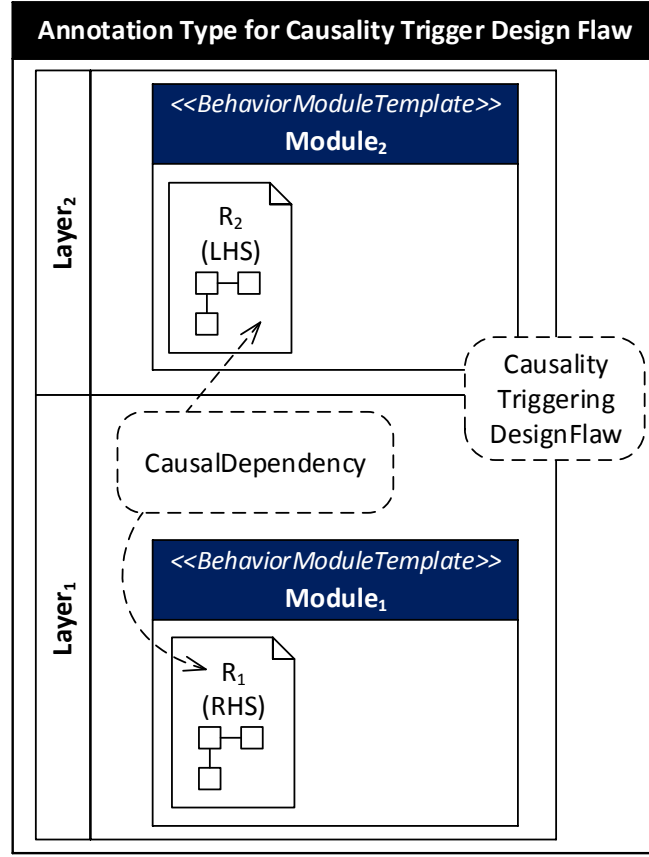


Figure 6.19: Analysis of architectural design flaws

6.4. Discussion

In summary, the Deurema analysis framework facilitates the static analysis of the adaptive SoS by applying specified analysis rules directly on the Deurema models. This thesis proposes several basic metrics concerning the causality, knowledge, and adaptation effects, which are realized by corresponding analysis rules. Furthermore, the basic metrics are combined to investigate the distribution of knowledge within the system, to predict the emerged interaction effects of the local adaptation logic, up to the detection of architectural patterns within the layered systems. However, all presented aspects in this chapter are meaningful proposals that can be extended to the specific underlying problem. Thus, this thesis does not claim that the presented analysis metrics are complete nor that all important aspects of a specific domain are covered. Nevertheless, it presents examples how basic metrics are detected and afterwards combined to reason about complex situations throughout the modeled SoS. The extension of the static analysis is very easy by defining new analysis rules or creating new combinations of existing metrics that target the corresponding key points of interest.

This thesis uses the deductive inference engine from Beyhl et al. [35] for implementing the Deurema analysis framework. The inference engine is an implementation of a model management approach as introduced in the preliminaries in Section 2.2.4. It considers the Deurema models as graphs, which are formally defined by the Deurema metamodel. The inference engine provides a graphical editor to specify the analysis rules as well as to use existing

rules and combine them to complex rules. Furthermore, it is able to execute the analysis rules on the underlying Deurema model detecting corresponding matches in the modeled system. These matches are directly marked in the Deurema model in form of annotations as motivated at the beginning of this chapter. Thereby, the inference engine maintains the annotations as well as enables their combination to infer higher level patterns as comprehensively discussed above. All implemented analysis rules of this thesis together with an overview about all realized metrics can be found in the Appendix C. All of these rules are implemented in the language of the inference engine, which enables a direct execution of the analysis rules. Details of the maintenance of found annotations as well as the language semantic of the inference engine are comprehensively discussed in [35]. The following chapter describes the Deurema simulation framework for executing modeled adaptive SoS architectures, which is another possibility of investigating the emergent SoS behavior on basis of defined Deurema models.

7. Simulation

Model-based simulation helps understanding the interplay of system interactions as well as the modeled adaptation effects in the overall SoS. This chapter introduces the Deurema simulation capabilities introducing an execution framework for Deurema models. As sketched in Figure 7.1, the starting point for a model-based simulation is a system template description modeled with the Deurema approach. The system template may comprise several module instances, subsystems, and collaborations. Therefore, the developer decides which parts of the adaptive SoS should be simulated by placing the corresponding module instances in the system template specification. As a consequence, there are different possibilities for a Deurema model simulation by varying the number of modules and collaborations in the system template. For example, the Deurema simulation framework is able to execute a single module instance, e.g., a feedback loop, which enables the investigation of a single adaptive self-* behavior capability. Of course, the simulation can be extended by multiple feedback loops investigating if the independent adaptation logic works as expected. Later, collaborations can be integrated that coordinate the independent adaptation effects or other subsystem instances can be deployed, which facilitates the simulation of the emergent adaptive SoS behavior. Therefore, beside the analysis of the modeled system as comprehensively discussed in the former chapter, the Deurema simulation framework aims at the investigation of the modeled SoS towards an understanding of the emergent behavior. If the simulation shows unwanted co-adaptation effects, contention due to system interactions, or violations of requirements, the developer can change the underlying Deurema model and start the simulation again.

There are different responsibilities for the execution and simulation of the adaptive SoS architecture modeled with Deurema. As sketched in the running example in Figure 7.1, the system architecture as defined by the system template with all included instances and their corresponding template descriptions is the starting point for a simulation. During the simulation, the initial architecture may change due to adaptation effects or new collaborations between systems. According to the running example, if the simulation executes the traffic flow of the smart city over time, new smart cars enter or leave the city. Furthermore, platoon collaborations may appear or disappear according to the needs of the driver and the available smart cars, which can build such a platoon interaction. However, during a simulation, the Deurema interpreter realizes the execution of a single Deurema element, e.g., an adaptation activity within a feedback loop. Each element in a Deurema model follows a predefined state model during its execution. For example, if the Deurema interpreter executes an adaptation activity, it marks the corresponding activity as well as the parent feedback loop as active indicating their execution. Therefore, executing different Deurema elements over time, e.g., by following the defined control flow of the feedback loop, changes the states of the corresponding feedback loop module. Module and system instances are considered as independent except for defined collaborations. Thus, executing the independent instances leads to several state changes over time within the overall SoS. The interplay between Deurema elements, which are possibly in different states, are handled by the Deurema simulator. During the execution, the interpreter marks corresponding state information directly in the Deurema model, which can be recognized by an inference engine. The same inference engine as outlined in the previous

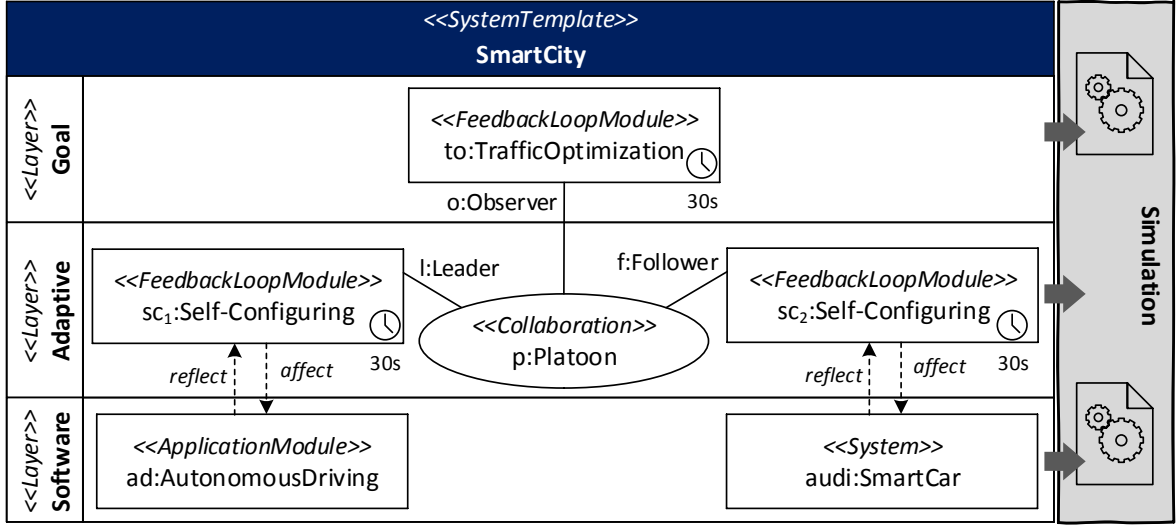


Figure 7.1: Smart city running example: Deurema simulation

chapter is used to reason about the Deurema model and the annotated state information. Furthermore, the inference engine detects state changes during the simulation. Simulation rules help retrieving the state of Deurema model elements, e. g., an active adaptation activity, which is afterwards used by the simulator to pick the next element for execution. The simulator hands over the picked element to the interpreter, which executes the element. The execution causes the modeled adaptation effect in the SoS and changes the state of the corresponding element, which is again detected by the inference engine. Thus, the inference engine uses the simulation rules to capture the overall states of each modeled element in the adaptive SoS. The Deurema simulator provides different scheduling strategies to pick the next element for execution. Finally, the interpreter realizes the modeled adaptation effect by executing the beforehand picked element. The three parts consisting of the Deurema interpreter, simulator and used inference engine are denoted as Deurema execution environment for the rest of this thesis.

In the following, the Deurema execution state model is introduced in Section 7.1, which is the basis for the simulation of Deurema elements. Afterwards, the execution semantic of each Deurema model element realized by the Deurema interpreter is discussed in Section 7.2. Subsequently, the Deurema simulator together with different simulation strategies is outlined in Section 7.3. This chapter discusses an exemplary simulation run in Section 7.4 showing the interplay between the Deurema interpreter, inference engine, and simulator. Afterwards, the idea of combining the Deurema analysis and model simulation towards a runtime analysis are highlighted in Section 7.5. The Deurema simulation capabilities are summarized in Section 7.6. All implemented simulation rules, which are used by the inference engine to maintain the states for each Deurema model element, are enumerated in the Appendix D. The simulation rules are created in the syntax of the inference engine tool.

7.1. Execution State Models

Deurema distinguishes between an execution state model for module and interaction instances as well as Deurema elements, which are contained in module templates. Modules and interactions are container that aggregate and encapsulate multiple adaptation effects, e. g., a feedback loop, which is defined by a corresponding template description. In contrast, Deurema module elements (e. g., behavior models) are contained in modules or interactions and specify a concrete adaptation effect, e. g., an activity within the feedback loop.

7.1.1. Modules and Interactions

The state model for modules and interactions is depicted in Figure 7.2. Modules are the basic entity for execution in an adaptive system that define the adaptive capabilities and are deployed on a certain layer in the architecture of the corresponding system template (cf. Section 5.3). Furthermore, interactions belong to a system collaboration, which is also deployed in the adaptive, layered system architecture (cf. Section 5.5). Both, modules and interactions follow the same execution state model, whereas in the following each state in Figure 7.2 is explained in the context of a module, but is applicable for interactions as well. Differences between both are explicitly highlighted in the state discussion below.

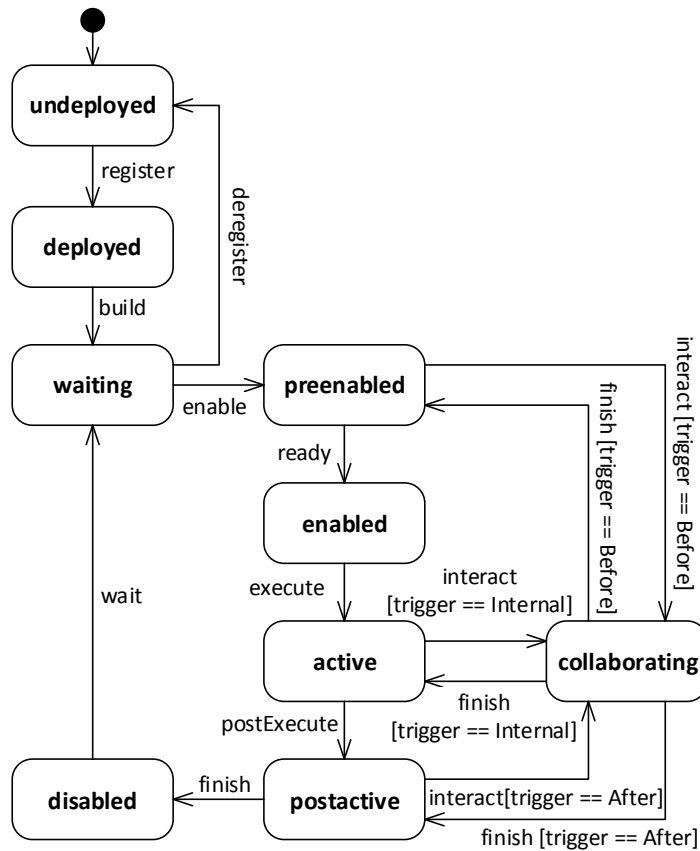


Figure 7.2: Deurema state model for modules and interactions

State: undeployed

A module is considered as *undeployed* if it is not placed on a system layer in the overall adaptive SoS architecture defined by the system template. Therefore, there is no module instance in the Deurema model and thus, the module is not considered for execution or simulation.

State: deployed

An undeployed module can be registered in the Deurema execution environment by instantiating a module template and thus, creating a corresponding module instance, which is placed on a layer in the system architecture. A successfully registered module changes into the *deployed* state afterwards.

State: waiting

After the deployment, a module is preprocessed by the Deurema execution environment. Thereby, the execution environment checks the specified Deurema model and refuses syntactically incorrect models, e. g., a missing action specification for software modules as discussed in Section 5.3.3. If the module is syntactically correct, it becomes visible for the inference engine and waits for further execution, which is denoted by the *waiting* state. Beside the deployment, a module in the waiting state can be deregistered, which removes the module from the execution environment. As a consequence, the inference engine and the simulator do not further consider the corresponding module for execution.

It has to be noted that Deurema supports the deployment of a complete system architecture, which is a system instance that is placed on a system template. The corresponding subsystem may comprise several layers, multiple module instances, and further collaborations. This corresponds to the openness characteristic of the adaptive SoS (cf. Section 3.1), where complete systems may join or leave the overall SoS during its lifetime. The Deurema execution environment unfolds the system architecture as well as contained subsystems and deploys all module and collaboration instances accordingly. Therefore, Deurema can dynamically handle a varying number of system and module instances during the simulation of the adaptive SoS.

Figure 7.3 shows one simulation rule in the concrete syntax of the inference engine for detecting deployed modules and interactions (denoted as *AbstractModule* class in the rule). The specified rule pattern searches for a deployed instance and its corresponding template description. Thereby, the inference engine retrieves the deployed abstract module annotation (the dashed rectangle in Figure 7.3) beforehand by means of other simulation rules during the registration process for modules as outlined above. For each deployed instance and its template description, the inference engine creates a corresponding waiting annotation directly in the Deurema model denoting the availability of the adaptation logic for further processing.

State: preenabled

A module changes its state from waiting to *preenabled*, if all specified triggers are fulfilled (cf. Deurema module triggering concept in Section 5.4). The triggering of a module comprises the modeled event triggers, the time trigger, and the module trigger conditions. In contrast to a module, an interaction becomes preenabled, if the corresponding player module changes to the collaborating state (see description below).

State: enabled

If the module is preenabled and it does not participate in a collaboration, it becomes immediately *enabled*. Interactions cannot participate with other interactions, thus the collaborating state on the right in Figure 7.2 is omitted for interaction instances. Therefore, interactions

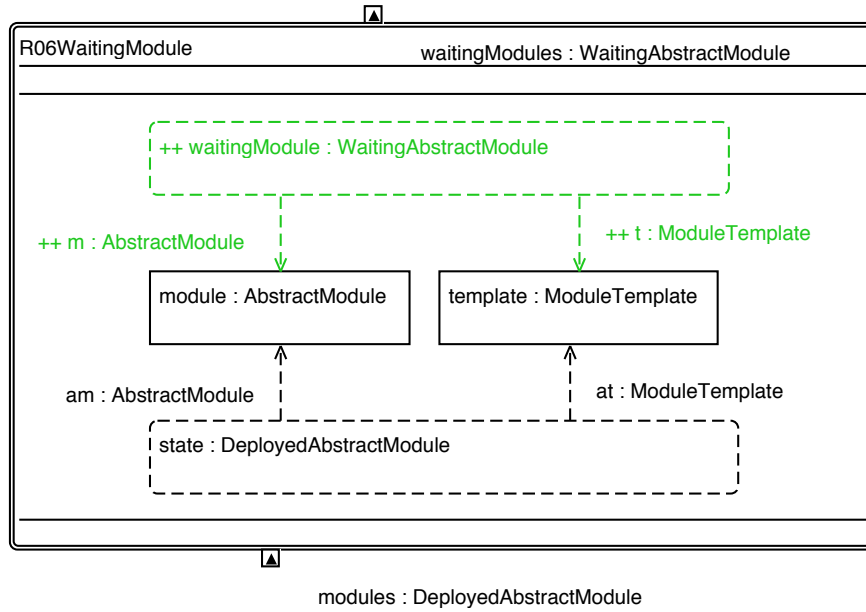


Figure 7.3: Simulation rule for detecting waiting modules

become directly enabled, if they are preenabled before. Enabled modules and interactions are ready for execution and thus, can be picked by the Deurema simulation environment.

State: active

The *active* state denotes the execution of the module. Therefore, the Deurema simulator picked the enabled module beforehand and hands it over to the Deurema interpreter. The interpreter marks the module instance as active, which can be further detected by the inference engine. Furthermore, the interpreter retrieves the template description of the module instance and all contained elements become visible for the simulation environment, e. g., the operations of a feedback loop module or the concrete interaction behavior of a role. The visibility of the concrete adaptation effects are denoted by the waiting state as discussed below for module elements (cf. Figure 7.5). The module stays active as long as an included adaptive behavior is possibly enabled. For example, once a feedback loop module is active, it stays active as long as the execution of the adaptation activities reaches a final or destruction node of the feedback loop.

With respect to interactions in a collaboration, the active state implies that there is another module in the collaborating state, which plays this active interaction as specified in the role mapping of the module.

State: postactive

A module becomes *postactive* after no internal adaptation behavior is possibly enabled, e. g., because the feedback loop reaches a final node. In this state, the module can collaborate again with other modules by playing the modeled role or become disabled.

State: disabled

The *disabled* state denotes the complete execution of the module. The simulation environment cleans up all marked annotations and the module waits for its next triggering by changing in the waiting state again.

State: collaborating

This state can be reached from the preenabled, active, and postactive state of a module. According to the interaction trigger discussion in Section 5.5.6, a Deurema module can play the deployed role at different points in time, which is modeled by a role trigger. The Deurema metamodel defines three possible role trigger that are Before, Internal, and After. The Before trigger denotes the execution of the interaction behavior before the local, internal adaptation behavior of the module is executed. Therefore, a successfully triggered module, which is denoted by the preenabled state, change its state to *collaborating*, if the corresponding role trigger is defined as Before. Of course, a corresponding module interaction must be defined for that module. The states for modules and interactions are independently handled in the Deurema execution environment. Thus, there are two state transitions denoting a collaboration. First, the player module changes from the preenabled to the collaborating state. Second, the corresponding interaction changes from the enabled to the active state.

Furthermore, the Internal role trigger denotes the execution of a modeled interaction within the execution of the player module. In this case, the interaction is woven into the local adaptation behavior as discussed in Section 5.5.5. Therefore, the module must be in the active state and change for each interaction in the collaborating state accordingly.

Straight forward, the After role trigger denotes the execution of the collaboration behavior after the local behavior of the module is executed. This is represented by the postactive state as discussed above and explicitly considered in the execution state model in Figure 7.5. The coupling of the two states of the module and the corresponding interaction is reflected in the example simulation rule in Figure 7.4.

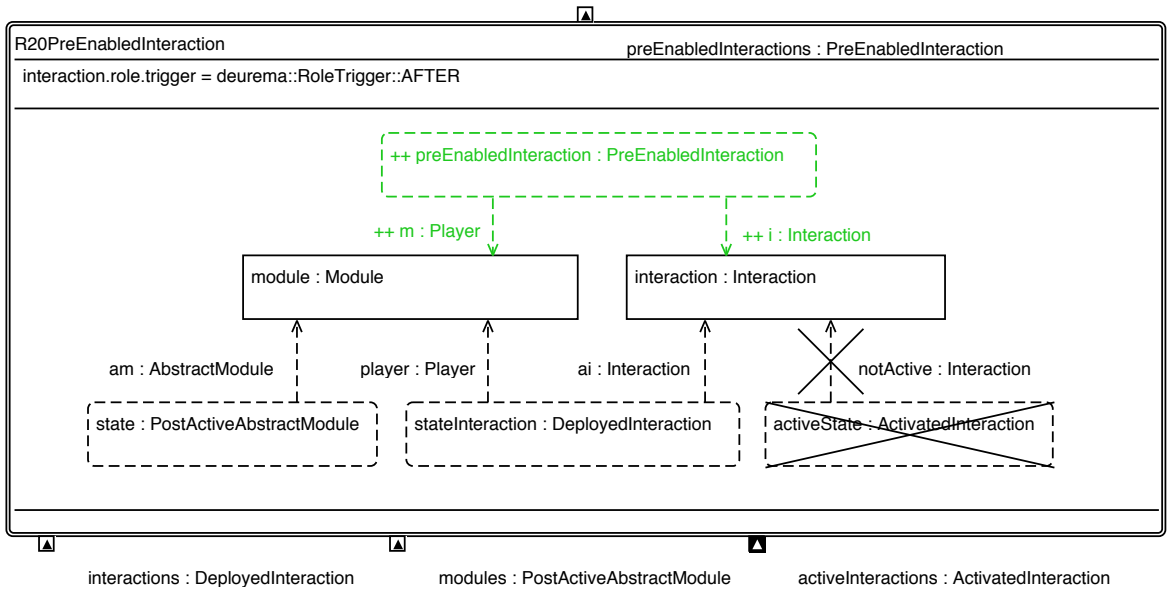


Figure 7.4: Simulation rule detecting a preenabled interaction for an After role trigger

The inference engine searches for an interaction, where a corresponding module is the player of the mapped collaboration role. Furthermore, because of the After role trigger, the module must be in a postactive state denoted by a corresponding state annotation. If this combination is found, the inference engine marks the interaction as preenabled, which corresponds to a state switch. As mentioned above, a preenabled interaction becomes enabled afterwards and thus, is ready for execution by the Deurema interpreter. The depicted simulation rule focuses

on the *After* role trigger. The simulation rules for the other role trigger combinations can be found in the Appendix D.

As a consequence of the explicit modeled collaborating state for modules, the Deurema execution environment can clearly distinguish at every point in time between the execution of local adaptation behavior of a corresponding module and its interacting behavior within the collaboration. Furthermore, as long as the module is considered as active or collaborating, contained Deurema behavioral elements are visible for the simulation environment and thus, can be executed.

7.1.2. Module Template Elements

The state model for elements that are contained in a module template is depicted in Figure 7.5. There are different element types in a module depending on the corresponding module template. For example, a FLD contains operations as initial nodes, adaptation activities, or final nodes. A BRD contains adaptation rules and an ACD contains components, tasks as well as runnables. In the following, each state is described for those elements, whereas the term *adaptation effect* is chosen as representative.

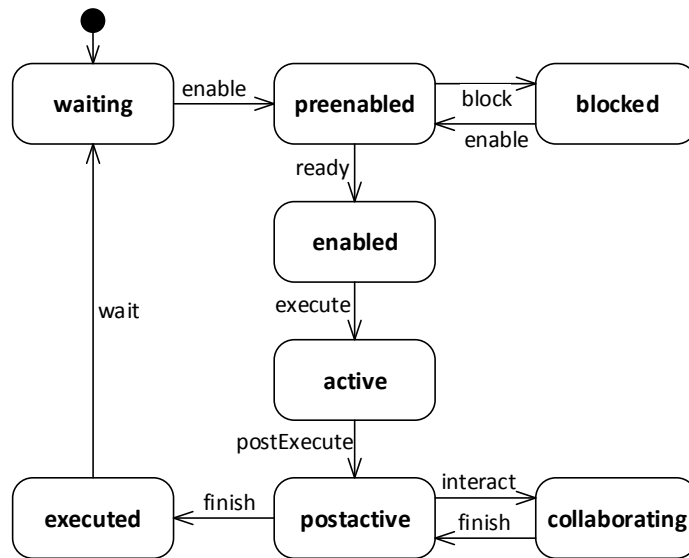


Figure 7.5: Deurema state model for contained module elements

State: waiting

If the parent module becomes active, all contained elements enter the *waiting* state. As a consequence, they are visible for the execution environment. Therefore, the inference engine reasons about these adaptation effects to determine next possible states and track state changes during their execution.

State: preenabled

This state denotes that the adaptation effect (e.g., an activity in a feedback loop or a behavior rule) fulfills all preconditions to be executed. These preconditions depend on the module template type and its contained elements. A behavior rule in a BRD is *preenabled*, if a match for its LHS is found and all behavior rule properties are fulfilled (cf. discussion in

Section 5.3.5). A runnable in an ACD is preenabled, if the corresponding task is active and the previous runnable in that task is active (cf. description in Section 5.3.4).

As an example, Figure 7.6 shows the simulation rule for detecting a preenabled operation in a feedback loop. An operation is preenabled, if the previous operation is active and the specified control flow denotes the operation as the following adaptation effect. Therefore, the active operation references the following operation via the next reference as defined in the Deurema metamodel (cf. Section 5.3.2). If the following operation is in the waiting state, which implies that it is not active, an appropriate preenabled state annotation is created by the inference engine for that operation. All other simulation rules for detecting preenabled adaptation effects, as for example the initial node in a feedback loop, runnables in a task, or behavior rules, are enumerated in the Appendix D.

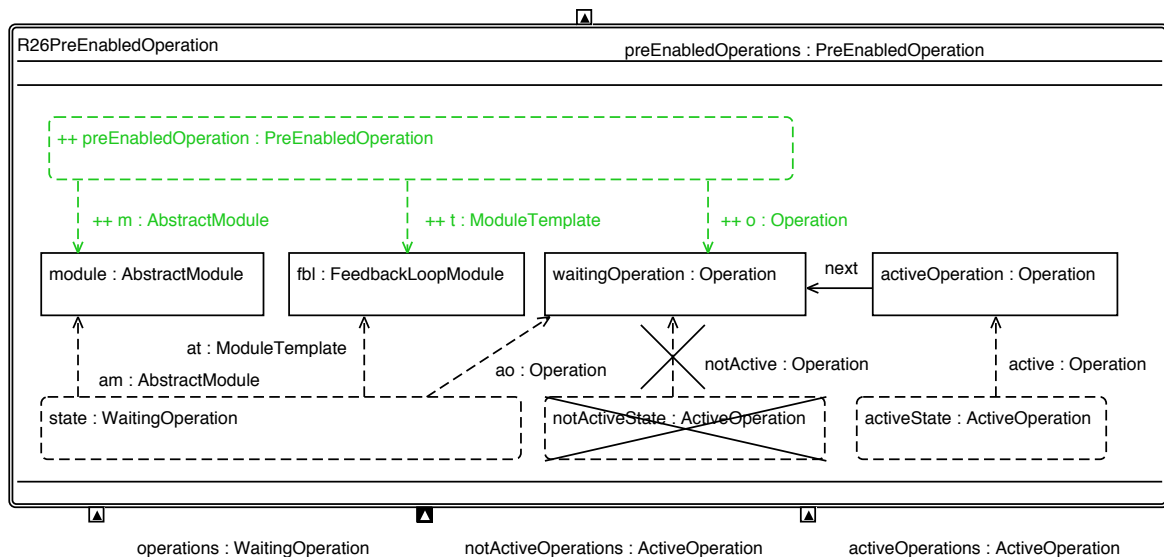


Figure 7.6: Simulation rule for detecting a preenabled operation.

State: blocked

This state denotes that the preenabled adaptation effect is *blocked* by an active interaction. Therefore, another adaptation effect in the same module currently collaborates with another module. The blocked adaptation effect has to wait until the interaction is performed. Afterwards, it becomes preenabled again.

State: enabled

If no active interactions are performed in the parent module, the preenabled adaptation effect becomes *enabled*. In this state, the adaptation effect can be picked by the scheduler for further execution.

State: active

Similar to modules, this state denotes the *active* execution of the adaptation effect by the interpreter.

State: postactive

In this state, the local adaptation effect is completely executed and thus, denoted as *postactive*. Afterwards, collaborative behavior can be enabled via an interaction.

State: collaborating

In the case of an available interaction, a local adaptation effect changes in the *collaborating* state denoting the active collaboration with another module.

State: executed

In this state, the adaptation effect is processed for cleanup by the execution environment. This implies that the local adaptation effect was already *executed* and there is no collaborating activity. After the cleanup, the adaptation effect waits again for its execution. Furthermore, the parent module can only be disabled (cf. state model for modules and interactions in Figure 7.2), if all local adaptation effects are in the states executed or waiting.

In summary, the Deurema execution environment maintains for each modeled element an individual state, which denotes the current status of that element during a simulation. Enabled model elements can be picked by the Deurema simulator and active elements are currently executed by the Deurema interpreter. The current state is directly marked in the Deurema models, whereas transitions between states are encoded in simulation rules. An inference engine can directly execute these simulation rules on the modeled adaptive SoS, which creates new state annotations in the model. During a simulation, the states change, which is observed and maintained by the inference engine, too.

7.2. Interpreter

Beside the Deurema execution state models, this section describes the concrete execution semantic for each Deurema model element. Thereby, each element defined in the Deurema metamodel is considered. Executing elements is realized by the Deurema interpreter. The interpreter is only responsible for the execution of model elements and does not maintain any states. Therefore, it executes every element, which is picked by the Deurema simulator beforehand.

7.2.1. Execution Semantic Module

Deurema modules are execution entities, which are placed in the layered system architecture. Therefore, systems are not directly executed by the interpreter. The hierarchical, layered architecture of a system is unfolded and the contained modules become visible for the execution environment. A module can be executed by the interpreter, if the trigger conditions are fulfilled. The interpreter marks the module as active, where the contained internal adaptation activities as defined by the corresponding template become visible for further execution. Exceptions are software modules that encapsulate black box functionality, which are handled as behavior models as discussed below. After all internal adaptation effects of the module are executed, all specified event trigger are emitted by the interpreter and the list of incoming event trigger is reset. Furthermore, the optional time trigger is reset and the corresponding module becomes disabled for the specified waiting period of the time trigger.

The adaptation effects within a module can be conceptually distinguished into two parts. First, there are elements that are provided and fully controlled by Deurema. Second, behavior models, e.g., an activity or runnable, can contain domain specific adaptation logic, which leaves the focus of the Deurema modeling language. The following section describes the elements that are fully controlled by Deurema. Afterwards, the execution semantic of behavior models is discussed.

7.2.2. Execution Semantic Deurema Elements

Feedback loops offer different operation types for specifying the intra-loop coordination between adaptation activities as summarized on the left in Figure 7.7. Initial nodes are the starting point of a feedback loop, whereas their execution is very simple. The interpreter marks an initial node as active, which will enable the subsequent operation following the control flow over the next reference. Beside the active marker, the execution of an initial node has no further adaptation side effect.

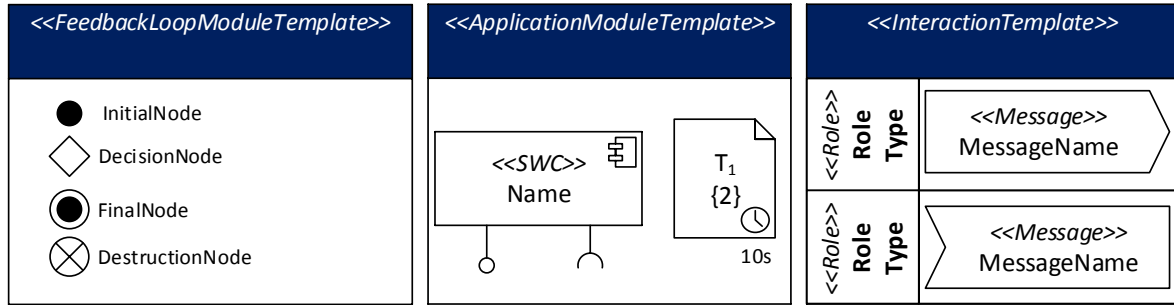


Figure 7.7: Deurema controlled elements

A decision node branches the control flow within the feedback loop. Therefore, the interpreter successively evaluates the specified branch conditions. The first condition, which returns true, enables the corresponding path for further execution. The evaluation of the other conditions is aborted. Furthermore, the order of evaluating the branch conditions is nondeterministic and the developer has to ensure that one of the branch conditions evaluates to true. Otherwise, no following operation can be determined and the overall simulation of the feedback loop ends in a deadlock.

The execution of a final node ends the overall feedback loop. Therefore, all state marker from the contained feedback loop operations are removed and the parent module is marked as postactive. A destruction node has the additional effect, that the module instance is removed from the system template architecture. Therefore, the feedback loop module instance cannot be executed again. Thereby, the corresponding template description is still available for other module instances.

An ACD facilitates the specification of a component-based adaptation logic. Components and corresponding ports encapsulate the available adaptation logic specified in form of runnables. Because both Deurema elements define the structure of the corresponding module, they are not executed by the interpreter. In contrast, tasks define executable groups of runnables. During execution, the interpreter marks a given task as active with the consequence that all mapped runnables are visible for further execution. If the last runnable in the task is executed, the complete task becomes inactive and the interpreter cleans all state marker from the mapped runnables. Furthermore, the period timer of the task is reset so that the simulation environment can determine the next time slot, where the task is enabled again.

Interactions in collaborations use the additional Deurema message concept defining the interaction protocol. The specified message properties define the execution behavior as comprehensively discussed in Section 5.5.3. Therefore, if a message is enabled for execution or not is encoded in appropriate simulation rules. The interpreter is only responsible for realizing the execution effect, which is rather simple in contrast of determining enabled messages. First, the Deurema interpreter determines the direction of the message (sender or receiver). For

sending a trigger message, the interpreter retrieves all receivers and stores the corresponding trigger event in their local queue. Straight forward, receiving a message is realized by retrieving and deleting all events from the local event queue. For a model message, the beforehand retrieved amount of data is stored into respectively retrieved from the local queue. A service is considered as behavior model as discussed below. Again, if the interaction can proceed after the message is sent/received depends on the defined message properties and belongs to the responsibility of the simulator and not to the Deurema interpreter.

7.2.3. Execution Semantic Behavior Model

Deurema behavior models integrate domain specific behavior into the adaptive SoS architecture. As comprehensively discussed in Section 5.3.2, Deurema supports behavior models following a black-gray-white box concept. Behavioral black boxes point to a domain specific implementation that can be invoked by the Deurema interpreter (cf. Figure 5.21). Gray boxes specify an additional trigger condition known by Deurema, which describes the applicability of the behavior model. The adaptation implementation remains unknown in the gray box case. White boxes define the adaptation effect in form of a graph transformation rule by means of a trigger-action condition, which is completely known by Deurema. Therefore, white box behavior can be fully analyzed with the Deurema analysis concept as outlined in Chapter 6. Figure 7.8 subsumes the different behavior model types within the Deurema modeling language.

On module level, a software module allows the integration of domain specific behavior, which can be directly placed on the layered system architecture. The execution steps are highlighted in the running example *Wheel* software module in Figure 7.8. First, for step (1), all interactions for all played collaborations with the specified role trigger *Before* (cf. Section 5.5.6) must be executed. Detecting enable interactions belongs to the responsibility of the inference engine and choosing the interactions for execution belongs to the simulator. However, each module can only be executed if all of these interactions are successfully processed. The execution of a software module starts with an update of the local knowledge base by executing all reflect module operations denoted with step (2) in the figure. Afterwards, the optional trigger condition is evaluated in step (3). If no trigger condition is specified or if the trigger condition is evaluated to true, the behavior implementation is invoked in step (4). Step (4) to (6) are skipped, if the trigger condition is not fulfilled. Executing the adaptation effect might affect other modules denoted in step (5). After the successful execution of the adaptation effect, all interactions of played collaborations with the *After* role trigger are ready for execution denoted as step (6). Finally, the interpreter emits all outgoing events in step (7) and resets the period of the timed trigger in step (8).

The execution of feedback loop activities in FLD, behavior rules in BRD, runnables in ACD, and services in interactions follows the same principle and is sketched at the bottom in Figure 7.8. At first, all reflecting module operations in step (1) are updated, which point to a runtime model view that is read in step (2) by the behavior model afterwards. Note, reflect and affect module operations are specified between modules only. The Deurema view delegation concept is used to delegate the reflected information to the corresponding runtime model view (cf. Section 5.6). For the sake of simplicity, the delegation is omitted for the feedback loop activity on the lower left in Figure 7.8 and the reflection module operation directly points to the runtime model view. However, after the update of the local knowledge by reflecting (1) key points of interest and reading (2) the runtime model data, the trigger (3) and action (4) is executed for the behavior model. The execution of the concrete adaptation

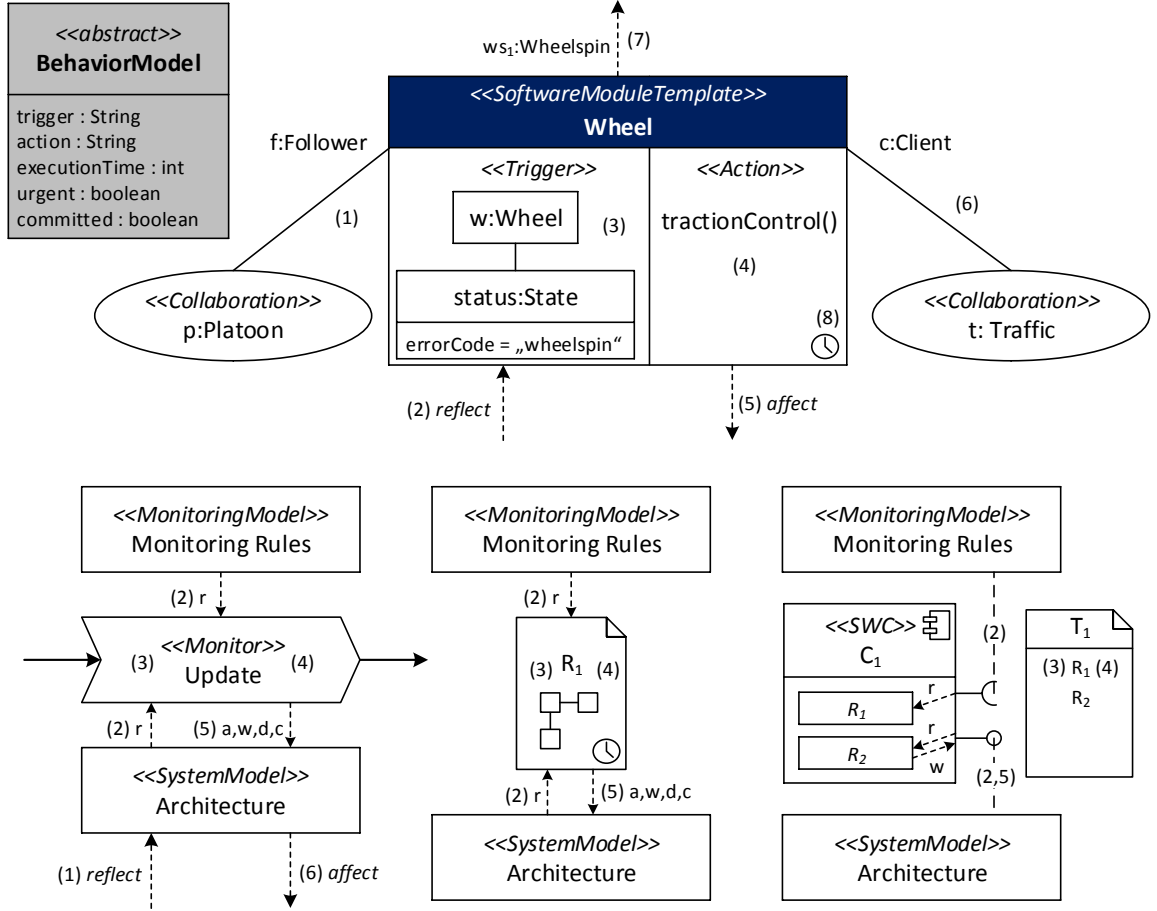


Figure 7.8: Deurema behavior models

effect depends on the specified black-gray-white box situation. For all three situations, the optional trigger condition is evaluated first in step (3). Only if the trigger condition indicates the further application of the adaptation action (the return value of the trigger condition is true), the defined action is executed as denoted by step (4). For black boxes, Deurema subsequently hands the beforehand retrieved knowledge over to the domain specific trigger and action implementation. Because the trigger condition is known for gray boxes, the Deurema interpreter applies the specified condition and invokes the unknown domain action afterwards. For white boxes, the trigger and action are combined in a graph transformation rule, which can be directly applied by the interpreter. During the execution of the adaptation effect, the Deurema execution environment recognizes changes on the underlying runtime models depicted as modifying runtime model operation in step (5). Therefore, modifying model operations (e.g., write or annotate in step (5)) as well as affecting module operations in step (6) can be applied by the Deurema interpreter accordingly.

Model operations consist of model queries, whereas each query is considered as executable behavior model (cf. Figure 5.22). The trigger attribute corresponds to a knowledge retrieval step and the action to an optional filter operation, which is applied on the retrieved data. The Deurema interpreter first invokes the retrieval implementation that can be a domain specific function (black box) or a graph pattern (gray and white box). Afterwards, the optional filter

operation is invoked, which is always domain specific. A discussion about all combinations of black-gray-white box behavior models and model operations is given in Section 5.3.2 in Table 5.3.

7.2.4. Execution Semantic Interaction

The execution of an interaction, which is woven into the local adaptation behavior of a module, is similar to a behavior model explained above. The execution steps are depicted in Figure 7.9. The Deurema interpreter updates the knowledge base by performing all reflecting (1) module operation dependencies and read model operations (2). The retrieved knowledge is handed over to the interaction. The internals of the interactions are defined by a corresponding template definition and depend on the played role. Due to the Deurema metamodel, an interaction is a feedback loop template (with an extended message concept) and thus, executed in the same way as above explained for modules. The interpreter marks the interaction instance as active denoted by step (3), where the contained operations become visible and are further processed by the Deurema simulation environment. Incoming (2) and outgoing (4) knowledge of an interaction is defined in the knowledge collaboration specification as discussed in Section 5.5.2. Therefore, the interpreter can track changes in the runtime model during the execution of the interaction, where the data becomes visible for the player module after the interaction is executed (step (4)).

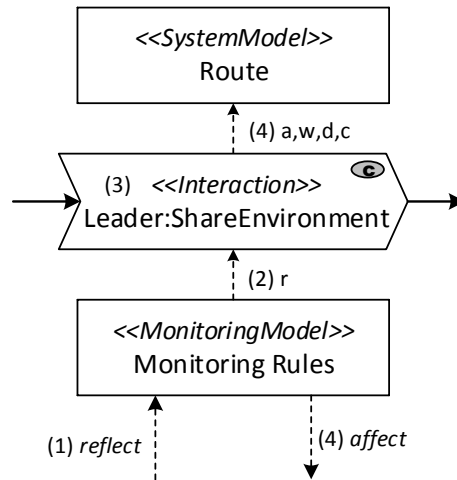


Figure 7.9: Deurema interaction

7.3. Simulator

The overall Deurema simulation workflow is depicted as activity diagram in Figure 7.10. At first, the modeled adaptive SoS in form of the Deurema models must be registered in the simulator to define the amount of adaptation logic and contained systems that have to be simulated. Additionally, the available domain knowledge, which is captured in runtime models, must be registered in the simulator. After the Deurema models and domain knowledge are known, the simulator preprocesses the given models. Thereby, it validates the Deurema models against syntactical correctness and deploys the available modules from the system template

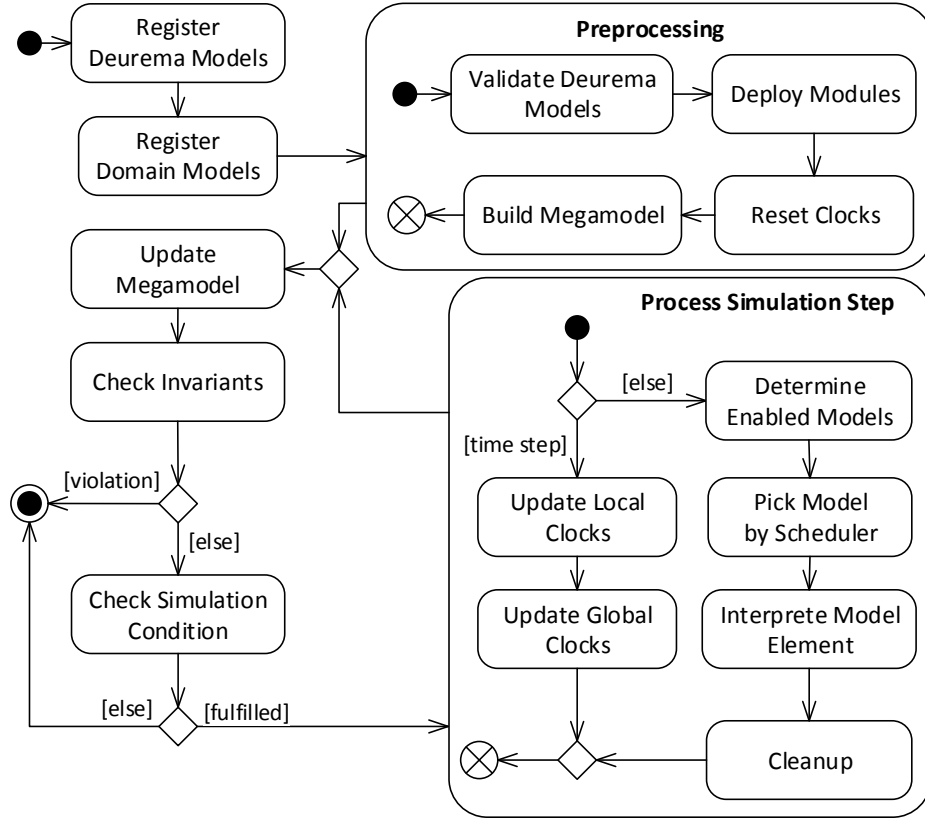


Figure 7.10: Deurema simulation workflow

architecture in the inference engine. Furthermore, all global and local clocks are reset and the initial build of the megamodel is started. The megamodel contains all Deurema models, runtime models as well as states in form of annotations from the inference engine during the simulation. The inference engine together with the underlying megamodel monitors and keeps track of every change in the runtime models and the specified Deurema architecture during the simulation, which is caused by the execution of the adaptation effects.

After the preprocessing, the main simulation loop starts, indicated by the `UpdateMegamodel` activity in Figure 7.10. During the update of the megamodel, the inference engine applies the defined simulation rules on the registered Deurema and runtime models. Thereby, it creates state information for each visible module and adaptation effect. Because it is possible to run analysis rules along with the simulation, the simulator checks if all specified invariants (runtime analysis rules) hold on the updated megamodel. If an invariant is violated, the simulation is aborted. The last simulation state is maintained and can be used for further offline analysis, e.g., for investigating the occurred, erroneous situation. If all domain specific invariants are fulfilled, the simulator checks defined simulation conditions. Simulation conditions indicate when a simulation should stop. Examples for simulation conditions are a fix number of simulation steps, when a global simulation time is reached, or if a specific situation occurs (e.g., a concrete collaboration instance is active). Checking invariants and simulation conditions belongs to the decision if the overall simulation process should be stopped or not. If the simulation is not stopped, one simulation step is processed (cf. activity on the right in Figure 7.10).

One simulation step is distinguished in a time step and an adaptation step. A time step comprises the update of local clocks for each module instance as well as the update of the global clock. The simulator performs a time step, whenever no adaptation step can be performed. Updating the local and global clocks is equal to a waiting step, where time elapses. This leads to new enabled modules and adaptation effects after the specified waiting period is reached. In contrast, in an adaptation step, the simulator determines all enabled Deurema model elements by looking at the state annotations. Afterwards, the simulator picks one of the enabled elements according to a defined scheduling strategy and hands it over to the Deurema interpreter for execution. Finally, the simulator cleans up intermediate data structures and ends the simulation step. Therefore, the megamodel is updated again and the next simulation round is processed.

Time Step Semantic

There are different possibilities for the simulator realizing a time step. If the developer wants to investigate the overall interplay between systems within the adaptive SoS to ensure that the modeled interaction works as expected, a *zero execution time* semantic can be applied very early during the development process, where the exact execution times of the modeled elements are not known or less important. A zero execution time simulation denotes that the time of executing an adaptation effect is not considered. Therefore, the simulator maintains a logical time for each deployed module. During the execution of a module, e.g., a feedback loop, no time elapses. If the execution of adaptation effects stops, e.g., because there is a time trigger specified that denotes a waiting period, the simulator increases the local or if necessary global logical time, updates the megamodel, and continues the execution of new enabled adaptation effects. Therefore, the zero execution time simulation focuses on the general interplay of adaptation effects, whereas the order is determined by the chosen scheduling strategy. On the one hand, this simulation strategy can be helpful in the early design process by investigating the interplay of local and collaborative adaptation activities. On the other hand, it is unrealistic with respect to the timing domain by ignoring the influence of elapsing time during the execution of an adaptation effect.

The simulator supports two further strategies by measuring the execution time or taking the specified execution time. The former strategy can be applied if the timing behavior is important, but the concrete execution times are not known by the developer. Therefore, the simulator measures how long the execution of each adaptation effect takes and updates the local and global clocks accordingly. Of course, the measured execution time depends on the underlying hardware capabilities of the system. Executing the simulation on a multi-core desktop computer will be much faster than the execution of the simulation on an embedded device. Avoiding these different execution times leads to the real-time simulation capabilities of the Deurema simulator. For this simulation strategy, the execution times must be specified by the developer along with the definition of the adaptation effects. This implies that the execution times are known or can be estimated upfront, which is a realistic assumption for embedded real-time systems but may not be predictable for long term analysis as well as planning activities.

Scheduling

Beside timing aspects, different scheduling strategies are thinkable for simulating the adaptive SoS. Caused by the independence of system and module instances, multiple adaptation effects can be enabled for execution at the same point in time. Furthermore, picking one enabled model element and execute it, may affect other model elements, which can lead to

further enabled as well as disabled models. Thus, the scheduling strategy has an effect on the overall simulated SoS behavior. The simplest policy is a random based scheduling, where the simulator randomly picks one enabled Deurema model and hands it over to the interpreter.

More complex policies are for example a *FIFO* or *LIFO* scheduling, where the simulator maintains an additional order of incoming enabled Deurema models. Thereby, a FIFO scheduling realizes a kind of fairness criterion, where model elements that are enabled before other model elements are executed before those. In contrast, a LIFO strategy is related to a most recently enabled adaptation effect scheduling policy, where new appearing, possible system adaptation and interaction effects are preferred.

Beside the discussed three scheduling policies, further strategies such as collaborative behavior before local adaptation behavior, adaptive behavior on lower system layers before behavior on higher layers, or temporally urgent adaptation effects first are thinkable. Additionally, tracing of the simulation execution trace can be attractive. If for example an error is found during a simulation, a reexecution of the logged trace can help verifying whether the error is fixed in the next system version or not. However, each scheduling strategy introduces an additional overhead to determine the next element for execution. Furthermore, the scheduling strategy mostly influences the observed adaptive behavior of the SoS, which might be completely different by exchanging the underlying scheduling algorithm. The Deurema simulator can be extended by realizing a scheduling strategy that fits best to the aimed simulation progress. As explained above, the scheduling algorithm is always applied on the beforehand retrieved enabled Deurema models and belongs to an adaptation simulation step (cf. pick model by scheduler activity in Figure 7.10).

7.4. Simulation Run Example

In the following, an exemplary simulation run is described that summarizes the discussion about the Deurema simulation framework above. According to the running example of this thesis, two self-configuring feedback loops sc_1 and sc_2 that interact with each other over the platoon p collaboration are considered. Pinpointing to the individual simulation steps in combination to an overview of the collaboration situation between the two feedback loops, a mixture of a Deurema LD and FLD is chosen as depicted in Figure 7.11. Thereby, the simulation steps are annotated on the corresponding elements.

The focus of the simulation run is on the feedback loop sc_1 and its played `SharedEnvironment` interaction in the leader role of the platoon collaboration. As precondition of the module execution, the event trigger e_1 from type `ModeSwitch` must occurred at least once and the time trigger with a period of 30 seconds must be elapsed, which is denoted with step (1) in Figure 7.11. The inference engine detects the enabled sc_1 module and marks it with a corresponding annotation. During a simulation step, the simulator selects the enabled module sc_1 and sends it to the Deurema interpreter for execution. As discussed above, the interpreter marks the module as active (step 2 in the simulation run) with the effect that all contained feedback loop operations become visible for the inference engine. The inference engine detects the enabled initial node of the feedback loop, which is again sent from the simulator to the interpreter for execution. Detecting enabled model elements and picking those by the simulator is done for all following Deurema model elements of this simulation run and thus not mentioned for the rest of the example discussion. The interpreter marks the initial node as active denoted by step (3), which enables the `Update` activity. Because this activity is a behavior model element, it defines a domain specific adaptation effect. For its execution, which is subsumed by

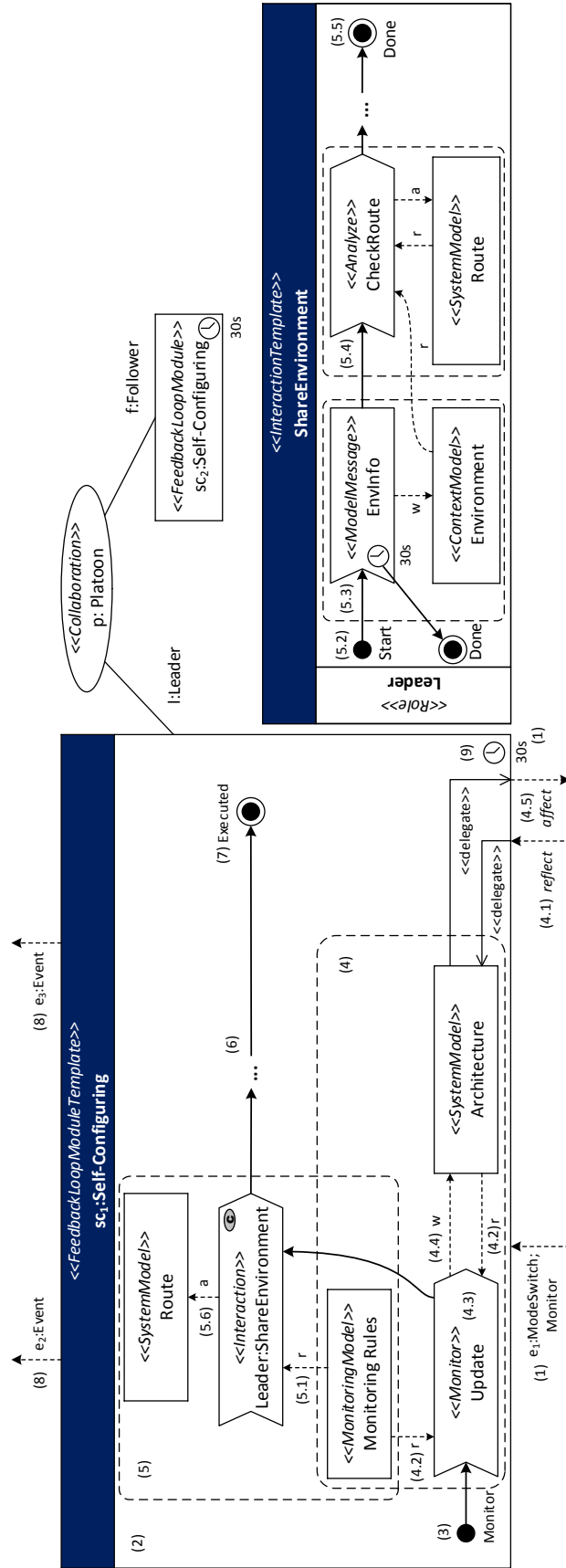


Figure 7.11: Deurema simulation run example

step (4), the interpreter follows several steps. It updates all accessed runtime models, whereas the Deurema reflection module operations are evaluated first (4.1) and the amount of runtime data is retrieved over the read model operations afterwards in step (4.2). In the example, there is one reflection dependency, whereas the information is delegated to the Architecture runtime model. Furthermore, there are two read model operations from the Monitoring Rules and Architecture runtime model to the Update activity. Once all information is retrieved, the data is handed over to the activity and the adaptation effect is executed (4.3). Depending on the black-gray-white box specification of the Update activity, the execution can comprise several steps as comprehensively discussed above and in Section 5.3.2. Changes on the runtime model by the Update activity are recognized by the Deurema execution environment and appropriately applied to the corresponding runtime model denoted by step (4.4). Furthermore, changes that influence an underlying module or collaboration are propagated accordingly via the affect module operation in step (4.5). Afterwards, denoted by the control flow, the ShareEnvironment interaction becomes enabled and thus, is executed as step (5) by the interpreter. According to the collaboration knowledge specification, all incoming runtime models are retrieved (5.1) and are now available for the interaction. The details of the interaction for the leader role are shown on the right in Figure 7.11. As soon as the interaction becomes active, which happens by executing the initial node (5.2), the corresponding interaction player module sc_1 changes its state to collaborating. In this example, the collaboration consists of exchanging data via a model message, which is stored in the Environment runtime model (5.3) and an internal CheckRoute analyze activity within the leader role (5.4). Executing the analyze activity follows the same interpretation sequence as described for the execution of the Update activity in step (4). After the final node of the interaction is reached and executed (5.5), the updated Route runtime model is provided to the local context of the feedback loop sc_1 (5.6). The interaction is successfully completed and changes its state to disabled. Furthermore, the module instance sc_1 becomes active again and all other operations of the feedback loop can be subsequently executed (step (6) and (7)). If the execution reaches and processes the final node (7), the corresponding feedback loop module becomes postactive and finally disabled. The outgoing module events e_2 and e_3 are emitted by the simulator in step (8) and the timed trigger is reset denoted by step (9) so that the module must wait the defined period before it can be executed again.

There are some implications of this exemplary simulation run. First, the other module instance sc_2 must send the environmental data in step (5.3) within a time slot of thirty seconds. Otherwise, the interaction is aborted over the final node, which can be reached from the timed trigger of the corresponding model message. Second, all operations of the second feedback loop instance sc_2 run completely independent from the operations in sc_1 . In this example, both modules synchronize their behavior only by exchanging the model message. Therefore, each module and interaction has a corresponding template definition, whereas the Deurema execution environment maintains individual states for each module and interaction instance.

7.5. Runtime Analysis

In contrast to the static analysis of the modeled adaptive SoS architecture as outlined in the former Chapter 6, the Deurema simulation framework facilitates the coexistence of analysis rules and the simulated SoS, which enables a runtime analysis of the adaptive behavior. Figure 7.12 sketches two possibilities of applying a runtime analysis during a Deurema SoS simulation. At first, the same static analysis rules can be applied during the simulation to

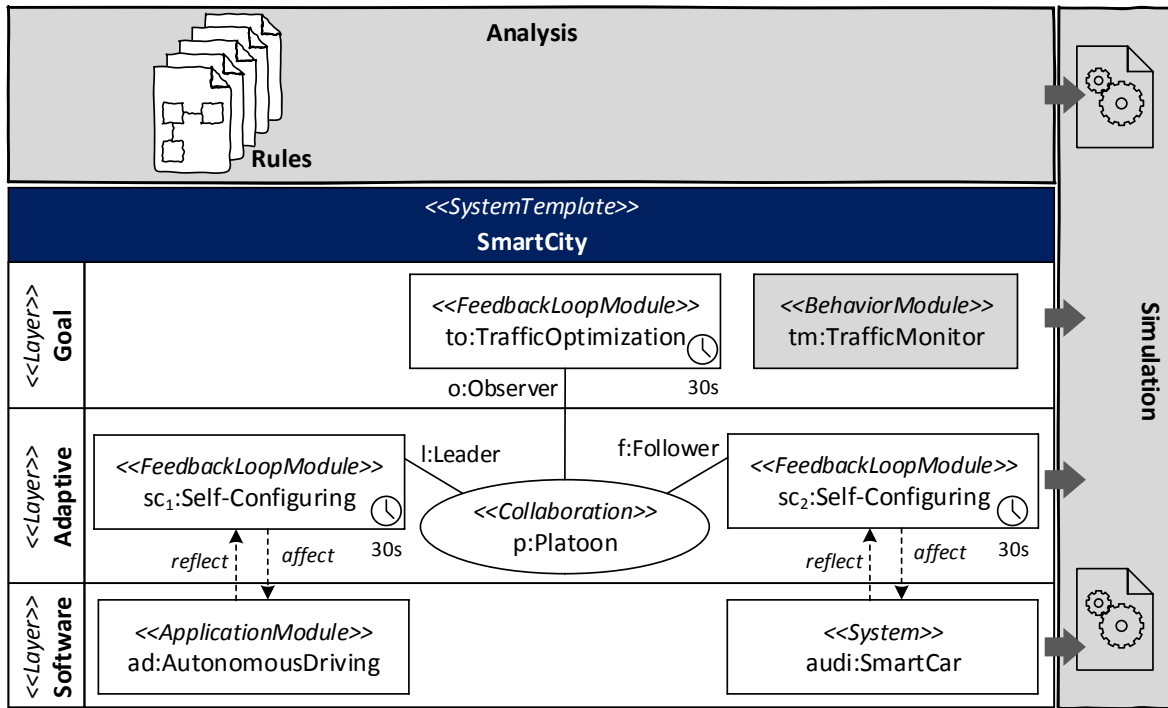


Figure 7.12: Smart city running example: Deurema runtime analysis

reason about different metrics within the Deurema models and thus, of the adaptive SoS architecture during runtime. In this case, the analysis rules are separated from the modeled adaptive SoS and both, the analysis rules as well as the Deurema models are hand over to the simulation framework. In the second possibility, the analysis rules are modeled within the adaptive SoS by using the Deurema rule-based behavior module concept. In this variant, the analysis rules are encoded into a behavior module template, whereas a corresponding instance is placed on the layered system template definition of the SoS. The second case is sketched by the TrafficMonitor module on the goal layer in the SmartCity system template as highlighted in gray in Figure 7.12. The focus of both runtime analysis possibilities is different. The separation of analysis rules and the modeled adaptive SoS targets the analysis of the Deurema models and thus, retrieves dependencies as well as patterns between modules, systems, and collaborations. The integration of analysis rules within the SoS in form of module instances enables the reasoning about the domain specific behavior. For example, the traffic monitor can reason about existing platoon collaborations in the smart city, which clearly belongs to a domain specific analysis concern instead of investigating causal dependencies between Deurema modules as done for the first variant. Furthermore, the integrated module containing the analysis rules can use the Deurema reflection and adaptation concepts to define the overall amount of local knowledge that is used to apply the modeled analysis rules. In this case, the dependencies of the runtime analysis are explicitly defined in the Deurema model and thus, become visible for the system developer. The decision of modeling analysis rules separately or within the adaptive SoS belongs to the system developer and the aimed analysis metric (domain specific or Deurema model specific). However, the Deurema execution environment supports both variants.

With respect to the separated analysis variant, analysis rules are applied on the concrete instance situation of the running system during the simulation, which enables the investigation of concrete runtime effects instead of statically analyzing all possible effects of the system. For example, there are two possible adaptation paths in the feedback loop in Figure 6.18. Therefore, the static analysis retrieves both paths together with all causal dependencies as well as their closures as discussed in Section 6.1.1. At runtime, only one path can be executed, which is for example *Activity*₁ followed by *Activity*₂ in the upper path in Figure 6.18. This narrows the overall possible causal dependencies and their corresponding causal closures down to the both activities in the upper path, where the both activities at the bottom cannot be causally happen in this concrete situation. As a consequence, annotations for all the different analysis metrics pinpoint to the concrete simulated situation of the running SoS by considering the currently executed entities and their current states in the system instead of enumerating all possible dependencies as done in the static analysis. Of course, the static analysis rules must be slightly adapted to detect those entities in the overall adaptive SoS that are currently running (e.g., *Activity*₁ in the example above) and those, which are not able to run (e.g., *Activity*₃ and *Activity*₄) by detecting the annotated state information as explained above. Thereby, the analysis results change during the execution of the adaptive behavior. For example, feedback loops usually run periodically. If the feedback loop in Figure 6.18 is executed again, it might be possible that the lower path is enabled in the subsequent execution run instead of the upper path. Thus, causal dependencies arise as well as disappear during the simulation by applying runtime analysis. Another example for a difference between static and runtime analysis is the concrete effects of design flaws. Consider Figure 6.18 again, the static analysis retrieves an annotation for a design flaw, whereas the overall causality contradicts the knowledge dependencies in the feedback loop as discussed in Section 6.3. At runtime, it might happen that the lower path of the feedback loop is never executed and thus, the potential static violation of the causality and knowledge is never detected at lifetime of the adaptive SoS.

The integrated runtime analysis variant supports the monitoring of concrete situations in the corresponding domain. Instead of analyzing the modeled Deurema architecture, the runtime analysis can focus on the available information in the local runtime models. Furthermore, the local information can be enriched by reflecting other modules, collaborations, and subsystems in the system template. This enables the runtime analysis with respect to the underlying problem domain. Transferred to the smart city running example of this thesis, an analysis rule can be responsible for detecting a car accident within a platoon, where both smart cars run in an autonomous driving mode. Of course, such an accident should never happen, otherwise the autonomous driving functionality of the smart car seems to be erroneous. Therefore, runtime analysis rules can be used to detect forbidden or special situations in the concrete domain looking at the runtime models of the system.

Penalties of applying runtime analysis are the additional overhead of evaluating the analysis rules along with the system execution. Usually, this overhead can be handled in a simulation of the adaptation logic but exits the resources in the concrete system realization. Especially small embedded systems within the overall SoS have limited computation and memory resources, where an additional runtime analysis cannot be realized. Furthermore, the analysis rules must be able to monitor the execution states to reason about the current simulation situation. Because the Deurema analysis framework annotates a found situation directly into the Deurema model, the developer must decide about the consequences if for example a

forbidden situation was found in the system. This can for example trigger additional offline development activities such as bug fixing or redesigning parts of the layered system structure.

7.6. Discussion

The basic design decision of independently running systems and modules during a simulation leads to the separation of responsibilities of the inference engine, which maintains the overall state situation, the simulator, which controls the simulation run, and the interpreter, which executes a single Deurema model element. Thereby, each Deurema element follows a predefined state model during the simulation. The starting point of a simulation is a modeled SoS architecture in form of a Deurema system template, which evolves during the simulation. In general, an inference engine detects enabled Deurema elements with the help of simulation rules. The simulation rules are enumerated in the Appendix D. For the execution of the simulation rules, the inference engine implementation from [35] is used. On basis of the retrieved enabled elements, the Deurema simulator picks one element and hands it over to the Deurema interpreter for execution. Thereby, the simulator supports different scheduling strategies deciding which element is picked next for its execution. The Deurema interpreter realizes the execution and invokes the corresponding adaptation effects, which may lead to changes in the initial deployed system template specification, e. g., a feedback loop may change the behavior of an underlying module instance. As a consequence, the inference engine detects changes in the Deurema runtime megamodel, which comprises the state annotations, the runtime models, and the Deurema system model itself, and reevaluates depending simulations rules. The reevaluation leads to new enabled elements, which are again picked by the simulator.

The simulation workflow described in Figure 7.10 and the interpreter are completely implemented in Java. Thereby, the Deurema metamodel is realized by the Eclipse Modeling Framework. On basis of the metamodel, simulation rules are defined to detect the current state situation for all deployed module and interaction instances within the adaptive SoS. Execution metrics of a Deurema simulation for the running example are enumerated in the Appendix D.

Finally, the analysis rules discussed in Chapter 6 are also realized as declarative rules with the inference engine tool. Therefore, these rules can be executed side by side with the system simulation, which realizes the runtime analysis capabilities of the Deurema execution framework. During the simulation, found analysis metrics are directly annotated in the Deurema models. Thereby, the Deurema execution environment facilitates the direct simulation of Deurema models and thus, no implementation artifacts as for example Java or C++ code are derived from the models for its execution. Deriving a concrete implementation belongs to the realization of a modeled adaptive SoS, which is discussed in the next chapter.

8. Realization

This chapter discusses the realization of Deurema modeling concepts with focus on the embedded, automotive domain. Whereas the Deurema analysis and simulation framework helps verifying the behavior of the modeled adaptive SoS, this chapter is concerned with an exemplary mapping of Deurema concepts to a concrete application domain showing that the modeling language concepts can be realized in current systems. Furthermore, realizing the modeled adaptive SoS enables analysis and simulation of the system within the real target platform, which introduces additional, domain specific execution effects (e. g., sensor noise, realistic timing behavior) that cannot be investigated in a *virtual* simulation as discussed in Chapter 7. In general, the Deurema concepts can be realized differently depending on the underlying execution platform, domain, or preferences of the software developer. Therefore, this chapter pinpoints to one possible realization using the AUTOSAR de facto standard from the automotive, embedded domain and does not claim that this realization is required nor the only possible implementation of the modeling concepts. The AUTOSAR standard is used, because it is widely adopted for the development of modern cars. Furthermore, smart cars are complex cyber-physical systems. On the one hand, they include different adaptive capabilities and may collaborate with other smart cars to optimized the local behavior. On the other hand, cars have high demands on safety, including hard timing constraints, and must be efficiently realized with respect to a resource restricted execution environment.

The main idea of realizing Deurema models is sketched in Figure 8.1. The starting point for a concrete implementation is a system template description, which comprises modeled modules, templates, and collaborations. Although a complete implementation of the system template is highly desirable so that the resulting realization is as close as possible to the modeled SoS, this chapter describes a mapping for each Deurema template type to the AUTOSAR standard. This facilitates a partial step-by-step realization of the modeled SoS architecture, which further enables a refinement of domain specific modifications such as a concrete distribution to physical available computing resources.

This chapter is structured as follows: Section 8.1 introduces the modules and Deurema concepts of the smart car running example, which are used to explain the mapping to the AUTOSAR standard. Afterwards, Section 8.2 explains all necessary concepts from the AUTOSAR standard that are needed for the realization discussion. Based on the introduced excerpt of the running example and the AUTOSAR concepts, the mapping of each Deurema module template type is discussed, starting with software module templates in Section 8.4, application module templates in Section 8.5, feedback loop module templates in Section 8.6 to behavior module templates in Section 8.7. Within the module template type realization discussion, the mapping of Deurema cross cutting concepts such as collaboration and the reflection mechanism are highlighted. This chapter concludes in Section 8.8 by pinpointing to the used software tools and an exemplary modeling process, which comprises analysis, simulation and realization steps discussed in this thesis.

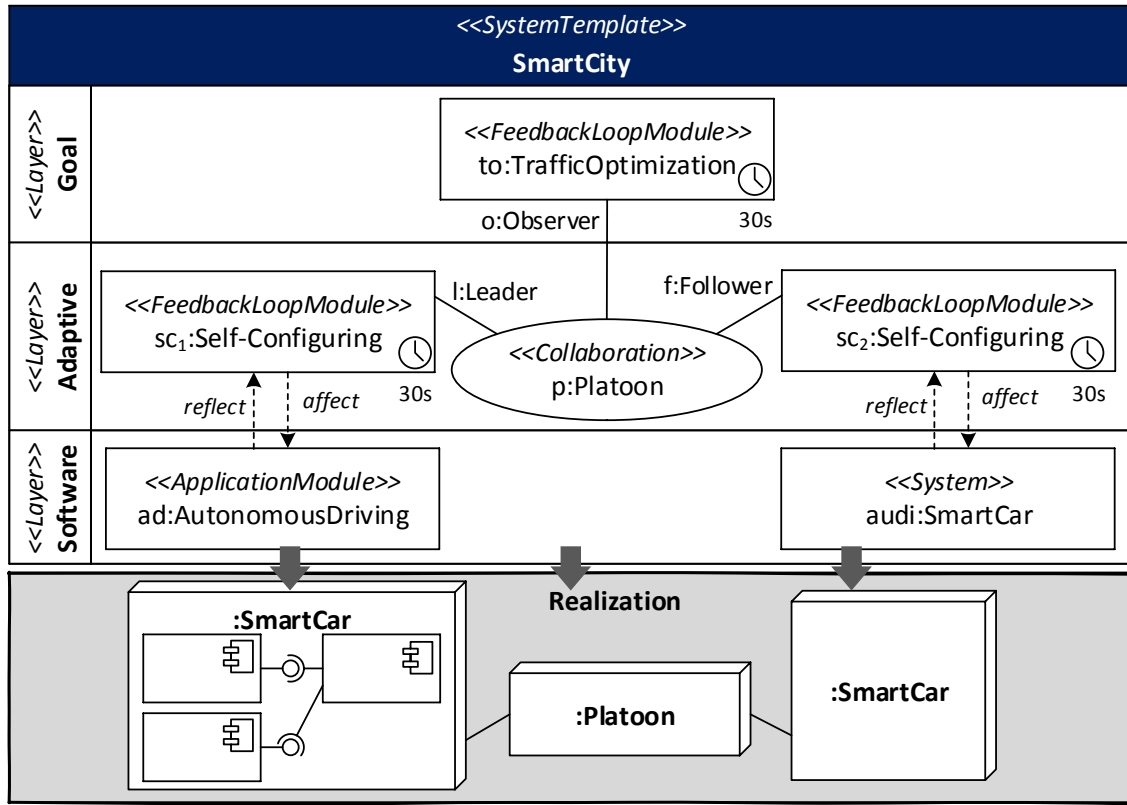


Figure 8.1: Smart city running example: Deurema realization

8.1. Scope

A comprehensive discussion about Deurema concepts along with the running example of this thesis is given in Section 5.7. This chapter focuses on dedicated modules of the SmartCar example as highlighted gray in Figure 8.2. The selected modules comprise all supported Deurema template types, the Deurema reflection capabilities and the collaboration modeling concept. At the lowest layer L-0, the Motor software module encapsulates the control functionality of the car engine. This module is reflected by the AutonomousDriving module at layer L-1 and the ESP module at layer L-2. The former follows the component-based specification of the behavior and realizes autonomous driving functionalities. The latter module uses the rule-based approach and implements a supervising traction control behavior of the smart car. An additional ABS module is modeled as feedback loop that supervises the wheels of the car and ensures an antilock braking functionality. The ALC module runs in parallel to the other modules and implements an adaptive light control functionality, which turns up and down the headlights of the smart car depending on oncoming vehicles. A black board collaboration connects the independent modules ESP, ALC and AutonomousDriving and enables the data exchange. The autonomous driving module is the head of the collaboration and the other both modules contribute important data.

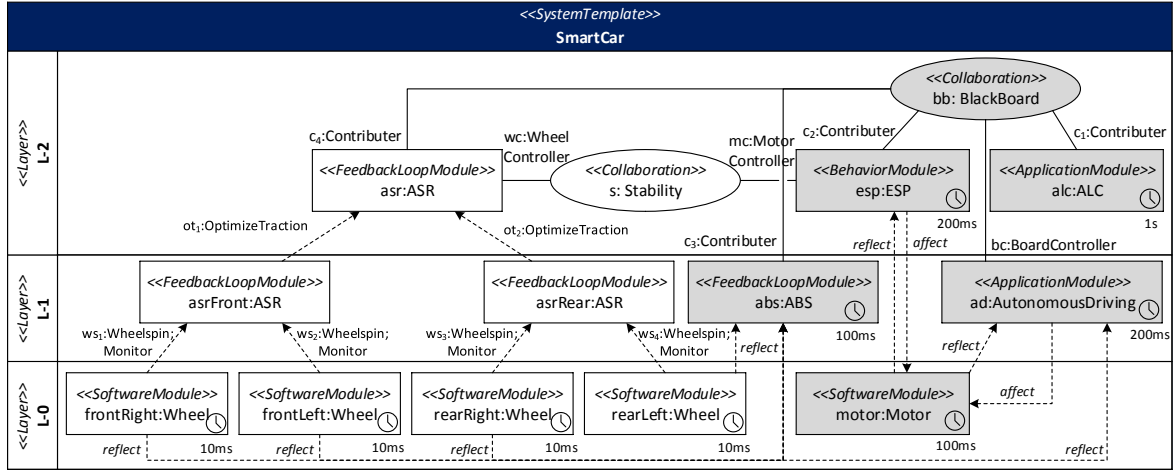


Figure 8.2: Highlighted Deurema modules that are discussed concerning a realization

8.2. AUTOSAR

The Automotive Open System Architecture (AUTOSAR) is the de facto standard in the automotive domain¹ for the development of complex, distributed systems such as modern cars [62]. The standard defines a layered reference architecture, provides standardized communication mechanisms and a complete development methodology. This facilitates the cooperation between different car manufactures and their suppliers. Figure 8.3 gives an overview of the layered AUTOSAR architecture, which is based on the standard specification in [61]. The lowest layer at the bottom encapsulates the concrete hardware including the access to the underlying microcontrollers called Electronic Control Units (ECU) and communication buses in the car. On top, a basic software layer provides standardized interfaces to the hardware layer below as well as it realizes basic functionality of an operating system, which includes task scheduling as well as access to operating system resources such as memory. The AUTOSAR runtime environment is responsible to realize the concrete communication from and to the software layer on top. The three lowest layers are completely standardized by AUTOSAR. Due to resource restrictions in automotive systems, the runtime environment and basic software can be efficiently generated considering only the communication mechanisms and the operating system functionality needed by the software abstraction layer above as well as the available hardware resources at the lowest layer.

The highest layer contains the application logic, where the AUTOSAR architecture style changes from a layered to a component-based development approach [61, 62]. AUTOSAR provides concepts to specify the domain specific functionality by a set of AUTOSAR software components, which can communicate with each other over well-defined ports using AUTOSAR interfaces. Thereby, the modeled communication is realized by the AUTOSAR runtime environment layer. Each AUTOSAR component consists of behavioral entities that are named *Runnables*, which encapsulate an individual piece of functionality. Usually, runnables are implemented by means of C/C++ functions. Furthermore, they can read and write data from the available ports of the corresponding component. For execution, runnables are mapped to operating system tasks, which are scheduled by the operating system functionality included in

¹www.autosar.org

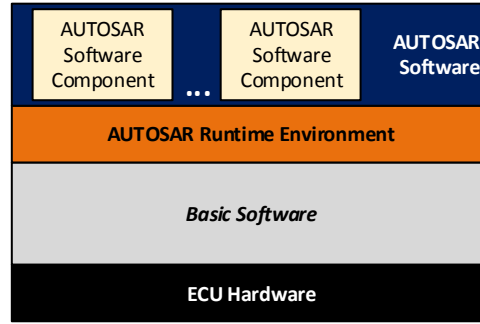


Figure 8.3: Layered AUTOSAR architecture [61]

the basic software layer. AUTOSAR defines different port concepts, where the developer can choose an appropriate type according to the underlying problem. For example, a component with a client-server port can provide (server) a specific service for other components (client). Thus, a client component can invoke a possible distributed service, where the computational effort is transferred to the server component. Another example are sender-receiver interfaces that are designed for the data exchange between components. The direction of the data flow is defined from the sender component to the receiver. A comprehensive discussion of applying the AUTOSAR standard for the development of embedded robotic systems is given in own former work in [14].

Figure 8.4 summarizes the AUTOSAR elements, which are necessary for the mapping of the Deurema concepts discussed in this chapter. In the following, each element is briefly introduced, where a comprehensive definition can be found in the standard specification in [62].

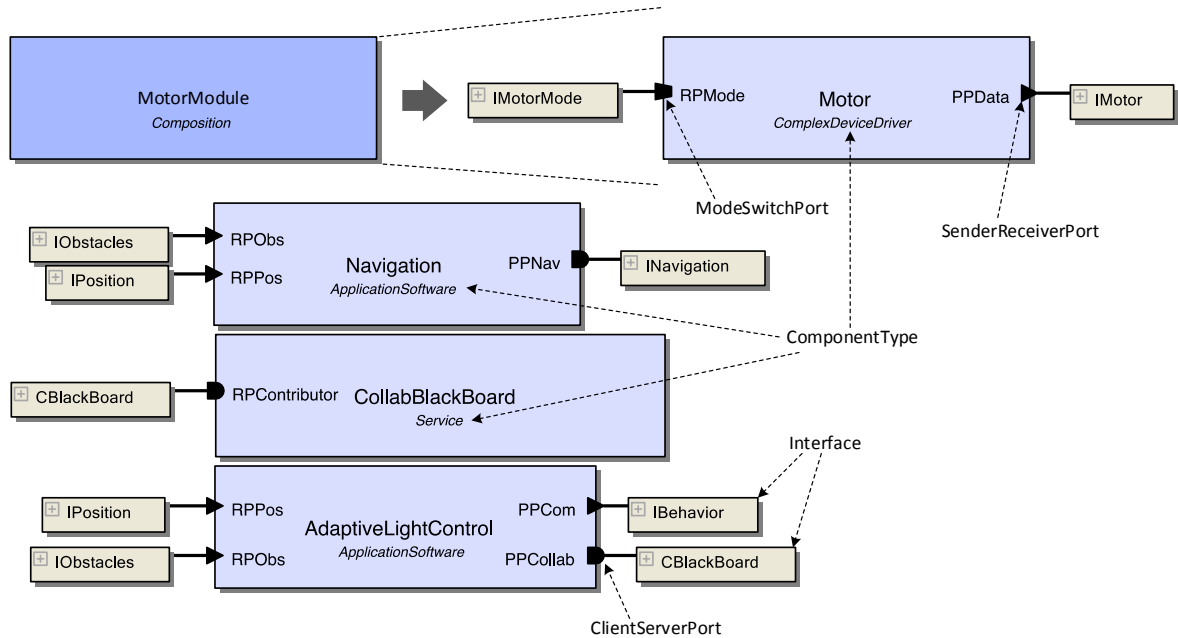


Figure 8.4: AUTOSAR components, ports, and interfaces

Composition

A *composition* is a structuring software component type that has no direct functionality specified. It contains other subcomponents such as compositions and software components. Therefore, compositions can be used for grouping components and building a hierarchical software architecture. Figure 8.4 shows the concrete syntax of an AUTOSAR composition for the MotorModule, which is labeled with the tag `Composition`.

Component

An AUTOSAR *atomic software component* (also application software component, or short component) is the basic entity for the specification of the overall architecture that defines the application software functionality (cf. Navigation component in Figure 8.4). A component contains a behavior description, which can be separated in different internal functional units called *Runnables*. In contrast to a composition, a component must be deployed on an Electronic Control Unit (ECU), which is the execution environment of the defined software specification. Components can communicate with each other as well as exchange data over ports, whereas the type of the communication is defined by the corresponding interface.

A *service component* is a special software component that provides an accessible service functionality, which can be invoked over an appropriate port (cf. CollabBlackBoard component in Figure 8.4).

A *complex device driver* is a special software component, which encapsulates non AUTOSAR specific domain functionality (e.g., device specific control commands as done for the Motor component in Figure 8.4). Encapsulated, domain specific functionality in a complex device driver component can be accessed by other AUTOSAR components and thus, can be integrated in the overall software architecture.

Port

Ports are connection points of a component that enable communication and data exchange using an appropriate communication type defined by an interface description. The port type must correspond to the interface type, which is described below. The communication flow is defined from the provided port to the requested port. Furthermore, compositions may contain ports, whereas the communication flow is forwarded by the composition to an inner component.

Interface

Interfaces define the protocol and content of the communication between components and are attached to a port. Only ports between two components with a compatible interface description can be connected with each other. AUTOSAR provides, among others, different interface types, which are *sender-receiver*, *client-server*, and *mode switch* interfaces. The concrete syntax for each interface type and corresponding port is depicted in Figure 8.4.

A sender-receiver communication is unidirectional, where the sender distributes data to one or more receivers. The receiver asynchronously processes the data but does not respond to the sender.

A client-server interface enables service-oriented, bidirectional communication that can be synchronous or asynchronous. The server provides a certain functionality, which can be invoked by the client. The computation is executed on the server side, where data can be provided by the client and the computation result can be returned from the server.

A mode switch interface can be used to inform the software component about changing its mode. Modes must be predefined in mode declaration groups. A mode switch enables or disables a certain functionality of the corresponding ECU.

Runnable

Runnables are the smallest entities inside a component and define the internal behavior. They can be compared with a C function or Java method. Runnables are assigned to tasks and are scheduled by the operating system for their execution. Furthermore, runnables can read/write data from/to ports as well as share data with other runnables within one software component via interrunnable variables. Therefore, once a specific piece of information is read over a port, it is available for all runnables in the parent software component.

Task

A task references a list of runnables, which are subsequently executed by the operating systems. Tasks run concurrently with other tasks and are controlled by the operating system scheduler. Thus, components define the structural architecture of the software, where tasks are behavioral entities executed on the underlying ECU.

Having a common understanding about the AUTOSAR concepts, the mapping of each module template type as well as the collaboration aspects highlighted in Figure 8.2 are discussed in the following.

8.3. Systems and Modules

In general, the AUTOSAR standard follows the component-based development approach for the specification of the software functionality. Therefore, all Deurema concepts must be translated to the component-based AUTOSAR methodology. From a high to a low granularity level, a first step is the mapping of Deurema systems. In Deurema, system templates can contain multiple module instances as well as subsystems, which specify the overall adaptive SoS architecture. An AUTOSAR composition can contain other compositions and components, which defines a hierarchical structure of the software architecture. Therefore, Deurema system instances are mapped to AUTOSAR compositions. Furthermore, AUTOSAR distinguishes between the structural specification of a composition and its instantiation (deployment). This is similar to the Deurema system template as well as module template and their corresponding instances placed on the layered SoS architecture, which enables a one-to-one mapping of Deurema templates and instances to AUTOSAR composition descriptions and their deployment specification. With respect to the example in Figure 8.2, the SmartCar system template is realized by an AUTOSAR composition as shown in Figure 8.5. The figure sketches the hierarchy of AUTOSAR compositions. At the highest level, the smart car system template is realized as AUTOSAR composition and contains all other compositions.

Unfortunately, AUTOSAR does not support the layering of a software architecture as first class concept. In contrast, Deurema system templates can have arbitrary layers. Therefore, each layer is emulated in AUTOSAR by encapsulating the complete content of a layer (the placed modules and subsystems) in a composition, which results in a hierarchical AUTOSAR software architecture. According to the example, there are three compositions in Figure 8.5 encapsulating the three layers of the SmartCar system template.

Finally, Deurema modules are the smallest entities that can be placed on a layer in the system template description. Because modules encapsulate the adaptive behavior in corresponding template definitions, each Deurema module is mapped to an AUTOSAR composition as well. Depending on the template type, the module composition can be refined by an AUTOSAR component architecture that follows the template specification. Consequently, each Deurema

module from Figure 8.2 is realized by an AUTOSAR composition as shown in the mapped layer compositions in Figure 8.5.

In summary, Deurema system, layer, and module elements that are specified in the system template definition are mapped to AUTOSAR compositions accordingly. Thereby, a system composition contains layer compositions, which represent the layer from the Deurema system template description. Again, each layer composition contains appropriate module as well as subsystem compositions. Subsystems are unfolded following the same strategy of mapping system layers and contained subsystems until there are only modules left at the lowest hierarchy level. The module compositions are further refined by a component-based specification depending on the module template type as described in the following.

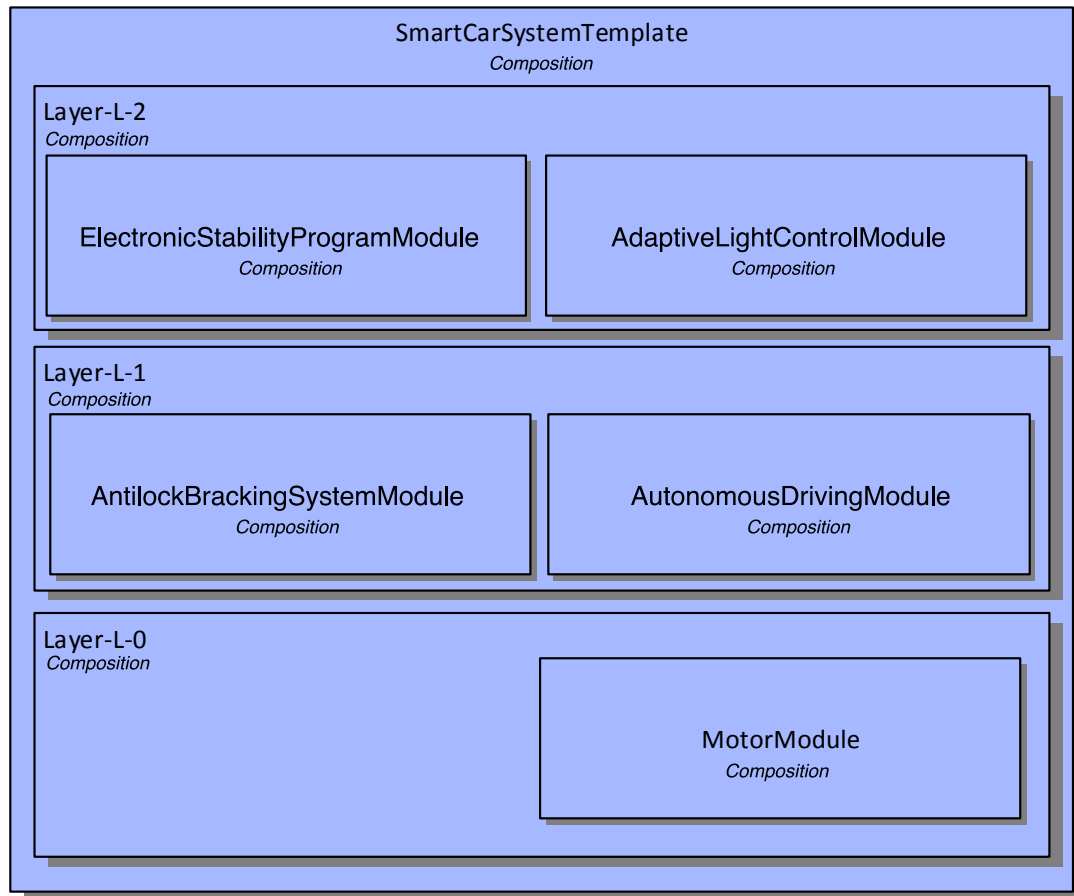


Figure 8.5: Deurema modules as AUTOSAR compositions

8.4. Software Module Template

At the top, Figure 8.4 shows the mapping of a Deurema software module (template) to an AUTOSAR component. Software modules encapsulate black box adaptation behavior and the internals are not known in Deurema. Therefore, the domain specific implementation can be mapped to a ComplexDeviceDriver component in AUTOSAR. Although the naming of this component might be confusing, a complex device driver component has similar characteristics to a software module. First, the intention of this AUTOSAR component type is to encapsulate

domain specific functionality by simultaneously hiding implementation details. This might comprise different levels of complexity by providing simple driver for devices up to complex wrapper to integrate company, non AUTOSAR, specific legacy software. Thus the concrete internals are not known and appear as black box component. Second, although the behavior is not modeled, the component type may have AUTOSAR ports and interfaces, which enables an integration of black box behavior into the overall AUTOSAR architecture as well as the interplay with other components. As a consequence, each Deurema software module turns into an AUTOSAR composition, which is further refined by one or more complex device driver components. Thus, the Motor software module from Figure 8.2 appears as corresponding component at the top of Figure 8.4.

Furthermore, the Deurema *reflect* module operation, which is performed by the Autonomous-Driving module, retrieves information about the Motor module. Because AUTOSAR does not support dynamic component reflection, the Deurema module operation is realized by a sender port with a corresponding sender-receiver interface. In general, AUTOSAR sender-receiver ports are used to exchange data between components, whereas the data types are defined by the interfaces that are attached to the ports. In Deurema, module operations are defined by one or more model queries that are performed on the corresponding module (cf. Section 5.3.2). Therefore, the module operation is mapped to a sender port of the reflected AUTOSAR component. Consequently, the Motor component in Figure 8.4 has a sender port named *PPData* and a corresponding sender-receiver interface named *IMotor*, which enables the retrieval (reflection) of internal information.

In contrast, the Deurema *affect* module operation may result in changing the underlying module behavior, which is realized by a mode switch port and corresponding interface in AUTOSAR. Mode switches can be used to change the behavior of an AUTOSAR component, whereas the concrete modes depend on the specified reconfiguration possibilities of the corresponding Deurema module. Thus, the Motor component has an incoming mode switch port *RPMode*, which enables the affecting access to this component from the outside.

It has to be noted that the AUTOSAR *MotorModule* composition encapsulates the Motor component. Therefore, another component cannot access the defined ports of the contained component directly. However, AUTOSAR allows the specification of ports on a composition and uses *delegation connections* to forward the communication from the composition to inner components. According to the hierarchical composition structure in Figure 8.5, such ports and corresponding delegation connections must be added for each parent composition. For the running example, the communication that realizes the reflection module operation between the AutonomousDriving and Motor module requires a sender port as well as delegation connections in the MotorModule and Layer-L-0 composition. Straight forward, it needs receiver ports as well as delegation connections in the Layer-L-1 and AutonomousDrivingModule composition (cf. Figure 8.5).

The internals of a Deurema software module are defined by the corresponding template specification. As discussed in Section 5.3.3, a Deurema software module template specification comprises an optional trigger and action definition, which refers to the domain specific implementation. Both entities are mapped to an AUTOSAR runnable as shown in Figure 8.6. These runnables are contained in the AUTOSAR component and must be mapped to an operating system task for their execution. Therefore, for each software module a corresponding task is created as exemplarily shown on the right in Figure 8.6. The task gets the same timed trigger as the software module, which is directly supported by the AUTOSAR standard. Because an operating system task must have a priority, which is not supported by a Deurema

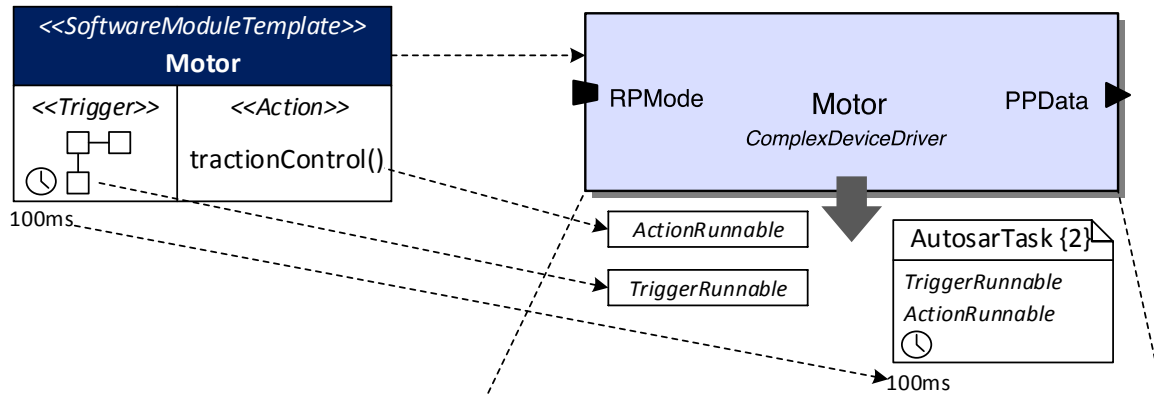


Figure 8.6: Deurema software module as AUTOSAR component

module specification, it is a design decision of the developer to define an appropriate priority for the task in AUTOSAR.

Beside the both runnables referring to the trigger and action definition of the software module, there must be at least one runnable that accesses the ports for reading and writing data. Conceptually, there are two possibilities of realizing this behavior. First, accessing the ports is realized within the beforehand mentioned trigger and/or action runnable. Second, additional runnables can be created, whereas the behavior of these runnables can be derived from the specified Deurema model operation queries. Afterwards, these runnables must be mapped to the AUTOSAR task for execution.

8.5. Application Module Template

Both, the AUTOSAR standard and Deurema application module templates follow the component-based development approach. Furthermore, the concepts of the application module template are developed with respect of an easy integration of the development paradigm in the embedded domain, where the AUTOSAR standard plays an important and inspiring role. Consequently, Deurema and AUTOSAR are very similar with respect to the component model, which eases the mapping between both. Figure 8.7 shows the internals of the AutonomousDriving and AdaptiveLightControl Deurema module template. The autonomous driving template contains nine components. Three sensor components retrieve data from the environment of the smart car, which are preprocessed and used for navigation afterwards. The Driving component reads the planned navigation path and realizes the autonomous driving by sending appropriate commands to the engine and wheel controller.

Beside the autonomous driving module, the adaptive light control module regulates the light intensity of the car depending on environmental conditions and oncoming vehicles. Therefore, the sensor data is aggregated by the SensorFusion component, evaluated by the LightCalculation component, and finally realized by the LightController actuator.

The modeled Deurema components in an application module template can be directly translated to AUTOSAR components. Thereby, Deurema sensor and actuator components are realized by AUTOSAR SensorActuatorSoftware components as shown for the autonomous driving module in Figure 8.8. Deurema software components are translated to AUTOSAR

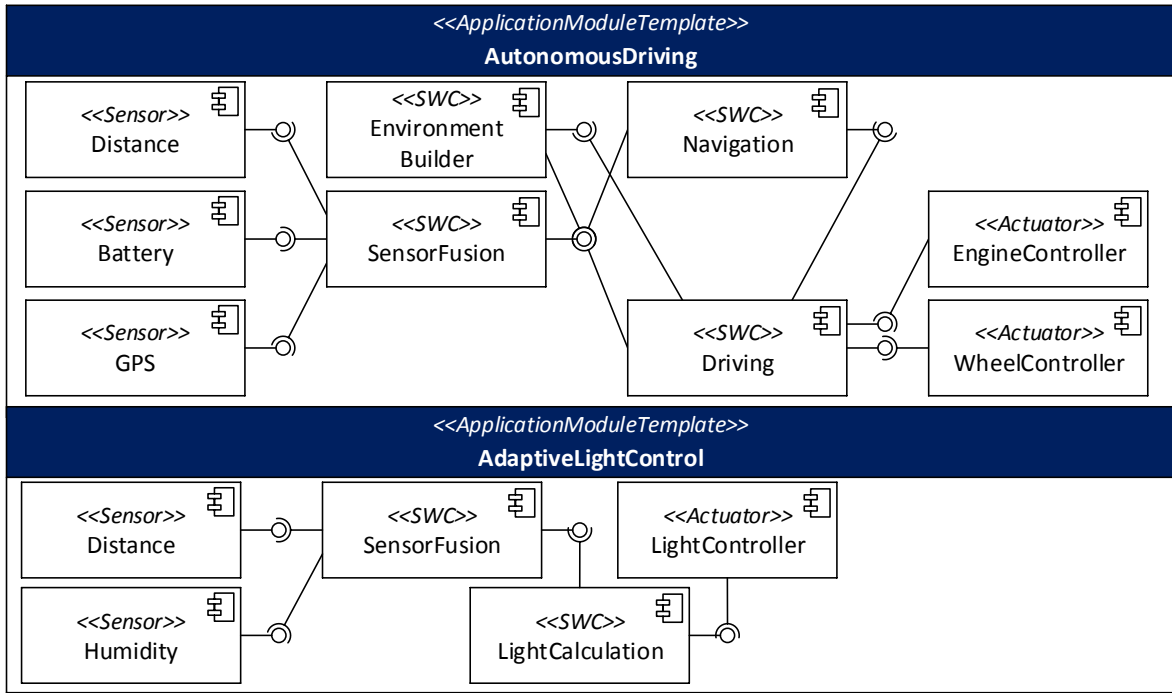


Figure 8.7: AutonomousDriving and AdaptiveLightControl Deurema module templates

ApplicationSoftware components respectively. As a consequence, each Deurema component in Figure 8.7 appears in the AUTOSAR component architecture realization in Figure 8.8 (for autonomous driving) and Figure 8.9 (for adaptive light control).

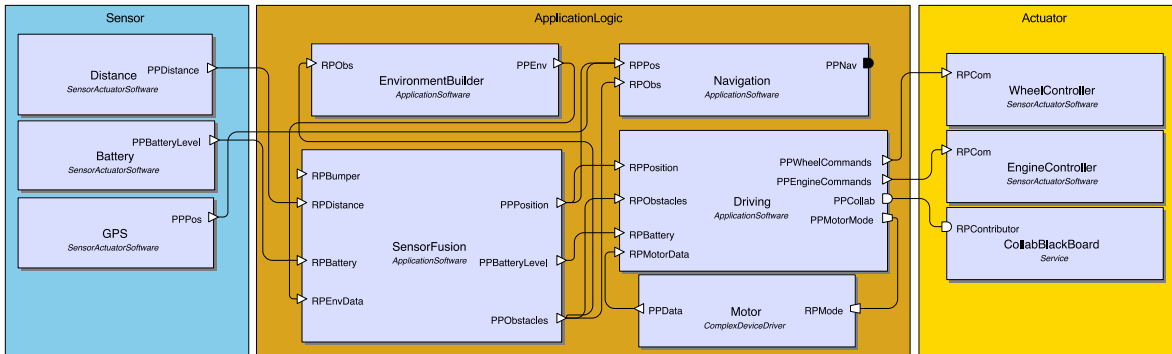


Figure 8.8: AutonomousDriving as AUTOSAR component architecture

Beside components, Deurema runnables can be mapped one-to-one to AUTOSAR runnables, whereas the access to knowledge is realized by appropriate AUTOSAR interfaces. Furthermore, AUTOSAR allows the fine-grain specification of data access for each modeled runnable according to the Deurema model. After realizing the AUTOSAR component architecture, runnables must be mapped to operating system tasks, which defines the behavioral aspect of the smart car. Thereby, the AUTOSAR task definition follows the Deurema task specification, where priorities, periods, and task trigger can be translated one-to-one.

There are two further aspects for the autonomous driving module in the Deurema SmartCar system template shown in Figure 8.2. First, the module reflects and affects the Motor software

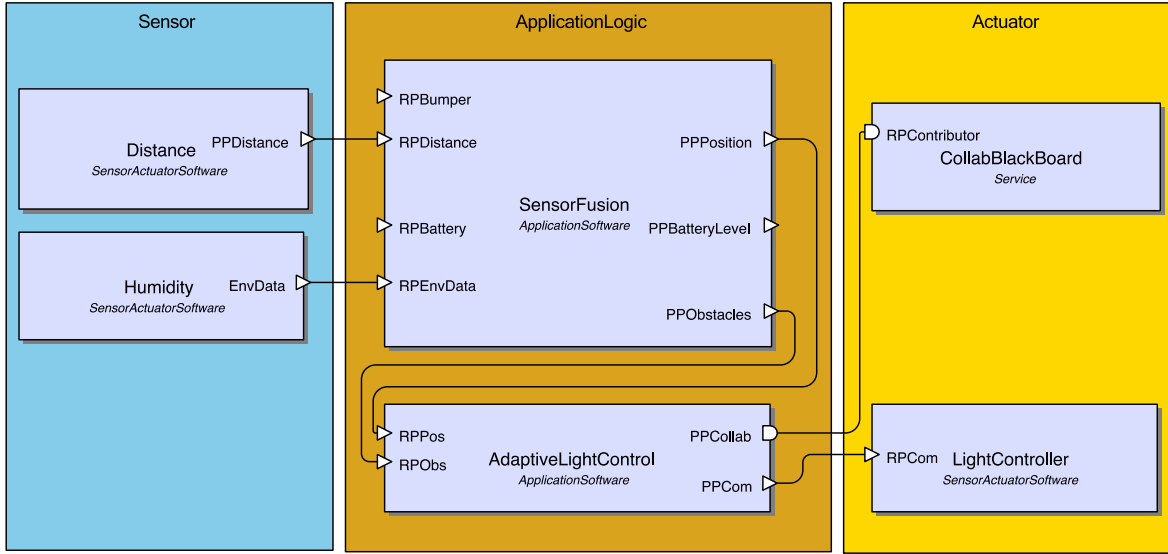


Figure 8.9: AdaptiveLightControl as AUTOSAR component architecture

module. Second, it participates in a BlackBoard collaboration. As already discussed before, the software module is realized by an AUTOSAR complex device driver component. Therefore, this specific component can be reused and appears in the component architecture in Figure 8.8, whereas the wiring of the interfaces enable the reflection and affection of the corresponding module (cf. the software module realization of the *Motor* in Figure 8.6).

Concerning the second aspect, Deurema collaborations are realized by AUTOSAR service components. The service component contains the appropriate number of runnables, which corresponds to the modeled interactions in the Deurema orchestration specification. Additionally, the service components offer a client-server port/interface, which enables the invocation of the interaction by a runnable according to the Deurema collaboration mapping as discussed in Section 5.5.5. Depending on the collaboration and modeled knowledge specification, the AUTOSAR collaboration service component needs additional sender-receiver ports to read and write data that are necessary for realizing the collaboration interactions.

The mapping of Deurema roles and the corresponding interactions to runnables within the AUTOSAR collaboration service component has two implications. First, the AUTOSAR component can be reused from all modules that participate in the collaboration. Thus, the service component appears in the realization of the AUTOSAR autonomous driving architecture in Figure 8.8 as well as adaptive light architecture in Figure 8.9. Second, Deurema modules play the role as defined by the collaboration deployment, which corresponds to an invocation of the runnable in the AUTOSAR collaboration component. In the example in Figure 8.2, the adaptive light control module plays the c_1 :Contributor role, whereas the autonomous driving module realizes the bc:BoardController role. Both roles are mapped to AUTOSAR runnables in the collaboration service component and thus, can be invoked properly.

8.6. Feedback Loop Module Template

The Deurema feedback loop module template defines the adaptation behavior in form of subsequently executed adaptation activities as discussed in Section 5.23. In the embedded domain, a feedback loop controller is usually developed according to the embedded control model as sketched in Figure 8.10. A dedicated controller is aware of the system goals, senses the state of the environment and computes an appropriate reaction according to the current situation, which is realized by effectors afterwards. This control functionality can be realized in different ways. As emphasized in own former work in [14], the model-based development of the control functionality with tools as *Matlab* focuses on the realization of distinct control functionalities that ranges from the single adaptation activity to a complete control loop. Furthermore, those realized control artifacts can be integrated in AUTOSAR components as runnables, which is also comprehensively discussed in [14].

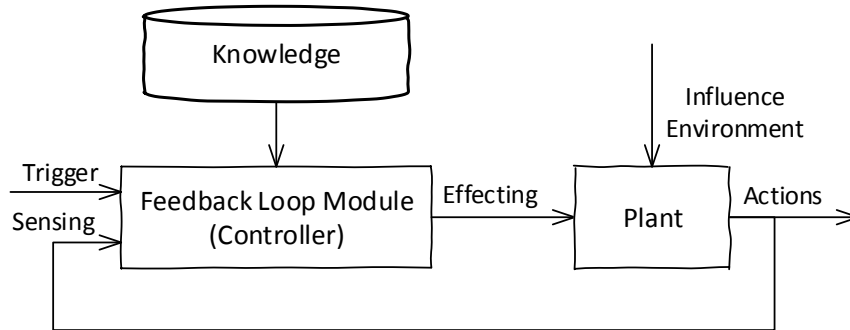


Figure 8.10: Embedded control model

As a consequence, the Deurema feedback loop module template is mapped to an AUTOSAR application software component. The adaptation activities in a feedback loop module appear as AUTOSAR runnables, where the concrete mapping depends on the implementation. Thus, one runnable may realize several adaptation activities, where the implementation must preserve the causal order as defined in the Deurema model. The mapping and invocation of collaboration interactions and the realization of knowledge access by runnables follow the same concepts as discussed above. Because runnables must be assigned to operating system tasks in AUTOSAR, all mapped runnables from the feedback loop module should be assigned to one independent operating system task. The period/trigger of the Deurema module must correspond to the period/trigger of the AUTOSAR task.

8.7. Behavior Module Template

The declarative character of behavior rules in a Deurema behavior module template can be realized in appropriate if-else conditions of an AUTOSAR runnable implementation. Thereby, each rule should be mapped to one runnable, where the parent module is realized by an AUTOSAR application component. In Deurema, each behavior rule runs independently from other rules. To get the same behavior in AUTOSAR, each runnable should be conceptually mapped on an independent operating system task, where the period and priority correspond to the behavior rule properties. However, if a large set of behavior rules must be mapped to runnables and further to tasks, the large number of arising operating system tasks can

be impractical. Therefore, rules may be grouped according to their properties (e.g., same priority), where each group is mapped to a task definition. Of course, this introduces a causal order between rules that corresponds to the order of runnables in the task, which is not modeled in Deurema. On the one hand, the causal order might be negligible, if the evaluation of the rule application condition is very simple and fast. On the other hand, grouping rules to one and the same task may reduce the scheduling overhead of the operating system. However, the concrete mapping of Deurema rules to AUTOSAR runnables as well as the assignment to tasks depends on the specific application problem and thus, must be individually optimized by the developer.

8.8. Discussion

The realization of a modeled Deurema architecture enables the analysis and simulation of the adaptive SoS for a concrete implementation setting. This chapter shows how Deurema concepts can be mapped to an AUTOSAR architecture. On the basis of this mapping, this section outlines possible analysis and simulation steps of Deurema models during different development stages modeling as well as realizing adaptive SoS architectures. Afterwards, software tools and frameworks, which are used for the realization described in this chapter, are subsumed.

8.8.1. Deurema Modeling Process

Staying in the embedded domain, Broekman et al. [43] show typical development stages and emphasize to typical testing as well as simulation activities in such stages. Figure 8.11 gives an overview of the three development stages *simulation*, *prototyping*, and *pre-production*. The first simulation stage focuses on the conceptual design of the system and the development of a simulation model representing a possible solution [43]. Therefore, the goals of verifying the system are a proof of concept testing the preliminary solution and optimizing the system design. As simulation techniques, *one-way simulation* (model test (MT)) and *feedback simulation* (model-in-the-loop (MiL)) are used on basis of the simulation model. In the context of this thesis, the modeled Deurema SoS architecture represents the simulation model, which can be directly analyzed and executed. Therefore, one-way and feedback simulation can be applied on the Deurema models as highlighted gray in Figure 8.11. Another step in the simulation stage is the *rapid prototyping*, where the simulation model is connected with the real world environment to verify assumption of the simulation model against the real world.

The second stage transfers the simulation model to a concrete realization in the problem domain. In the software-in-the-loop (SiL) step a concrete implementation for the model elements is derived and simulated, whereas the hardware-in-the-loop (HiL) step focuses on the deployment on the real underlying hardware platform. Finally, the pre-production stage focuses on the verification of the overall system behavior in comprehensive system tests using the concrete system implementation and hardware together. A comprehensive discussion about the development stages in the embedded domain and their application to embedded robotic systems can be found in own former work in [14].

On basis of the former experience in [14], the development stages of the embedded domain in Figure 8.11 and their corresponding simulation activities can be partially mapped to the analysis and testing of an adaptive SoS architecture using the Deurema modeling language. The considered simulation activities are highlighted gray in Figure 8.11. The described

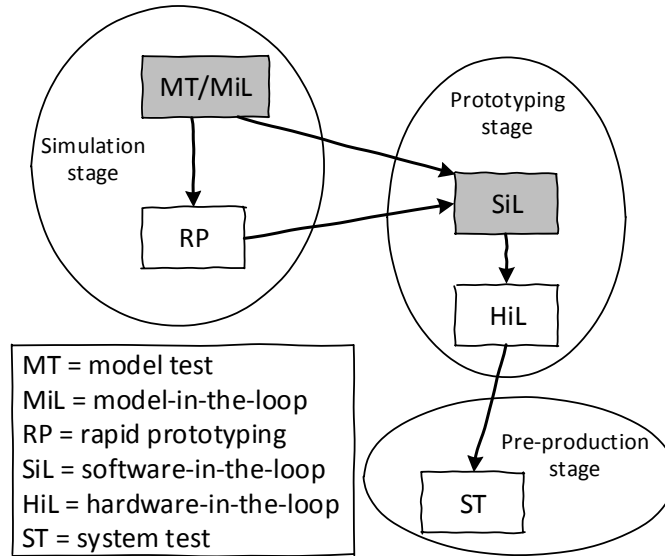


Figure 8.11: Development stages in the embedded domain according to [43]

Deurema analysis (cf. Chapter 6) and simulation (cf. Chapter 7) capabilities enable one-way simulation (MT) as well as feedback simulation (MiL). The realization of Deurema model elements to the AUTOSAR standard contributes to an implementation of the modeling concepts, whereas software tools can be used to perform SiL simulations. In contrast, the rapid prototyping step requires a connection of the simulation model to the real environment, which is hard to realize in the context of an adaptive SoS. Furthermore, HiL simulations focus on testing the underlying hardware, which is not in the focus of this thesis. Finally, system tests are necessary in the embedded domain to verify the interplay between the hardware and software realization of the complete system. Because a SoS comprises a large number of different, independent systems, which are also independently developed and governed, a complete setup of SoS system tests are unrealistic. However, focusing on single systems within the SoS, system tests and integration scenarios are thinkable. In the following, verification steps during the development of an adaptive SoS by modeling with the Deurema language are exemplarily highlighted.

Figure 8.12 shows four different situations as they might appear during the development of the running example. At the top, the figure depicts single developed modules, which is usually the case in an early stage of the development. The developer can use the Deurema software module concept to put black box placeholders at the layered system architecture. Over time, those modules can be refined by developing corresponding module templates and linking those to the module instance in the system architecture. The example in Figure 8.12 shows a single feedback loop module that encapsulates the adaptive behavior of the ABS functionality, an application module that realized the autonomous driving of the smart car and an instance of the BlackBoard collaboration together with two black box modules that realize the corresponding required roles. Each of these modules and collaboration instances can be individually analyzed and executed using one-way simulation tests as indicated on the upper left in Figure 8.12. From an engineering perspective, the internals of the modules can be stepwise refined until the overall design corresponds to the requirements of the aimed target system. In this early stage, the Deurema analysis helps understanding dependencies

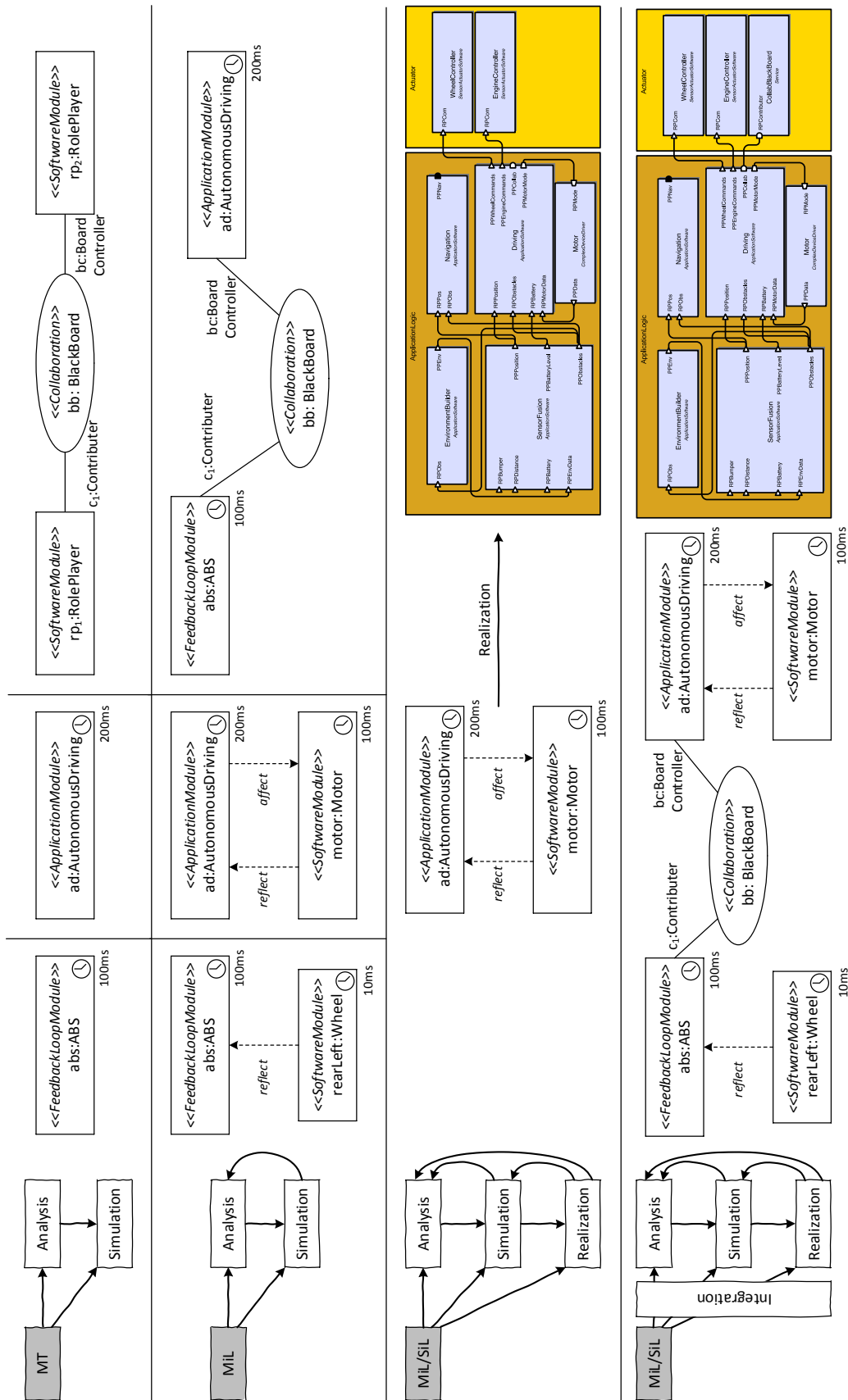


Figure 8.12: Verification steps during the adaptive SoS modeling with Deurema

within a single module as well as collaboration instance, helps developing the adaptation logic according to typical patterns (e.g., the MAPE feedback loop and the corresponding access patterns of single adaptation activities to the runtime models as described in Section 6.2.2), and pinpoints to flaws in the module template design. Furthermore, one-way simulation runs, which are restricted to one module instance, already provide a proof of concept for the internal adaptation behavior.

A first combination of multiple modules on a pure model basis is shown in the second situation in Figure 8.12. The developer can specify dependencies between modules such as using the Deurema reflection or module triggering concept. In the example, the beforehand developed ABS feedback loop module reflects another software module named *Wheel*, which enriches the overall available knowledge base in the feedback loop module and might affect the internal behavior. Beside a reflection dependency, the autonomous driving module affects an underlying software module, where this interplay of reflecting and affecting the underlying module emerges to new overall behavior. Additional to the Deurema reflection mechanism and module triggering, the interaction behavior can be investigated by connecting the developed single modules to the collaboration instance as shown for the *BlackBoard* collaboration. Thus, the beforehand mentioned black box software modules rp_1 and rp_2 in the first depicted situation in Figure 8.12 are refined by a concrete feedback loop instance *abs* and application module instance *ad*. Connecting modules and collaboration is done later in the development process and can be more and more extended by developing new modules. In this situation, the Deurema analysis can additionally pinpoint to dependencies that cross the boundaries of a single module. Furthermore, complex architectural patterns can be proposed, transitive adaptation effects become visible and the knowledge distribution can be investigated. From a simulation perspective, the Deurema execution framework facilitates the periodical simulation of well-defined subsets of the overall SoS to investigate the influence between modules and the emergent adaptive behavior over time. Of course, the subset of modules on the system architecture can be extended towards a simulation of the complete adaptive SoS.

Up to this point, all analysis and simulation steps are performed on the Deurema models. Realizing the modules in a concrete domain introduces new implementation specific effects (e.g., sensor noise or limited computational power of the embedded hardware processor), which must be analyzed and tested independently. This chapter discusses a mapping of the Deurema concepts to the AUTOSAR framework, which is exemplarily shown in the third situation in Figure 8.12. In the example, the two Deurema modules together with the reflecting and affecting dependencies are translated to an AUTOSAR component architecture as discussed above. The derived implementation can be analyzed and simulated by using available software tools from the corresponding domain. Because there are many different ways of realizing a modeled SoS, it is not in the focus of this thesis to provide such domain specific analysis and simulation tools nor to focus on a specific analysis or simulation technique.

However, during the realization, which is the transition from the simulation stage to the prototyping stage in Figure 8.11, more and more Deurema modules as well as collaborations can be integrated into the SoS. This integration is depicted in the fourth and last situation at the bottom in Figure 8.12. All beforehand isolated modules, their dependencies and the collaboration behavior are mapped to an AUTOSAR architecture, which can be analyzed and executed afterwards.

With respect to a SoS, another step is the integration of a new realized system into the existing adaptive SoS. On model level, this integration is easy in Deurema by deploying a new system instance and the corresponding system template. For the concrete realized SoS,

it might be much more complicated to integrate a new implemented system instance, which depends on the domain and thus, is not in the focus of this thesis.

In summary, Deurema models can be analyzed and simulated at different points in time during the development process. A mapping of the Deurema concepts to a concrete domain contributes to the implementation of the modeled adaptive behavior but leaves the boarder of a model-based analysis and simulation. Although, Figure 8.12 pinpoints to different steps of analyzing as well as simulating Deurema models at different points during the development, it is not in the context of this thesis to provide a predefined modeling methodology nor preferring a specific domain for realizing the Deurema concepts. This chapter discusses a realization to the AUTOSAR standard highlighting how the Deurema concepts are mapped to the embedded system domain. However, other domains or realization mappings are thinkable, but not further discussed in the context of this thesis. In the following, the used software tools for realizing the Deurema concepts to the AUTOSAR framework are discussed.

8.8.2. Software Tools

For the realization of a single adaptation effect and corresponding control functionalities the software tool Matlab/Simulink² was used. This tool allows the specification of the behavior by means of block diagrams. Furthermore, it supports code generation, where the Targetlink³ extension is used. Beside the realization of single adaptation effects, the SystemDesk⁴ tool is used to realize the component-based AUTOSAR architecture. A screenshot of this tool showing the realization of the AutonomousDriving Deurema module is depicted in Figure 8.13.

Thereby, the generated code from the Matlab tool is reused for the definition of the runnable behavior, which are located in the AUTOSAR components. Both Matlab and SystemDesk support the simulation of the modeled functionality. A comprehensive discussion about the complete toolchain can be found in [14]. Further information about the AUTOSAR standard can be found in [62]. Beside AUTOSAR, other standards are thinkable for the realization of the Deurema concepts such as OSGi, where an interesting approach is discussed in [93].

²<http://de.mathworks.com/products/simulink/>

³<http://www.dspace.com/en/pub/home/products/sw/pgcs/targetli.cfm>

⁴http://www.dspace.com/en/pub/home/products/sw/system_architecture_software/systemdesk.cfm

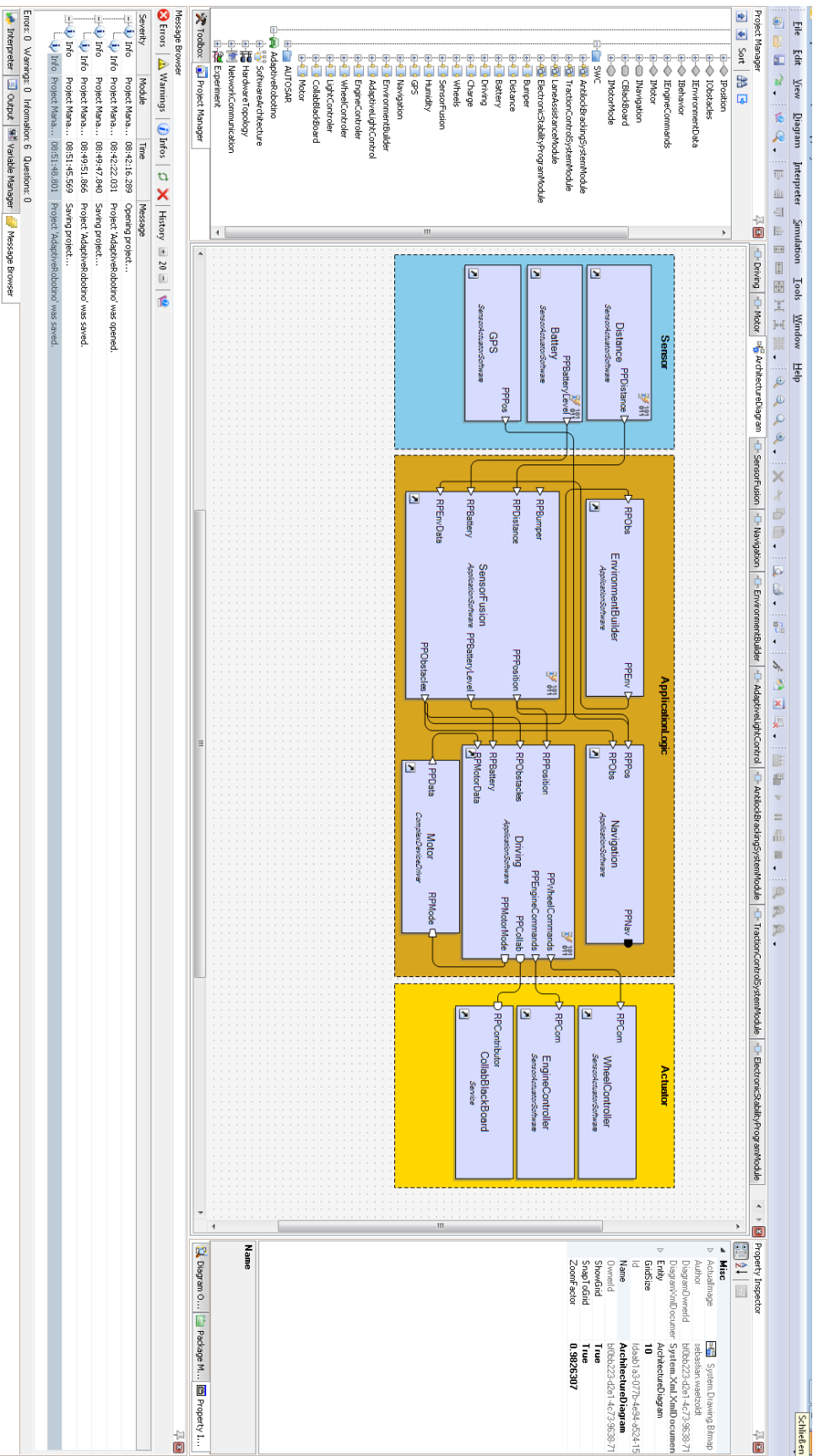


Figure 8.13: Screenshot of the SystemDesk software tool

9. Application

Additionally to the discussion in the former chapters about the Deurema verification capability by means of static and runtime analysis in Chapter 6 as well as simulation in Chapter 7 and the realization of Deurema concepts in Chapter 8, this chapter remodels two case studies to show that the Deurema modeling language concepts are powerful enough to cope with current research scenarios. The two case studies are chosen from different domains to target a broad spectrum of Deurema concepts, which is discussed in Section 9.1. Beside the remodeling of research scenarios in the two case studies, Deurema is compared with an up-to-date research approach called *DEEC*o in Section 9.2, which allows the modeling of dynamic ensembles in component-based systems. Thereby, this section discusses how DEECo concepts can be represented in Deurema to show that Deurema is expressive enough to describe such dynamic ensemble structures. In summary, the application of the Deurema modeling language in the two case studies and the DEECo approach is sketched in Figure 9.1.

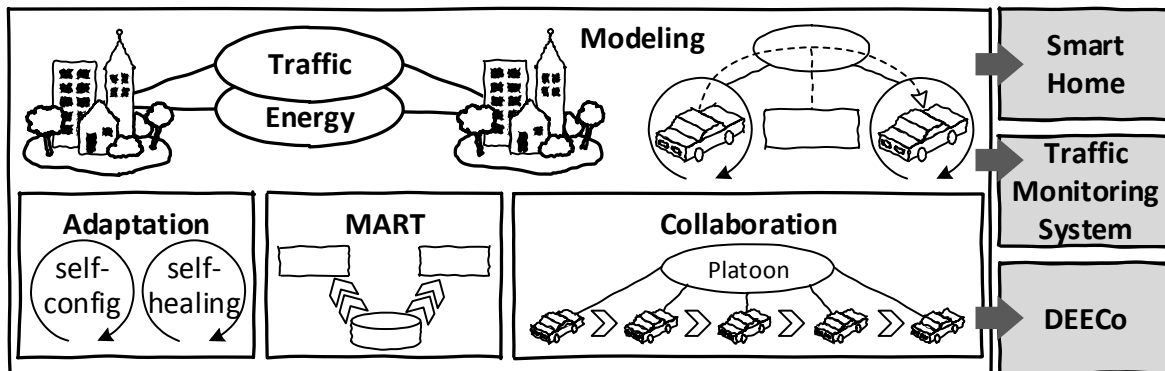


Figure 9.1: Smart city running example: Deurema application

Beside the remodeling of the two case studies and the comparison to the DEECo approach, Vogel et al. [175] already discuss the application of the predecessor modeling language Eurema to the *Rainbow* [79], *DIVA* [138], and *PLASMA* [166] approach and show that the Eurema language is expressive enough to capture these approaches. This thesis significantly extends the Eurema concepts and thus, the Deurema language can cope with these approaches, too. Therefore, this chapter does not discuss the application to these approaches again and refers to [175] and the related work discussion in Chapter 10, where differences to the Deurema language are comprehensively discussed for all the mentioned approaches above.

9.1. Case Studies

This thesis chooses two case studies to evaluate the applicability of the Deurema modeling language. First, a distributed traffic monitoring system is modeled, which is originally introduced by Vromant et al. [179]. Beside the distribution aspect, individual systems use

collaborations to coordinate each other and to contribute a specific functionality within the overall traffic monitoring system. Therefore, this case study is used to evaluate the Deurema collaboration concept. The second case study is in the context of a smart home, which extensively uses the Deurema reflection and reconfiguration mechanism in a layered, adaptive system architecture. The smart home case study is originally introduced by Cetina et al. [57]. In the following, both case studies are modeled with the Deurema language and discussed in detail.

9.1.1. Traffic Monitoring System

The traffic monitoring system introduced by Vromant et al. [179] uses multiple, interacting feedback loops in varying collaboration settings. Thereby, the overall SoS consists of individual cameras that are placed along the road, where each camera detects traffic jams in its fix viewing range. Cameras adaptively build so-called *organizations*, where the viewing range of all included cameras spans the detected traffic jam. The coordination inside an organization follows the master slave interaction pattern, where the master role is dynamically elected. Furthermore, the system is able to detect camera failures that eventually trigger a reorganization of existing camera interactions. Therefore, the inter-loop coordination is necessary between cameras inside an organization and its neighbors. Beside the inter-loop coordination, cameras are designed as self-adaptive systems, where a self-healing feedback loop runs on top of the domain functionality (an agent that realizes the traffic jam detection called *local traffic monitoring system*). The case study comprises different types of collaborations that are mandatory to realize the overall SoS functionality. The system as well as the collaborations are modeled in the Deurema modeling language in Figure 9.2 according to the description in [179] without redesigning the system structure. Furthermore, the same names for subsystems, interfaces and interactions are used to model the case study as close as possible to the original. Consequently, the Deurema modeling language is powerful enough to cope with the distributed collaboration and adaptive system aspects of this case study.

At the top, Figure 9.2 shows an excerpt of an instance situation in form of a Deurema LD with three cameras inside one organization instance *org*. On the left, the internals of the camera system template are shown. Each camera consists of two subsystems and an application specific communication infrastructure placed on an overall two layer system architecture. The lowest layer contains the communication infrastructure, which is encapsulated in a black box Deurema software module. The top system layer contains the two subsystems that are the *local traffic monitoring system* and the *self-healing subsystem*. The local traffic monitoring system consists again of two architectural layers, where the *organization middleware* offers services for an *agent* to initially create and maintain organizations. Furthermore, the agent realizes the domain functionality of detecting a traffic jam. Both parts are modeled as Deurema software modules, where the middleware module triggers the agent module functionality.

The self-healing subsystem within each camera copes with different failure scenarios such as the failure of a master or slave node within one organization. In general, a camera failure affects the behavior of the direct neighbor camera nodes. Additionally, the failure of a master node affects neighbored organizations. The self-healing subsystem has a two layer architecture, where the *communication manager* software module on the bottom facilitates the local and remote communication of the camera system. Local messages are sent to the local traffic monitoring system, where remote messages are forwarded to the communication infrastructure module. Therefore, Figure 9.2 depicts two communication collaboration instances c_1 and c_2 within the camera system. The c_1 collaboration enables a communication with a self-

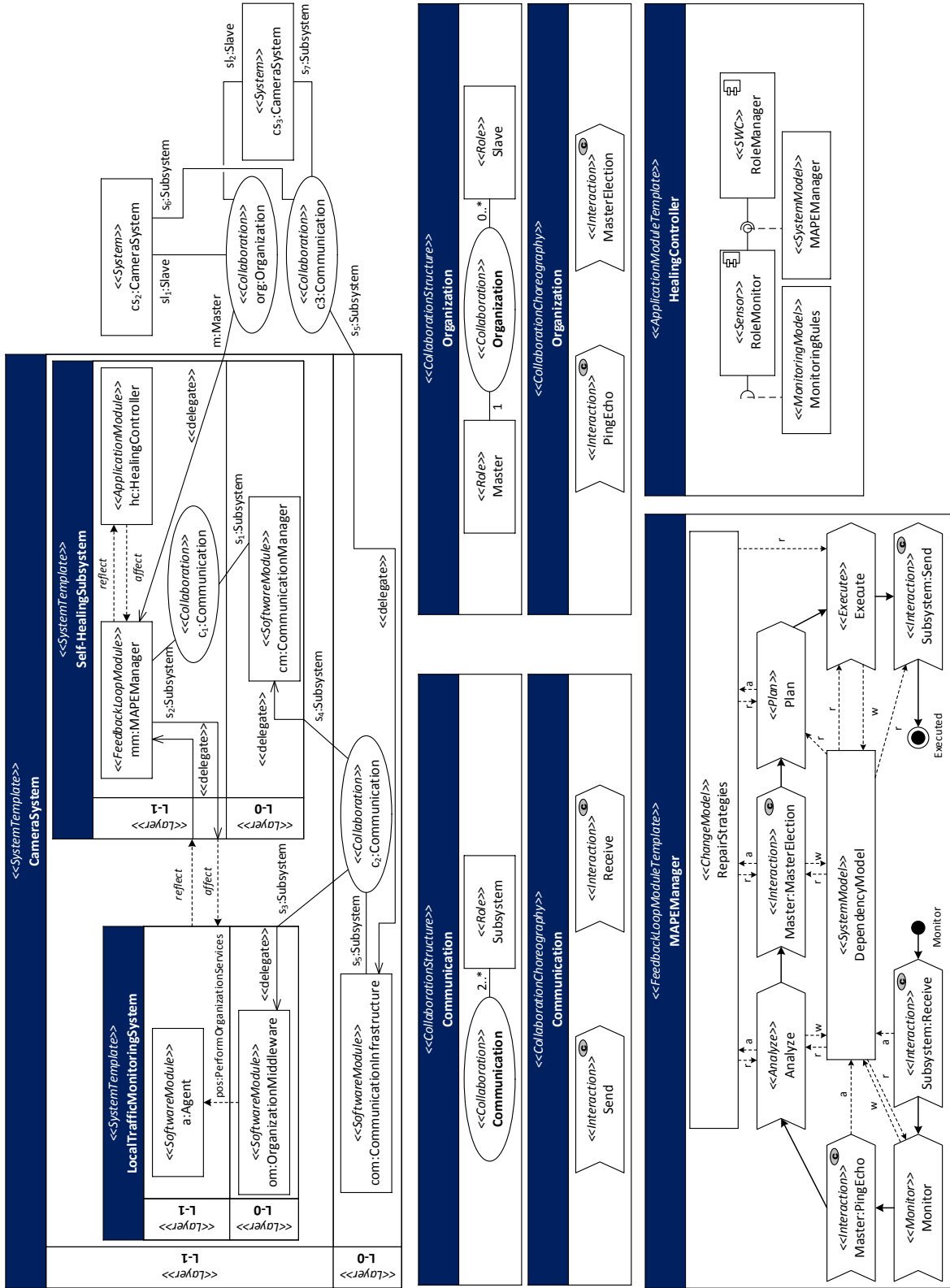


Figure 9.2: Traffic Monitoring System modeled with Deurema

healing feedback loop at layer L-1 in the camera system. Whereas the c_2 facilitates the local message transfer to the traffic monitoring system and the remote communication over the communication infrastructure module. The self-healing logic is realized by the MAPEManager feedback loop module in the self-healing subsystem at layer L-1. Therefore, the MAPE manager reflects the local traffic system, detects failures and maintains a dependency model to other cameras in the organization. If a failure is detected, the manager may communicate with other MAPE managers to choose the best repair strategy for the organization, which is based on the dependency model and includes the current played role of the camera. Beside the feedback loop, a *healing controller* observes the self-healing logic and employs appropriate repair strategies according to the failure scenario and the role of the camera system. Therefore, the healing controller adapts the underlying pool of functionality of the MAPE manager.

In the LD in Figure 9.2, there are two cameras performing as slaves within one organization collaboration instance. The camera system, which is depicted in the system template notation, performs the master role. Furthermore, the communication collaboration c_3 realizes the message exchange of the distributed cameras. In the following, the details of the collaborations are described.

There are two collaboration types in this case study that are depicted in the middle of Figure 9.2. The first collaboration realizes the communication infrastructure that is used for two aspects. On the one hand, it enables a communication between cameras, which is needed for the master election as well as to report camera failures. On the other hand, the communication collaboration links the domain logic (traffic jam detection) with the self-healing subsystem. Therefore, the Communication collaboration has one role type named *Subsystem*. The number of role instances is not limited as long as two subsystems collaborate with each other, which is defined by the multiplicity 2.* in the collaboration structure definition in Figure 9.2. The choreography specification of the communication collaboration comprises a generic *Send* and *Receive* interaction, which allows the sending respectively receiving of arbitrary message types. In [179], the authors present different message types that are used for communication, which are directly supported by the Deurema message concept. Thereby, synchronization messages of arbitrary types are supported. Furthermore, Deurema model messages can be used to send additional data to another collaboration participant.

The second collaboration type considers the coordination between cameras and is named *Organization* accordingly. One organization consists of exact one *Master* role and an arbitrary number of *Slave* roles, which realizes the mentioned master-slave pattern. It has to be noted that the *Slave* role type is an optional multi-role (multiplicity 0..*). As a consequence, an organization may contain only one camera that is in this case the master node. In the context of the organization collaboration, there are two interactions between cameras. First, a *ping-echo* protocol detects camera failures and is encapsulated in the corresponding *PingEcho* interaction in Figure 9.2. The ping-echo protocol works in the same way as the *HeartBeat* interaction discussed in Section 5.5.3, where the concrete interaction template is depicted in Figure 5.50. The second interaction handles the master election within an organization. The internals of the master election are outlined in [179] and consist of simple sequences of exchanging messages between the participating cameras. The master election is performed in the case of a camera failure that acts as master node. The master election results in a new determined master camera node and triggers a reorganization of the current organization collaboration according to the new role allocation.

At the bottom of Figure 9.2, there are two module template definitions from the modules placed in the layer L-1 in the self-healing subsystem. As outlined above, the MAPE manager

detects and repairs camera failures based on the own role allocation and the current situation in the organization. Therefore, the MAPE manager consists of a full MAPE cycle, which additionally integrates the deployed roles as depicted in the LD on top in Figure 9.2. The template definition shows the feedback loop module template for the master (organization collaboration) and subsystem (communication collaboration) role. The feedback loop performs around a common knowledge base, which is represented by two runtime models. The `DependencyModel` describes the current situation of the cameras in the own organization as well as maintains relationships to neighbor organizations. Furthermore, the `RepairStrategies` runtime model contains appropriate recovering strategies in the case of a detected camera failure. According to the role deployment, all interactions from the collaborations are woven into the local feedback loop behavior.

At first, the feedback loop starts with the `Receive` interaction to coordinate necessary self-healing actions with other cameras, which is enabled by the deployed `Communication` collaboration. Afterwards, the local `Monitor` activity updates the dependency model, which includes the current status of camera failures. On basis of a consistent updated view, the `PingEcho` interaction is performed, which synchronizes the own status with other cameras in the same organization and in particular with the master node. The subsequent analysis activity checks whether a reorganization of the current organization is necessary, which can be in the case a camera failure is detected or if another camera joins the organization because of a larger detected traffic jam. Therefore, the master election interaction is executed afterwards, which leads to a planning activity of necessary local adaptation steps. Finally, the `Execute` activity performs the planned system adaptations and synchronizes the changes with other cameras using the `Send` interaction.

Beside the feedback loop, an additional `HealingController` supervises the feedback loop and optimizes its strategy according to occurring role changes. The healing controller follows the component-based specification approach and consists of two components and two runtime models. The `RoleMonitor` sensor component determines the current role of the MAPE manager, which is enabled by the `MonitoringRules` runtime model. If the role changes, the `RoleManager` component is informed via an annotation in the corresponding system runtime model. The role manager adapts the underlying feedback loop strategy for dealing with failure events that are relevant to the new role. If for example a camera changes its role to perform as master node within an organization, the healing controller detects this role change and adapts the strategy of the underlying `MAPEManager` feedback loop accordingly. Adapting the MAPE manager includes the exchange and integration of appropriate role interactions into the local feedback loop behavior.

The Deurema approach is powerful enough to describe the collaboration aspects of the distributed traffic monitoring system. Thereby, a broad spectrum of Deurema concepts is used. First, the encapsulation of the intra-loop and inter-loop communication in a collaboration type enables the reuse of the generic send and receive interactions on different layers within the adaptive SoS. For example, the c_1 communication collaboration instance between the communication manager and MAPE manager within the self-healing subsystem follows the same interaction specification as the c_3 communication collaboration instance between cameras. Furthermore, Deurema facilitates the deployment of collaboration instances on arbitrary system layers as well as directly supports the delegation of roles to an inner system module. The role mapping in a Deurema LD clearly distinguishes the different responsibilities of the collaboration participants, where the concrete interactions of the collaboration are woven into the local adaptation behavior of the corresponding role playing module. For this case

study, three different Deurema template types are used. The domain specific behavior of the traffic jam detection and the communication infrastructure are modeled as software modules. Additionally, the modeling of the self-healing feedback loop and the supervising healing controller are directly supported by an appropriate template type in Deurema. Thereby, the available data can be seamlessly integrated into the module templates as well as in the collaboration interaction. In summary, this case study uses a broad spectrum of Deurema concepts such as the reflection mechanism between systems and modules, collaborations, and the specification of a layered adaptive architecture. After the specification with the Deurema language, the modeled traffic monitoring system can be investigated using the Deurema analysis framework (cf. Chapter 6) to detect design flaws, architectural patterns, or to get a better understanding of the specified interactions as well as the influence of the adaptation effects. Additionally, the system can be simulated to further investigate runtime effects as comprehensively discussed in Chapter 7.

9.1.2. Smart Home

Where the traffic monitoring system heavily uses the Deurema collaboration concept for coordinating the distributed systems, the smart home case study shows the applicability of the modeling language for multi-layered adaptive system architectures. Furthermore, the Deurema adaptation and reconfiguration capabilities are used to change the overall available functionality at runtime. The smart home case study is taken from Cetina et al. [57], where the adaptive smart home architecture as well as the variability model are not changed, but directly adopted by modeling the system with the Deurema language. Figure 9.3 shows the modeled smart home case study.

In general, a smart home has several task controlling devices such as heaters, lights, cameras, and multimedia according to the individual needs of the user. For example, if the user is not at home, the smart home can switch off the lights and turn down the heaters to save energy and costs. Another scenario is the closing of open windows if it starts to rain outside. Beside the behavioral adaptation on environmental changes and user needs, the smart home should have self-configuration capabilities. New devices can be incorporated into the system such as a new movement sensor or heater controller. The smart home can support the integration of such new devices by offering configurations or new features that suit the current available pool of functionality. Beside new incoming devices, the smart home has to deal with device failures. If for example a single heater fails, the smart home can inform the user and automatically adapt the heating behavior of neighboring heaters to keep the room temperature as expected.

At the top, Figure 9.3 depicts the adaptive, layered system architecture of a smart home as described in [57]. There are three layers, which semantically group devices, services, and the reconfiguration logic of the smart home. The lowest layer named *Device* contains all physically available devices. The access to the specific device is encapsulated in a Deurema software module. There are five devices that are a TV, the lights, two movement sensors and an alarm. The *Lights* module controls all lights in the smart home and can individually switch them on or off. The two movement sensors are able to detect a human being and the alarm will inform the user as well as the police if an intruder is detected.

The middle layer of the adaptive architecture contains all active functionalities of the smart home, which are grouped in different running *services*. All services are modeled as application modules and thus, the internal adaptation logic follows the component-based design principle. Figure 9.3 shows two different scenarios of running services. In the first scenario, the user is not home and only the lighting service and the security service are running. The lighting

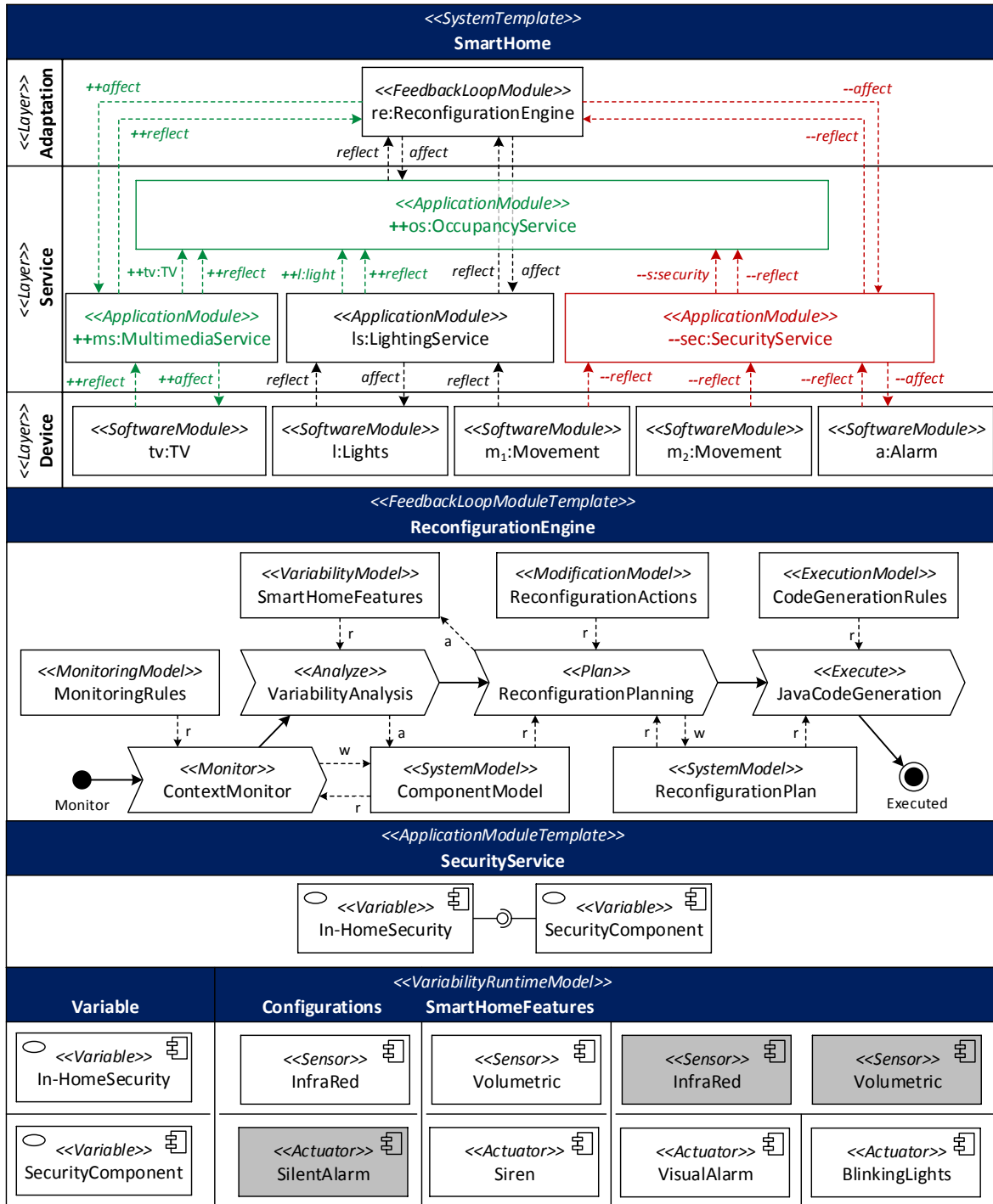


Figure 9.3: Smart home modeled with Deurema

service reflects the status of all lights in the house and because the user is not home switches all remaining activated lights off. The security service uses the movement sensors to detect intruders and sends in the positive case a silent alarm to the user as well as to the police. The second scenario deals with the situation, where the user is at home. The new available services for the second scenario are highlighted green and are denoted with a ++ sign. Furthermore, services that disappear are highlighted red with a – sign. The user moves through the house and an occupancy service handles the current situation and optimizes the available devices accordingly. For example, if the user turns on the TV, the occupancy service is aware of that situation by reflecting the corresponding multimedia service. Furthermore, it instructs the lighting service to dim the lamps. Again, the lighting service uses an in-home movement sensor to detect all people in the house and switches of or dims the lights accordingly by affecting the corresponding Lights module.

On top of all services, there is a single *reconfiguration engine* at the highest layer, which is realized in form of a feedback loop. The reconfiguration engine decides about the necessary amount of services that fits best to the current situation and the defined user needs. Therefore, the engine is the heart of the adaptation capabilities of the smart home. Consequently, the smart home in this case study is strictly hierarchically structured. At the bottom, the functionality of hardware devices is encapsulated in software modules and thus, becomes accessible for the services located in the layer above. The services provide different basic functionalities and know how to access as well as control the underlying devices. The reconfiguration engine supervises all services, deploys as well as deletes services, or changes the strategy of an existing service according to the given requirements. In the example scenario in Figure 9.3, when the user arrives at home, the reconfiguration engine determines the new situation and deploys the occupancy service and the multimedia service. At the same time, the engine removes the security service because the detection of intruders is not needed in this situation.

The reconfiguration engine is modeled as feedback loop, where the internals are depicted in the middle of Figure 9.3. According to the case study, there is a full MAPE cycle working around a predefined set of runtime models. The monitor activity retrieves the current situation of the smart home, which is denoted as context. Thereby, it updates a component model of available devices, services and with the current situation of the smart home (e.g., the home becomes empty). Afterwards, an analysis queries the updated runtime model searching for predefined situations that fit to an adaptation need. The planning activity derives an appropriate reconfiguration plan, which includes so-called *reconfiguration actions* to derive the underlying system architecture (services). These reconfiguration actions are translated to Java code by the execute activity, which is further used by an underlying OSGi framework to perform the adaptation steps. Therefore, the reconfiguration engine directly reflects the current running services and affects the complete service layer by generating an executable reconfiguration plan.

Cetina et al. [57] uses a predefined feature model that helps deriving the appropriate configuration at runtime. This feature model can be translated to the Deurema modeling language variability concept. The configurations for the security service module of the smart home case study are shown at the bottom in Figure 9.3. The corresponding application module template comprises two component variables, which are named In-HomeSecurity and SecurityComponent. The configuration of both variables can be changed during the lifetime of the corresponding smart home system. The in-home security component variable realizes the sensing functionality to detect people as well as intruders in the smart home. Therefore, it can access different

configurations as an infra red sensor, a volumetric sensor, or a combination of both. The combined sensor configuration is the current configuration of the example as highlighted in gray in Figure 9.3. Furthermore, the security component variable realizes the notification of the user as well as the police in the case of a positive intruder detection. The variable comprises four configurations that are a silent alarm (current configuration in the example), a siren, a visual alarm and the blinking of the house lights. The reconfiguration engine reflects the Deurema variability model, which is represented by the `SmartHomeFeatures` variability runtime model in the feedback loop template definition. Of course, new configurations can be created dynamically at runtime. The planning activity can derive such new configurations, which can be stored in the same runtime model and become available for the analysis activity in the next execution cycle of the feedback loop.

In summary, the smart home case study has a hierarchical architecture that uses the Deurema reflection mechanism to reason about as well as change underlying modules according to the current situation. Thereby, this case study uses the Deurema variability concept to model predefined configurations following a given feature model. Although the smart home case study focuses on one house, this case study can be easily extended by introducing more smart homes that start to collaborate with each other. A possible scenario is the optimizing of local electrical energy production and consumption with respect to a smart grid. In such a scenario, each smart home may negotiate its electrical power contribution to the grid by simultaneously optimizing its own behavior according local goals. However, the extension of the case study can be easily modeled with Deurema by instantiating the `SmartHome` system template multiple times and specifying additional collaboration protocols between the houses. Furthermore, variables in system templates can be used to individualize the smart home system templates, which introduce additional variability in the scenario. As a consequence, multiple smart house instances with corresponding collaborations shifts the focus from a hierarchical control architecture to an adaptive SoS scenario, which can be analyzed as well as simulated with Deurema.

9.2. Ensemble-Based Component Systems

Similar to SoS, an Ensemble-Based Component System (EBCS) is defined by Bures et al. as *"distributed system composed of components that feature autonomic and (self-)adaptive behavior and are organized into emergent ensembles to achieve cooperation"* [50]. Therefore, an EBCS shows the same distribution and emergent behavior characteristics as SoS. In contrast, an EBCS focuses on systems that follow the component-based paradigm, which is included but not required for SoS. Furthermore, Bures et al. [50] present the Distributed Emergent Ensembles of Components (DEECo) approach for describing EBCS. DEECo defines an own component model that facilitates the encapsulation of the adaptive behavior. Furthermore, it considers component interaction in so-called *ensembles* as first class entities. Finally, Bures et al. present a Java framework (jDEECo) that implements the DEECo component model for execution and simulation purposes.

Because DEECo targets the description of highly dynamic component ensembles as well as their interaction, it is a valuable candidate for showing the expressive power of the Deurema modeling language. Therefore, this section maps the DEECo concepts to the Deurema modeling language concepts showing that Deurema is powerful enough to cope with the DEECo concepts.

DEEC_{Co} introduces two first class concepts that are *components* and *ensembles*. Components are independent entities that comprise knowledge and processes. An interface description denotes the available knowledge in an ensemble. A set of components is linked within an ensemble, which is the only way for a component to interact and share data with other components. The DEEC_{Co} component mapping is shown at the top in Figure 9.4. Because Deurema application module templates are designed to define the adaptive behavior following the component-based paradigm, they can be used to represent DEEC_{Co} components. Thereby, each DEEC_{Co} component specification can be mapped to one Deurema component.

Furthermore, DEEC_{Co} components contain an arbitrary number of *processes* that manipulate the available knowledge of the component. Deurema defines the available knowledge in form of runtime models. Each Deurema component can have an arbitrary number of ports, where each port references a runtime model. The entities that manipulate those runtime information are Deurema runnables, which are located within the component and access the corresponding runtime models over the specified ports. Thus, each DEEC_{Co} component process is mapped to a Deurema runnable. DEEC_{Co} supports the direct execution of a component, which includes the execution of the internal processes. In contrast, Deurema does not directly execute a component description, but rather uses tasks for the execution of the adaptive behavior. Therefore, all DEEC_{Co} processes (runnables) must be assigned to an individual task, where the period of the Deurema task corresponds to the period of the DEEC_{Co} component specification. The execution semantic of a runnable within the task corresponds to a DEEC_{Co} process. A DEEC_{Co} process defines a period or a trigger, where the former specifies a periodic execution and the latter a condition that has to be met for execution. Both can be represented by the task period and trigger in Deurema (cf. Figure 5.28 in Section 5.3.4). Additionally, Deurema allows different priorities for tasks, which is not supported by DEEC_{Co}.

DEEC_{Co} uses ensembles to describe the dynamic interaction between components, where each ensemble has a predefined structure. There is one dedicated coordinator and an arbitrary number of members. The predefined structure of a DEEC_{Co} ensemble can be modeled with the Deurema collaboration concept. Therefore, an ensemble corresponds to a collaboration with the single role Coordinator and multi-role Member as shown in the collaboration structure specification in Figure 9.4. In DEEC_{Co}, components play their corresponding role as defined in the ensemble specification, which is similar to the Deurema role mapping as described in Section 5.5.5. Thereby, each DEEC_{Co} component can participate in multiple (overlapping) ensembles, which corresponds to a mapping of multiple roles from different collaboration instances to one module instance in Deurema. Thus, interactions of the same component in different ensembles are directly supported by Deurema.

The interaction between components within a DEEC_{Co} ensemble is limited to a knowledge exchange from the coordinator to the members. This one-to-many interactions can be modeled with a single Deurema model message as shown at the example interaction at the bottom in Figure 9.4. Furthermore, Deurema supports different message properties (e.g., synchronous and asynchronous messages), which are not further supported by the DEEC_{Co} approach.

For the execution of the DEEC_{Co} components, a corresponding framework translates the component description to a Java implementation. In contrast, an adaptive SoS architecture and included collaborations modeled with Deurema can be directly simulated. In summary, DEEC_{Co} allows the specification of components that include the adaptation logic and knowledge. These components can interact with each other over a predefined ensemble collaboration, where the coordinator distributes the knowledge to all member components. Deurema is able to map the DEEC_{Co} component concept within the application component concept.

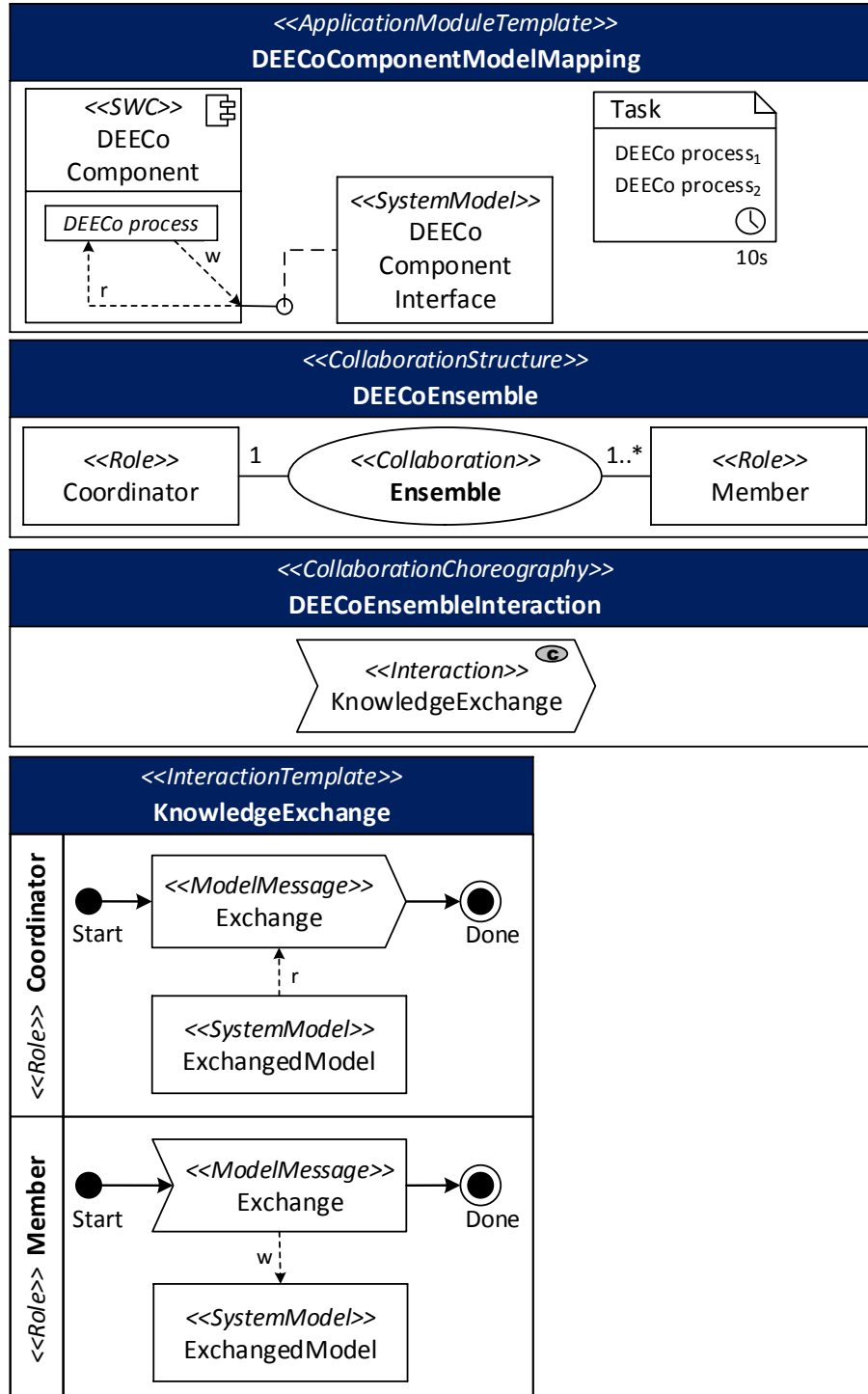


Figure 9.4: Mapping DEECO concepts to Deurema

Furthermore, the ensemble and the corresponding roles can be easily modeled by a corresponding Deurema collaboration structure. The possible interaction belongs to the Deurema collaboration choreography specification and includes a one-to-many exchange of knowledge realized by a Deurema model message. In contrast to the predefined single ensemble interaction concept in DEEC_o, Deurema supports arbitrary collaboration structures as well as choreography specifications. Consequently, the mapping of the DEEC_o component model together with the ensemble concept shows that the Deurema modeling language is powerful enough to remodel those concepts. Furthermore, DEEC_o focuses on distributed, component-based systems. In contrast, the Deurema concepts are developed to describe the adaptive architecture of a SoS together with the collaboration aspects of contained systems.

However, this chapter shows the applicability of the Deurema modeling language by presenting two case studies. Moreover, Deurema can easily map the concepts of a state of the art approach in the context of distributed, component-based systems. The next chapter outlines the related work for the Deurema modeling language and compares it with respect to the supported concepts.

10. Related Work

This chapter discusses related approaches concerning the modeling of collaborations in adaptive SoS on basis of the derived modeling language requirements from Chapter 3. According to the goals of this thesis, the Deurema modeling approach facilitates the systematical modeling of the adaptive SoS behavior, the integration of runtime models and the explicit description of collaborations. Furthermore, this thesis presents an analysis framework together with meaningful metrics to investigate the adaptive SoS modeled with the Deurema approach. Beside the analysis, a simulation framework supports the execution of Deurema models and the investigation of the system behavior by means of a runtime analysis. The realization of Deurema models in a concrete domain by mapping the Deurema modeling concepts to the AUTOSAR standard shows the applicability of the Deurema approach for actual systems. Finally, the remodeling of state of the art case studies from research emphasizes that the Deurema modeling language are powerful enough to cope with current research problems. In the following, these enumerated contributions of this thesis are compared with related approaches from the current literature.

At first, Section 10.1 describes different general purpose modeling languages in the context of software and systems engineering. Thereby, ideas that influence the design of the Deurema modeling language are highlighted. Afterwards, Section 10.2 focuses on related domain specific approaches that target different aspects such as using runtime models, collaborations, or modeling adaptation capabilities. Those existing domain specific modeling languages are enumerated and compared with the Deurema approach. Section 10.3 investigates related frameworks towards existing approaches for the realization, simulation and execution of modeled collaborative, adaptive behavior. Furthermore, this section enumerates approaches that propose patterns in the context of modeling multiple, interacting feedback loops. The roots of the Deurema modeling language are discussed in Section 10.4 by referring to former work from the research group. Finally, this chapter summarizes the related work discussion in Section 10.5 by pinpointing to own former work that contributes to the development of the Deurema modeling language.

10.1. General Purpose Modeling Languages

First of all, Deurema is a modeling language, whereas existing general-purpose modeling languages provide ideas and concepts that influence the design of the Deurema language summarized in Table 10.1. There are two conceptual perspectives for system modeling using a general-purpose modeling language, which is the *system perspective* and *software perspective*. From the systems engineering perspective, the Systems Modeling Language (SysML) [89] is a general-purpose modeling language specified by the OMG for modeling complex and possible distributed systems. Therefore, this modeling language comprises the specification of requirements, system structure, behavior and constraints on system properties. SysML does not focus on specifying the software part of a system, but rather defining the overall system architecture, its subcomponents, distribution, and resource allocation aspects.

Table 10.1: Modeling language requirements mapping for general purpose modeling languages

Feedback Loop Modeling	Collaboration	Runtime Models	Communication	Adaptation	Development	Analysis
Explicit (R-01) Intra-Loop (R-02) Inter-Loop (R-03) Trigger (R-04)	Distribution (R-05) Delegation (R-06) Roles (R-07) Protocol (R-08)	Explicit (R-09) Management (R-10) Partial (R-11) Exchange (R-12)	Multiple (R-13) Synchronization (R-14)	Reconfiguration (R-15) Adaptation (R-16) Meta-Adaptation (R-17)	Off-/ Online (R-18) Pattern (R-19) Domains (R-20)	Causality (R-21) Knowledge (R-22) Static (R-23) Runtime (R-24) Simulation (R-25)
SysML [89]	✓	✗	✓	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
ADL [60, 139]	✓	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
UML [87]	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
SoaML [88]	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
Combining UML and SoaML by Sanders et al. [157]	✓	✗	✓	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
Role Based Modeling Approaches by [152, 153]	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗						

✓: fulfilled; □: partially fulfilled; ✗: not fulfilled

One simple but powerful concept of SysML is the structural system specification via *blocks*. Blocks are modular units that can be hierarchical decomposed, interact with other blocks, or contain constraint properties of the corresponding subsystem part. Consequently, on basis of the SysML models, further design, verification, and validation activities are enabled [89]. On the one hand, due to SysML focuses on systems engineering, it supports a broad range of different system types such as CPS as well as SoS and combines hardware as well as software specifications. On the other hand, SysML lacks in comprehensive modeling techniques for the software related parts of the system. Especially, SysML does not consider structural dynamics of self-adaptive systems, feedback loop modeling, the representation of knowledge as runtime models, and collaborations as first class concepts.

Another possibility of describing systems are Architecture Description Languages (ADL). ADL *"are formal languages that can be used to represent the architecture of a software-intensive system."* [60]. Such an architecture specification comprises subsystem components, structural system patterns, and interaction mechanisms between components. For enabling further simulation and verification capabilities, ADL are based on well-defined formal notations. For example, the *ArchWare* ADL introduced by Morrison et al. [139] is based on the π -calculus process algebra and supports evolvable architectures as required by SoS. Although there are a lot of other formal architectural notations, there is no ADL covering dynamic collaborations between feedback loops of large SoS. In general, ADL focus on building concrete system solutions rather than describing changing system structures at runtime. Additionally, the representation of runtime knowledge, e. g., the requirements or runtime constraints, is not in the focus of ADL. A comprehensive discussion about different ADL can be found in [60, 139].

In contrast to the modeling from the system engineering perspective using SysML or ADL, the Unified Modeling Language (UML) [87] enables the modeling of SoS architectures from the software perspective. The SysML and UML are no distinct modeling languages but rather overlap in basic concepts (e. g., the SysML reuses a subset of UML concepts to define an own language extension).¹ However, as emphasized by Mittal et al. [137], both modeling languages can be used describing different architectural perspectives of the overall SoS. The UML [87] provides language concepts for modeling architectural collaborations. In these collaborations, roles with dedicated interfaces describe the behavior of the systems while the SoS level behavior emerges from the interactions of these roles. Furthermore, the Service-oriented architecture Modeling Language (SoaML) [88] is an UML profile that extends the collaboration concepts of the UML in the context of service compositions. SoaML provides advanced modeling concepts as for example the specification of service contracts, service choreographies, service roles, and service hierarchies (compositions) in the context of service-oriented systems. Both modeling languages UML and SoaML provide a set of building blocks to describe high level collaborations, an interaction behavior, or the collaborating roles. However, both approaches lack in a formal semantic and therefore in simulation as well as verification capabilities. Moreover, both approaches do not focus on adaptive SoS systems and therefore do not support concepts of feedback loop modeling nor taking the specifics of the tight interaction between the physical and cyber world (as needed for CPS) into account. Furthermore, both modeling languages do not consider runtime models as first class entities.

Sanders et al. [157] use the UML collaboration concept and extend the interaction between systems by so-called *semantic interfaces*. They use the Object Constraint Language (OCL) to define the goals of a collaboration and model the internal collaboration behavior in form of

¹For a comprehensive discussion about SysML and UML concepts, the corresponding specifications can be considered in [87] and [89].

extended state machines, which includes a behavior separation according to the collaboration roles. On basis of the interface description, the authors are able to compose elementary services to higher level services using the defined collaboration. As a consequence, Sanders et al. [157] enrich the basic UML collaboration concept with respect to a semantical notion of the collaboration interfaces. Thereby, different concepts from the UML and service specification of the SoaML are combined. Additionally, the authors show how abstract role specification can be instantiated (deployed) in a concrete service composition, which is similar to the role binding concept of the Deurema modeling language. However, the approach lacks in a formal semantic definition of the internal behavior of the services, which hinder an analysis and execution of the services as well as collaboration behavior. Furthermore, the approach is restricted to service specification and thus, it does not focus on adaptive systems or SoS. Consequently, the modeling of feedback loops, the use of runtime models and collaborations between adaptive behavior is not considered.

As emphasized for the modeling language requirements in Chapter 3, one important aspect is the encapsulation of local behavior related to an interaction in form of a role description (together with an interaction protocol and the knowledge specification). The general idea of applying role modeling concepts is introduced and comprehensively discussed by Reenskaug et al. [152] and further enriched with new role modeling concepts for framework designs by Riehle et al. [153]. In the papers, the authors describe how the role concept can be used to foster separation of concerns, which further enables the specification and usage of reusable patterns. In these papers, only general concepts concerning role modeling are discussed. The Deurema modeling language adopts the general idea of using roles to foster separation of concerns and refines the role concept into the context of adaptive SoS by tacking dynamic role assignments and emergent behavior into account.

Because of the generality of the discussed modeling languages respectively the role concept idea, they do not consider specifics of adaptive systems, emergent behavior in an adaptive SoS, feedback loop modeling, the representation of knowledge as runtime models, and the coupling respectively distribution of shared knowledge over collaborations. Therefore, as the name general-purpose modeling language implies, the discussed modeling languages provide general concepts to model a broad range of systems from different perspectives. Thereby, the modeling languages offer building blocks to describe structural as well as behavioral aspects of the system. Consequently, those general-purpose modeling language concepts are of limited suitability with respect to the derived modeling language requirement of this thesis in Chapter 3, which focus on the collaboration modeling inside an adaptive SoS. The limited applicability of the discussed general-purpose modeling language is reflected in the requirements mapping in Table 10.1.

Due to Table 10.1 subsumes the modeling language requirements concerning the feedback loop modeling (R-01 to R-04), it is not surprising that the discussed approaches above do not provide first class concepts for the explicit description of the adaptation logic in form of feedback loops (R-01), intra-loop coordination (R-02), inter-loop coordination (R-03), or the triggering of feedback loops (R-04).

In contrast, the requirements with focus on collaboration aspects (R-05 to R-08) are partially supported, where different ideas influence the design of the Deurema modeling language. Role based modeling introduces the concept of abstract roles for separation of concerns (R-07) together with the notion of a role protocol (R-08) to define the role specific behavior. In addition, SysML and ADL support the modeling of systems from a systems engineering perspective and therefore, provide concepts for modeling distribution aspects

(R-05). Furthermore, UML and SoaML adopt the role concept and offer basic collaboration concepts to model role specific behavior. Thereby, SoaML focuses on service specifications following the service-oriented architecture approach. Therefore, services can be distributed over several service providers, which is reflected in the SoaML accordingly.

Additionally, Table 10.1 shows the modeling language requirements for the explicit support of runtime models (R-09 to R-12). As emphasized above, the use of runtime models during the lifetime of an adaptive system is a specific research direction that is not in the focus of general-purpose languages. Therefore, the mentioned approaches do not offer modeling language concepts for runtime models. Although the UML, SoaML and SysML define a broad range of different model types, which is the basis for modeling systems, there are no first class concepts supporting runtime models. Of course, each model (modeled in a general-purpose language such as UML) can be potentially kept alive during system lifetime and thus, become a runtime model, but this is not in the focus of the modeling language itself.

Focusing on communication requirements (R-13 to R-14), general-purpose modeling languages offer a broad spectrum for communication and synchronization concepts. For example, the UML distinguishes between synchronous and asynchronous method invocation (R-13) in sequence diagrams. Furthermore, predefined sequences between communication partners realize a specific interaction protocol that includes the synchronization behavior (R-14) between participants.

Similar to the line of argumentation concerning runtime models, requirements with respect to adaptation aspects (R-15 to R-17) are not in the focus and thus, not supported by general-purpose modeling languages. Of course, the discussed approaches offer specific model types that can be extended to describe reconfiguration or adaptation aspects. For example, Korherr et al. [118] present an UML profile that extends the basic UML language for the explicit specification of variability models. This profile can be used to describe different variation points in a software system, which further enables the reconfiguration of the system. However, reconfiguration and adaptation concepts are not in the focus of the general modeling language approaches discussed above.

Focusing on the development modeling language requirements, general-purpose modeling languages support a broad range of different domains (R-20) by offering several model types. In contrast, best practices for applying the broad range of model types for the current underlying problem are not in the focus of these approaches. However, software and system design patterns (R-19) evolve over time and are often described in terms of a modeling language. For example, Gamma et al. [78] describe a set of common design patterns in an UML related notation. Another example is shown in Table 10.1 for the role based modeling approach described by Riehle et al. [153], who refer to a set of patterns that can be reused in the presented role based framework design.

The last category in Table 10.1 enumerates the modeling language requirements with focus on system analysis (R-21 to R-25). Because of the broad spectrum of general-purpose languages, they often lack in a formal semantical background for the provided modeling concepts. Exceptions are ADL that are based on a formalism (e. g., the π -calculus process algebra in [139]). Those ADL are further suitable to statically analyze the modeled system (R-23) or support its execution respectively simulation (R-25). There exists a subset of the UML, the Foundational UML (fUML) [91], that describes an execution semantic for a predefined subset of UML modeling concepts and thus, contributes to R-25, too. In general, there are many domain specific solutions that use a subset of existing general-purpose languages or define own additional concepts to remove the lack of a missing semantical background. With

respect to the modeling of adaptive SoS in the context of this thesis, important domain specific solutions are discussed in the following.

10.2. Domain Specific Languages and Approaches

Beside general-purpose modeling languages, there are several modeling approaches tailoring very specific problems in the context of adaptive systems. In the following, this section discusses related approaches that contain concepts or ideas, which influence the design of the Deurema modeling language. Thereby, the related approaches are again discussed with respect to the derived modeling language requirements in Chapter 3. At first, domain specific approaches from control engineering and adaptive systems are discussed in general. Afterwards, this section discusses specific directions, which are main concepts in Deurema, namely, runtime models with focus on adaptive systems, meta-adaptation approaches, formal approaches for adaptive systems, collaborations in adaptive systems and finally, related domain specific modeling languages. A comprehensive mapping of supported requirements for the following discussed approaches can be found in the Table 10.2.

Domain Specific Approaches for Adaptive Systems

This thesis follows the idea of modeling the adaptive behavior in form of feedback loops. As outlined in Chapter 2, the specification of adaptive behavior is well understood for embedded systems, where the corresponding control engineering discipline has a manifold set of formal theories and approaches to describe and develop control loops. Based on a formal foundation, the behavior of the developed control loops can be foreseen and specific characteristics (e.g., stability) can be ensured. From the control engineering discipline, Kokar et al. [117] introduce a set of different control loop designs on basis of a formal theory. The control loop designs, such as an *open-loop*, *closed-loop* (feedback), or *indirect-adaptive* control loop, are important to transfer the ideas from the hardware related control engineering perspective to a feedback loop design in software intensive systems of the corresponding software engineering domain. Although Kokar et al. present different control loop designs, which describe the general structure of the control loop, the internals of the adaptive behavior are not explicitly described.

Georgiadis et al. [80] start shifting the adaptive behavior to the software level by implementing a runtime environment, which supports the dynamic binding of domain specific functionalities in the context of distributed systems. Thereby, the authors encapsulate the domain logic in software components, which offer the inner services over ports according to an interface specification, but are treated as black boxes. Furthermore, the domain logic remains hidden in the implemented runtime framework and is realized by a dedicated management component. Thus, the approach from Georgiadis et al. is a first step towards the dynamic binding of domain specific functionality, which leads to an adaptation of the overall distributed system behavior (R-16), by tacking distribution aspects (R-05) of the components into account. A key characteristic of many approaches solving a concrete domain specific problem is the focus on a few (mostly one) self-* capabilities of the system. For example, the approach of Georgiadis et al. [80] enables a self-organizing infrastructure for distributed systems. Ortiz et al. [144] present an approach that allows a distributed task management of individual robots, which also belongs to an underlying self-organizing problem. Malek et al. [131] focus on the redeployment of individual components in distributed systems to improve the overall availability of a service, which belongs to the self-optimizing characteristic.

Table 10.2: Modeling language requirements mapping for domain specific modeling approaches

Feedback Loop Modeling				Collaboration				Runtime Models				Communication		Adaptation		Development		Analysis							
Explicit (R-01)	Intra-Loop (R-02)	Inter-Loop (R-03)	Trigger (R-04)	Distribution (R-05)	Delegation (R-06)	Roles (R-07)	Protocol (R-08)	Explicit (R-09)	Management (R-10)	Partial (R-11)	Exchange (R-12)	Multiple (R-13)	Synchronization (R-14)	Reconfiguration (R-15)	Adaptation (R-16)	Meta-Adaptation (R-17)	Off-/Online (R-18)	Pattern (R-19)	Domains (R-20)	Causality (R-21)	Knowledge (R-22)	Static (R-23)	Runtime (R-24)	Simulation (R-25)	
Control Engineering by Kokar et al. [117]																									
Self-* Approaches by [80, 131, 144]																									
✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	□
MAPE Approach by Kephart et al. [110]																									
□	□	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Coordination of Adaptation Logic Approaches by [15, 163, 164, 189]																									
□	□	✓	✓	✓	□	✗	✓	✗	✗	✗	✗	□	✓	✗	□	✗	✗	✗	✗	✗	✗	□	✗	✗	□
DSL by Fleurey et al. [73]																									
✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓
Considering Runtime Models by [57, 138]																									
✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	□
Local and Global Runtime Model Aspects by Trollman et al. [169]																									
✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	□	✗	✗	✗	✗
Partial Runtime Models by Götz et al. [86]																									
✗	✗	✗	✗	✓	✗	□	✗	✓	✗	✓	□	□	✗	✗	□	✗	✗	✗	□	✗	□	✗	✗	✗	✗
Meta-Adaptation Approaches by [18, 92, 99]																									
✗	✗	□	✗	□	✗	✗	✗	✗	✗	✗	✗	✗	✗	□	✓	✓	✗	□	✗	✗	✗	✗	✗	✗	✓
Distributed Meta-Adaptation by Piechnick et al. [149]																									
✓	✓	□	✗	✓	□	✗	✗	✗	✗	✗	✗	□	✗	✓	✓	✓	✗	□	✗	✗	✗	✗	✗	✗	□
Interleaving Meta-Adaptation with Runtime Models by Klös et al. [116]																									
✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗

✓: fulfilled; □: partially fulfilled; ✗: not fulfilled

However, all of these mentioned approaches do not explicitly model the adaptation logic in form of feedback loops (R-01) nor consider multiple interacting feedback loops (R-03, R-04) or collaborations (R-06, R-08). The approaches can be seen as representative solving existing problems in the context of distributed, adaptive systems, but they present specific solutions for a concrete problem rather than systematically use a modeling approach for the specification of the internal adaptation behavior or the coordination between components respectively systems.

Kephart et al. [110] subsume different self-* capabilities and propose a blueprint for a feedback loop (called autonomic manager) that consists of four dedicated adaptation activities named MAPE approach as already introduced in the preliminaries in Section 2.1.1. Therefore, the internal structure of the feedback loop is explicitly determined (R-01) and the sequence of the adaptation activities are known (R-02). This MAPE approach transfers the ideas from the control engineering domain to a software engineering perspective, which was first applied in the context of the autonomic computing domain, but became one dominant approach for the research concerning adaptive systems. In the context of feedback loop modules, Deurema adopts the idea of having different adaptation activities and follows the proposal from Kephart et al. [110]. In contrast, Deurema allows arbitrary sequences of adaptation activities, which extends the static structure of a classical MAPE feedback loop. Brun et al. [49] comprehensively discuss the modeling of feedback loops starting in the control engineering domain and shift the focus to a software engineering perspective afterwards.

Looking more precisely in the self-adaptation context that emphasize the use of the MAPE feedback loop approach, Alvares de Oliveira et al. [15] propose one generic synchronization protocol to coordinate different feedback loops by means of knowledge sharing and apply it to an application example in the cloud computing domain. This paper extends the MAPE approach by considering distribution (R-05) and coordination aspects (R-14), which arise by using multiple feedback loops. However, Alvares de Oliveira et al. only cover a single synchronization scheme for coordinating complete feedback loops while the Deurema modeling language supports the specification of arbitrary coordination schemes for individual feedback loop activities. Other examples for the coordination (R-08) and synchronization (R-14) of adaptive behavior are presented by Sykes et al. [164] and Stehr et al. [163]. Sykes et al. extend the ideas from Georgiadis et al. [80] in two ways. First, the monolithic manager component is replaced by a gossip protocol, which enables a decentralized coordination scheme. Second, they introduce a three layer architecture that realizes the overall adaptation logic, where each layer focuses on a specific adaptation concern. In contrast, Stehr et al. [163] describe the coordination in form of formal, declarative control rules, which specify the allowed interaction between independent CPS. In the context of the automotive domain, Zeller et al. [189] present a control architecture that handles hierarchically arranged MAPE feedback loops, where each loop realizes an individual piece of adaptation functionality. Additionally, feedback loops on a higher hierarchical layer have a unified, aggregated view on the whole knowledge and functionality of the layer below. In this approach, the feedback loop and knowledge dependency are rather fix and implicitly encoded in the formal model over the hierarchy. However, this approach extends the original MAPE approach by an explicit feedback loop triggering concept (R-04) and distribution aspects (R-05) in an overall hierarchical synchronization scheme (R-14) following the *hierarchical control* pattern. The discussed examples above extend the idea of a single MAPE feedback loop [110] by introducing multiple feedback loops together with a specific coordination respectively synchronization protocol. In contrast to the Deurema modeling language, they provide only specific solutions for a concrete underlying problem

rather than a modeling approach to systematically describe the adaptation logic together with the feedback loop interactions.

In MDE, Domain Specific Languages (DSL) are used to provide specific modeling elements that are tailored to the corresponding problem domain. Therefore, DSL can be used to fill the gap of missing concepts from the general-purpose modeling languages. On the one hand, because of the clear focus and the missing demand of generality, a domain specific modeling language often provides a small subset of modeling elements enriched with a formal semantic. On the other hand, a DSL is inherently restricted to the domain and often cannot be used for general modeling purposes. Fleurey et al. [73] propose a DSL for the specification, simulation, and execution of adaptive systems. Within the proposed DSL, system variability and adaptation rules can be modeled and the influence of specified constraints can be simulated at design time. The system variants are derived from a variability model together with rules as well as context constraints that have to be fulfilled. Therefore, system properties are modeled as first class entities, which consist of the beforehand mentioned variability model, context constraints, and rule model. The explicit modeling of feedback loops, collaborations, runtime model knowledge, and a runtime analysis of system constraints are not in the focus of the presented DSL.

Runtime Models as Knowledge Representation

Beside the specification of the adaptation logic, Deurema uses runtime models as first class entities for knowledge representation. The runtime model concept is introduced by Blair and Bencomo [32, 38] and describes the idea to keep development models alive during system execution. Furthermore, the basic characteristics of runtime models as for example the causal connection are discussed in both papers (cf. preliminaries in Section 2.2.3). Morin et al. [138] show that runtime models can be used for specifying the configuration space for a dynamic software product line. In this approach, the authors explicitly determine the runtime models in their system architecture (R-09) and link the runtime models via the causal connection to an underlying *business architecture* (the domain logic). According to a reasoning scheme and the current situation of the system, the approach generates an appropriate system configuration that is afterwards synchronized with the underlying domain logic. Thus, the authors consider different system configurations to support dynamic adaptation, which corresponds to R-16. A similar approach is introduced by Cetina et al. [57] in the context of smart homes as comprehensively discussed in the application chapter in Section 9.1.2. In contrast to the approach of Morin et al., Cetina et al. use predefined variability runtime models to describe the overall reconfiguration space of the smart home rather than generating a suitable system configuration on the fly. Trollman et al. [169] extend the view on runtime models by considering different model types. They separate the overall available runtime model types according to local and global adaptation concerns. As a consequence, Trollman et al. are aware of the fact that distributed adaptation activities (e.g., located in different feedback loops) may influence or contradict each other by manipulating the available knowledge base. This influence is increased in adaptive SoS, where different systems collaborate and share knowledge with each other as motivated in Chapter 3. Trollman et al. solve the problem by strictly separating local and global reconfiguration aspects and do not extend this idea with respect to system interactions.

Another important aspect is the availability and completeness of information stored in a (runtime) model, especially if systems share only parts caused by data privacy aspects. Götz et al. [86] present preliminary ideas, where distributed feedback loops formulate queries that are applied on the knowledge base on interaction participants. Therefore, the authors

emphasize the importance of considering different views over the complete available knowledge base, which is related to R-11. Unfortunately, in [86], they present only first ideas and pinpoint to possible solutions rather than describe a working modeling approach.

In summary, the discussed approaches handle runtime models as first class entities (R-09) and manipulate them at runtime to enforce, over the causal connection, the desired system adaptation. Thereby, each approach tailors a specific kind of runtime models (e. g., a variability model) or a limited set of different types, which is suitable to solve the underlying problem. In contrast, the Deurema modeling language supports arbitrary domain specific runtime models and pinpoints to their contained information by assigning an appropriate purpose. Furthermore, Deurema allows the integration of these runtime model information across the whole SoS, which includes, among others, the use of runtime models in modules and collaborations.

Meta-Adaptation

Beside the ability of a software system to adapt its pool of functionality to react on changing environmental conditions and requirements, for complex or long running systems the adaptation engine itself must be adapted to cope with software aging and evolution. The ability of adapting the adaptation engine itself is called meta-adaptation as introduced in Section 2.1.1 and requires an additional layer on top of the adaptation logic. Hillman et al. [99] introduce a three layer architecture that realizes the meta-adaptation concept (R-17). The lowest layer contains the domain logic. The middle layer realizes the adaptation engine called *reconfiguration manager* and a *change driver* layer monitors the adaptation engine and exchanges, if necessary, the underlying adaptation algorithm. Unfortunately, the approach does not model the internals of the adaptation logic (R-01) nor of the meta-adaptation layer and thus, the concrete implementation is hidden in the corresponding presented framework. A similar conceptual three layer approach is presented by Assis et al. [18] in the context of the hypermedia development environment called *adaptive semantic hypermedia design model*. A difference is that the authors use ontologies to semantically define the adaptation model, which is a common approach in the hypermedia context. Furthermore, RDF triples can be used to work on the adaptation model in form of *condition-action* rules, which realizes the meta-adaptation. Therefore, Assis et al. [18] use a declarative approach to define the meta-adaptation logic, which is similar to the Deurema behavior module template concept introduced in Section 5.3.5. A concrete example of using declarative rules for the modeling of meta-adaptation is discussed in Section 5.6.3.

In contrast to the declarative approach, Gui et al. [92] follow the component-based approach to encapsulate the adaptation logic in well-defined *composable adaptation planners*. The approach defines a component model, where each component contains a piece of an adaptation strategy together with some local knowledge. A *transformer framework* on top is responsible for connecting the adaptation components and conflict resolution, where the deployed set of aggregated, connected components realizes the overall adaptation logic. Thus, the framework can rearrange the number of adaptation components as well as their connections to other components, which is the meta-adaptation of the system (R-17). Deurema facilitates the specification of component-based adaptation logic by the application module template concept as explained in Section 5.3.4.

An approach that uses two coupled MAPE-K feedback loops to realize the meta-adaptation concept is introduced by Piechnick et al. [149] in form of the *ContextPoint* architecture approach. The first feedback loop is responsible to adapt the underlying domain logic and thus, realizes the adaptive capabilities of the system. This feedback loop can be extended

in two ways following the ContextPoint approach. First, the variability space (e.g., possible configurations of the system) can be adapted. Transferred to the Deurema approach, this is similar to change the underlying knowledge base that is the Deurema variability model. For example, a feedback loop on top can insert a new configuration or delete an existing one. As a consequence, an existing analysis or planning activity might change its outcome on basis of the changed knowledge base. This corresponds to an indirect influence of the adaptation logic. Second, the feedback loop on top can change the adaptation logic (feedback loop) below by adapting the modeled adaptation activities and therefore, the way these activities perform their functionality. For example, a monitor activity can be adapted by exchanging the available set of sensors (e.g., in the corresponding monitoring runtime model) and replacing the monitoring algorithm to the new available sensor situation. In this case, the feedback loop on top does not influence the adaptation logic indirectly over the knowledge base, but rather directly changes the available adaptation activities. Of course, combinations of direct and indirect meta-adaptation scenarios are thinkable. The ContextPoint approach [149] has a static architecture of exactly two coupled feedback loops, but already considers the deployment of the architecture on different devices. A runtime environment synchronizes different devices over a peer-to-peer network. Thus, the collaboration aspects between devices are not explicitly modeled and remain in the runtime environment implementation. In contrast, Deurema is not limited to two coupled feedback loops for realizing meta-adaptation and it explicitly considers the collaboration aspects between different systems respectively feedback loops.

An interleaving approach within one feedback loop is presented by Klös et al. [116]. Additionally to the classical MAPE activities, the authors add an *evaluation* and *learning* step. The former is explicitly triggered in the analysis step of the MAPE loop and the latter in the planning step. Both, the evaluation and learning, may lead to a change of the underlying knowledge base in form of removed or new generated (learned) adaptation rules. Therefore, the adaptation engine adapts itself without an external trigger (e.g., another feedback loop). On the one hand, one can argue that this behavior is no real meta-adaptation because the feedback loop works in the expected range as it was designed for. On the other hand, the available pool of functionality changes if new adaptation strategies are learned and thus, the overall adaptation behavior might change over time. Thus, the underlying adaptation engine is not static, which conforms to the definition of meta-adaptation. In the context of this thesis, it is not important to answer the question whether this changing of adaptation behavior caused by a learning step is meta-adaptation or not. The important observation is that the overall behavior of the adaptation engine evolves over time based on the underlying knowledge base. Because Deurema is not limited to a fix number of MAPE adaptation steps, but rather supports the modeling of additional adaptation activities, such evolving feedback loops can be directly specified in Deurema.

In summary, there are different approaches of realizing the meta-adaptation concept, which spawns a range from declarative rule conditions over the encapsulation in adaptive components to the use of coupled feedback loops or in-place adaptations. The presented approaches above choose one of these concepts, which fits best to the underlying problem. In contrast, the Deurema modeling language supports all of these variants for specifying the adaptation logic (e.g., component-based, via feedback loops, or rule based) and encapsulates it in corresponding module templates. Furthermore, Deurema allows arbitrary combinations of layered adaptation of these modules using the Deurema reflection mechanism (cf. discussion in Section 5.6), which enables the specification of such meta-adaptation behavior.

Formal Approaches Supporting Verification


The mapping of modeling language requirements to formal approaches for modeling adaptive systems is subsumed in Table 10.3 and subsequently discussed. In the context of service-oriented architectures, Baresi et al. [26] focus on the dynamic binding of services at runtime, which leads to a dynamic reconfiguration of the overall system functionality. Thereby, the authors describe a type graph to represent an abstract model of the underlying domain, which consists of components, ports, interfaces and basic messages. On basis of the type graph, the authors are able to formally describe varying service bindings via graph transformation rules. Unfortunately, the approach focuses on the structural representation of services in form of components and the service bindings are represented via a connected port. Internals of the interaction protocol are not considered. Although the authors use graph transformation as formal background to describe a deviating service composition, they do not describe the arising analysis capabilities in detail. Furthermore, the approach targets the modeling of service-oriented systems without taking the adaptive behavior as first class entity into account. Similar to Baresi et al. [26], Bauer et al. [27] use *partner graph grammars* to formally define evolving communication topologies for dynamic systems. Thereby, systems are represented as nodes in a graph and communication links as edges. On basis of the graph structure, graph transformation rules describe the creation of a collaboration, represented by creating edges between independent system nodes. The authors use the set of graph transformation rules and a given initial graph (instance situation) to define their *partner graph grammar*. This grammar can be used to statically analyze soundness and completeness criteria on the defined graph transformation rules and thus, of the specified system interaction. Because of the strong abstraction of systems to nodes and system interaction to edges, the approach does neither consider internals of the interaction protocol nor the internal, local behavior of the system.

Goldsby et al. [85] describe the *AMOEBA-RT* approach that targets runtime verification of adaptive software systems. Therefore, the authors use the Linear Temporal Logic (LTL) and a specific extension to define adaptation operations, which leads to the A-LTL approach. Furthermore, the authors extend an existing model checker called *AMOEBA* with respect to runtime monitoring and runtime analysis. The authors evaluate their approach by using *AspectJ* to introspect an adaptive Java program. Additionally, an automaton is generated from the LTL constraints, whereas the behavior of one execution sequence of the adaptive Java program is verified against the automaton specification. Constraint violations and the corresponding execution trace of the program can be detected and reported to the developer. This approach directly introspects the adaptation logic on bytecode level of the corresponding Java program. The authors refine the original approach [85] towards a modular verification at runtime in [191] to cope with the state space explosion problem and increasing verification costs (in terms of space and time) of their model checking approach. However, the approach neither considers collaborative behavior nor the explicit modeling of the adaptation logic in form of feedback loops.

In the context of SoS modeling, Gezgin et al. [81] describe preliminary ideas of describing the different system goals and the resulting adaptive behavior within a SoS. Thereby, the authors consider independent, possibly distributed systems, where each system realizes a specific functionality (called *service*) and can interact with other systems over ports. Furthermore, the authors describe a first idea of considering different interaction protocols that are separated by *roles*. Based on the high level service concept, the authors use graph transformation rules to describe the structure dynamics of the SoS, which consists of creating as well as deleting services and the connection of roles. The latter can be seen as establishing an interaction

Table 10.3: Modeling language requirements mapping for adaptive system modeling based on formal approaches

Feedback Loop Modeling	Collaboration	Runtime Models	Communication	Adaptation	Development	Analysis
Explicit (R-01) Intra-Loop (R-02) Inter-Loop (R-03) Trigger (R-04)	Distribution (R-05) Delegation (R-06) Roles (R-07) Protocol (R-08)	Explicit (R-09) Management (R-10) Partial (R-11) Exchange (R-12)	Multiple (R-13) Synchronization (R-14)	Reconfiguration (R-15) Adaptation (R-16) Meta-Adaptation (R-17)	Off-/Online (R-18) Pattern (R-19) Domains (R-20)	Causality (R-21) Knowledge (R-22) Static (R-23) Runtime (R-24) Simulation (R-25)
Using Graph Transformation Rules in the Context of the Service-Oriented Architecture by [26]						
Using Graph Transformation Rules for Static Analysis by Bauer et al. [27]						
AMOEBA-RT approach by [85, 191]						
A Modeling Formalism for SoS by Gezgin et al. [81]						
FORMS approach by Weyns et al. [182]						
ActiveFORMS approach by Iflikhar et al. [103]						
Mobile Learning Application Case Study on Basis of Timed Automata by Iglesia et al. [84]						
Traffic Monitoring System Case Study by Vromant et al. [179]						
Focus on Collaboration via Ensembles by Hölzl et al. [102]						
DECIDE approach with Probabilistic Models by Calinescu et al. [55]						

✓: fulfilled; : partially fulfilled; ✗: not fulfilled

between two systems. Unfortunately, the authors present only first ideas, which are promising to cope with the dynamics in the SoS, but are incomplete and lack in a formal basis.

Based on a standardized formal specification language called *Z*, Weyns et al. [182] present the *FORMS* approach. They introduce a set of basic primitives that can be used to model dependencies between the local adaptation behavior as well as the interaction aspects to other systems. Furthermore, the authors show a mapping of their FORMS approach to the MAPE-K feedback loop from [110]. Although the FORMS approach has a formal background, the authors emphasize the approach lacks in “a precise vocabulary for describing and reasoning about primary architectural characteristics of self-adaptive systems” [182]. Further research leads to the *ActivFORMS* approach [103], where the adaptation logic itself is modeled by a set of timed automata. The automata are directly executed along with the adaptive system, which enables a verification of the adaptive behavior at runtime according to given goals that are encoded on basis of the timed automata formalism. ActivFORMS uses the Uppaal² tool that again uses model checking as underlying verification concept, which has two major impacts on the approach. At first, the feedback loop consists of a set of automata that must be defined once and cannot change during the verification. Thereby, models are represented as automata or other data structures and do not appear in form of runtime models. Second, the verification has to deal with the state space explosion problem, which is inherent to the chosen verification technique. However, the ActiveFORMS approach demonstrates the verification of the self-adaptive behavior during runtime by formally modeling the underlying feedback loop. Unfortunately, the presented version of this approach in [103] does not consider collaboration aspects. Iglesia et al. [84] show how timed automata can be used to describe coordination aspects for a mobile learning application. However, in this case study, the timed automata are statically defined to realize the underlying problem and to not consider specific runtime information. Another case study is presented by Vromant et al. [179], which is the distributed traffic monitoring system as already discussed in Chapter 9.

Hölzl et al. [102] focus on the interaction aspect of distributed, independent systems by presenting a formal model for the definition of *ensembles*. Based on a global goal satisfaction and fitness function each system (an autonomous robot in the presented approach) contributes to the overall system goal. Therefore, the collaboration aspects are indirectly modeled, where each system knows about the global goals and optimizes its local behavior accordingly. Furthermore, the authors integrate the local adaptive behavior as well-defined *black boxes*, which means that the adaptation logic is not known in advance but has predefined input and output interfaces. An advantage of this approach is the decentralized control scheme. Disadvantages are the single, global goal optimization, the black box adaptation behavior and the missing integration of runtime information.

Recently, Calinescu et al. [55] presented the *DECIDE* approach that enables runtime verification of completely decentralized control loops on basis of probabilistic models such as probabilistic automata or continuous-time Markov chains. Probabilistic behavior suits the description of adaptive system behavior in unknown or uncertain environments. Furthermore, sporadic software errors or hardware failures can be modeled as probabilistic effects. Although of the growing state space for system verification that depends on the number of distributed DECIDE components and the defined system requirements, the authors show that DECIDE performs even for larger system sizes. However, the approach considers only one local control loop for each DECIDE component, whereas the internals are not modeled in detail. Furthermore, the collaboration aspects are restricted to an unspecified distribution of

²More information about the Uppaal software tool can be found at: <http://www.uppaal.org/>.

the calculated local behavior optimization to the overall global available requirements. Thus, each distributed component has a non-conflicting unique view on the overall system.

In summary, a formal basis enables the verification of the modeled adaptive system behavior according to given goals. The discussed approaches lack in specific design concepts for adaptive SoS, where the adaptation logic is defined by means of feedback loop activities. Furthermore, the use of runtime models and the resulting effects of manipulating these models are not considered. Beside verification, some approaches discuss the parallel execution of the formal models (timed automata) to facilitate runtime analysis and pinpoint to violations during the execution of the system. In contrast, Deurema supports static and runtime analysis by providing different metrics such as the causality between adaptation effects or the distribution of runtime models in form of declarative pattern. As described in Chapter 6, an inference engine runs in parallel to the adaptive SoS simulation and directly annotates found violations of the requirements or modeled patterns into the defined Deurema models. An overview about the derived modeling language requirements and discussed related formal approaches can be found in Table 10.3.

10.3. Frameworks and Patterns

Beside modeling languages, frameworks offer specific concepts and tools to ease the development or support verification of adaptive systems. This section discusses different kinds of frameworks that influenced the design of the Deurema modeling language. Thereby, frameworks from the embedded and cyber-physical domain are discussed first. Furthermore, frameworks often define a predefined overall software architecture reducing the complexity and offer well-defined customization points for inserting the domain specific adaptation logic. Therefore, layered architecture based adaptation frameworks are discussed in this context. Afterwards, frameworks with a special focus or supporting a specific concept (e.g., runtime models) are highlighted. Finally, literature that proposes patterns and best practices for the development of adaptive systems are retrieved. An overview of all discussed frameworks and pattern approaches is collected and compared against the derived modeling language requirements of this thesis in Table 10.4. In the following, each approach in the table is discussed in detail.

Embedded and Cyber-Physical Systems Domain

In the context of the automotive domain, the *DySCAS* project investigates and offers concepts for the dynamic reconfiguration of embedded and cyber-physical systems. In this domain, problems arise by the high safety demands, where the system behavior (including the reconfiguration and adaptation capabilities) must fulfill hard real-time constraints. Anthony et al. [17] define a set of reconfiguration scenarios in typical modern cars and propose first ideas of introducing the reconfiguration behavior within the *DySCAS* framework. Ward et al. [180] refine the ideas from [17] and introduce a *context manager* that is able to reconfigure the deployed software components in the system. The adaptation logic is directly encoded in the software components and due to the high demands on safety as well as real-time behavior, all reconfiguration possibilities (decision points) are developed upfront and precompiled into the software component implementation. At runtime, the context manager decides which components in which configuration could be activated, which leads to the overall available system behavior. In contrast, the Deurema modeling language follows the external adaptation approach, which splits the adaptation logic from the domain logic.

Table 10.4: Modeling language requirements mapping for frameworks and pattern approaches

Feedback Loop Modeling	Collaboration	Runtime Models	Communication	Adaptation	Development	Analysis
Explicit (R-01) Intra-Loop (R-02) Inter-Loop (R-03) Trigger (R-04)	Distribution (R-05) Delegation (R-06) Roles (R-07) Protocol (R-08)	Explicit (R-09) Management (R-10) Partial (R-11) Exchange (R-12)	Multiple (R-13) Synchronization (R-14)	Reconfiguration (R-15) Adaptation (R-16) Meta-Adaptation (R-17)	Off-/ Online (R-18) Pattern (R-19) Domains (R-20)	Causality (R-21) Knowledge (R-22) Static (R-23) Runtime (R-24) Simulation (R-25)
DySCAS approach by [17, 180]						
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
MARS approach by Trapp et al. [168]						
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
AUTOSAR and an Organic Middleware by Trummer et al. [170]						
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
Soft and Hard Real-Time in a Hybrid Component Model by Gui et al. [93]						
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
Rainbow approach by Garlan et al. [79]						
✓	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
Component Types and PLASMA by [70, 166]						
✓	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
DEECo for Ensemble-Based Component Systems by Bures et al. [50]						
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
Patterns for Decentralized Feedback Loops by Weyns et al. [183]						
✓	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗	✗
✗	✗					

✓ : fulfilled; ✗ : partially fulfilled; ✗ : not fulfilled

The *MARS* approach [168] extends the specification of different system configurations by taking the interplay between components into account. The approach offers basic modeling concepts such as *modules* to define the system behavior including different configurations. Due to static verification techniques and simulation, the correct interplay of modules is analyzed and specific safety properties such as deadlock freedom can be guaranteed. If the analysis fulfills the development requirements, the system implementation is generated on basis of the Matlab/Simulink tool. In contrast to the MARS and DySCAS approach, this thesis discusses a realization of the Deurema concepts due to a direct mapping to the AUTOSAR standard. Thereby, the adaptive behavior is realized by AUTOSAR software components that can be integrated side by side with the software components specifying the domain logic of the system. Integrating the modeled reconfiguration space of the Deurema variability model suits well with the static AUTOSAR standard [62]. Thereby, the same problems arise by realizing dynamic adaptation capabilities as known for the MARS and DySCAS approach.

Trumler et al. [170] extend the static reconfiguration capabilities of the AUTOSAR standard by introducing an *organic middleware*. In this approach, each middleware node follows a predefined architectural structure, but can differ in the amount of available *services*, which realize the adaptive behavior as well as the domain logic. The interplay between nodes is investigated by means of a corresponding simulation framework. Unfortunately, a mapping from the presented middleware to an AUTOSAR implementation is missing. On the one hand, the organic middleware enables the specification of dynamic adaptation capabilities. On the other hand, it introduces a gap between a concrete AUTOSAR conform implementation and the simulator realization.

To avoid this problem, Gui et al. [93] introduce a hybrid real-time component model that separates hard and soft real-time functionality of the system by design. Furthermore, for the hard real-time behavior mathematically modeled transfer functions are used that comprise the domain logic and reconfiguration capabilities of the system. In contrast, non-real-time behavior is separately developed and executed using an OSGi framework, which enables the specification and execution of dynamic adaptation capabilities. Furthermore, the authors present an interaction scheme of real-time and non-real-time behavior based on a global shared memory. The shared memory solution introduces the drawback of this approach, which is the missing distribution of components on different execution nodes. As a consequence, the interaction between components is not explicitly considered with the exception of the interface semantic. Therefore, components can only distribute and receive data, if the corresponding interfaces of the other components fit to the own interface description. Furthermore, the collaboration specific behavior is not explicitly considered.

In the context of this thesis, the Deurema modeling language concepts are mapped to the AUTOSAR standard demonstrating the realization in a concrete application domain as comprehensively discussed in Chapter 8. As a consequence, all reconfiguration and adaptation capabilities must be mapped to the AUTOSAR approach, which leads to an integration of the adaptive behavior into the AUTOSAR software components.

Architecture Based Adaptation Frameworks

Garlan et al. [79] describe the *Rainbow* framework that supports the decoupling of the adaptation logic from the domain logic following the external adaptation approach as introduced in Section 2.1.1. Thereby, the approach defines a two layer architecture consisting on a *system layer* and an *architecture layer*. Furthermore, the architecture layer consists of four predefined adaptation activities, which can be individualized according to the underlying problem. Thus, Rainbow explicitly considers the feedback loop in the overall architectural design, although

the activities are statically arranged. Rainbow does neither consider runtime models nor collaboration between feedback loops as first class entities.

Kramer et al. [119] subsume different architectural designs for the development of adaptive systems and present a three layer reference architecture, which was adopted in many following research approaches. For example, Edwards et al. [70] define a set of component types, namely *collector*, *analyzer*, and *admin*, for typical adaptation activities. The collector component is responsible for retrieving data of the underlying system, whereas the analyzer component evaluates the data. The admin component manipulates the underlying system according to the evaluation result of the analysis. The authors take the reference architecture from Kramer et al. [119] and place one instance of each component type on the adaptation layers, which leads to the following implications. First, the adaptation logic is explicitly determined by the introduced component types, which are placed on the adaptive, layered architecture. Second, the interactions between components is modeled via event triggering. Third, because the authors use two adaptation layers on top of each other, they enable the specification of meta-adaptation capabilities, although this concept is not discussed in more detail. Based on a refinement of the introduced component types, the three layer architecture from Edwards et al. [70] ends in the *PLASMA* framework as introduced in [166]. Similar to the Rainbow framework, this approach does neither consider runtime models nor collaborations in the overall adaptive architecture.

Frameworks with Special Focus

In the following, related approaches that have a very specific focus in a special domain or concept are highlighted. Although none of the following approaches focuses on the definition of a modeling language, those approaches are important because they describe ideas in related domains that are realized by Deurema for modeling adaptive SoS. At first, Cossentino et al. [65] describe the *ASPECS* approach that introduces a software engineering process for agent-oriented systems. The important aspect adopted by Deurema is the explicit consideration of collaborative behavior. Because agents are inherently considered as independent, they must interact with each other to offer high level services or contribute to a bigger system goal. Consequently, the authors introduce a set of terms such as *organization* (similar to a Deurema collaboration), *role*, *role task* (similar to the Deurema choreography specification), and *role plan* (similar to the Deurema role mapping concept). On basis of those basic terms, Deurema adopts the agent-oriented concepts and changes them to model adaptive SoS architectures that comprises several independent systems, the explicit specification of feedback loops, collaborations, and the integration of runtime models.

Kim et al. [113] propose an application framework for loosely coupled networked CPS. A key point of interest of this approach is the knowledge handling between multiple distributed nodes. The underlying framework reasons about the underlying distributed topology of so-called *cyber nodes* and provides an overall unique view on the shared knowledge base. Therefore, the approach decouples the real physical deployment of nodes by the cyber node concept and further prevents inconsistencies at the underlying data model. A fully decoupled application layer can access the underlying framework (and thus, the knowledge base) via a predefined API. Furthermore, the authors comprehensively discuss implications of simulating the *cyber framework* and realizing the modeled system for a concrete application platform. In the context of this thesis, similar observations occur. At first, the Deurema simulation framework provides a unified, non-conflicting access to an arbitrary number of runtime model views and maintains occurring changes during the simulation accordingly (cf. interpreter and simulation semantic in Chapter 7). Realizing the Deurema approach by means of the

AUTOSAR standard (cf. realization discussion in Chapter 8) introduces additional, domain specific problems of synchronizing and distributing the underlying data structures.

With respect to the component-based development paradigm, which is supported by the Deurema application module template type, this thesis is inspired by the comprehensive classification of software component models in [66]. One related approach is introduced by Bures et al. [50] called *DEEC*. DEEC combines the component-based development paradigm with the idea of dynamically build ensembles (similar to Deurema collaborations). Chapter 9 shows that the Deurema modeling language concepts are powerful enough to cope with the DEEC approach. Furthermore, Deurema additionally considers the modeling of adaptive behavior in form of feedback loops and declarative behavior rules. Moreover, DEEC does not focus on runtime model aspects. From an implementation point of view, the DEEC approach facilitates the code generation of a corresponding DEEC model, which is integrated into the jDEEC runtime framework. In contrast, Deurema supports the direct simulation of the modeled architecture, but needs a manual realization in an appropriate application domain as exemplarily discussed for the AUTOSAR standard.

Finally, a very interesting approach from the domain of smart spaces called *SeSaMe* [25] introduces an adaptive middleware infrastructure. The advantage of this approach is that existing components, as massively available in smart spaces in form of sensors or devices, can connect to the middleware to exchange data or interact with each other. Furthermore, an application layer on top of the middleware enables the decoupled development of domain specific functionality without coping with device specific details. The middleware itself is structured in different layers that deal with the component abstraction and adaptation handling functionality. Although the middleware approach enables a high integration of existing devices and an independent high level application development, the main drawback, from a modeling perspective, is the hiding of dependencies of the adaptation logic as well as for the collaboration aspect between different components within the middleware. Thus, the Deurema approach explicitly determine these dependencies in the adaptive SoS architecture to enable the reasoning about it by means of the presented Deurema analysis and simulation framework. However, from a user perspective, such middleware solutions are highly attractive, because they ease the interaction and integration of several devices in a predefined application domain.

In summary, frameworks target the simplification of the development by providing predefined architectural solutions that can be individualized in dedicated variation points. Furthermore, a corresponding execution or code generation environment helps realizing the individualized solution on a target platform. However, almost all frameworks hide the collaboration aspects within the framework architecture. Therefore, the interaction of the systems is hidden in the specific implementation details of the frameworks. In contrast, a modeling language allows the specification of a broader spectrum of adaptive system architectures, which is on the one hand only limited by the supported modeling language concepts and thus, more flexible. On the other hand, a modeling language might lack in the support for an automatic realization of the specified system solution.

Patterns for Adaptive Systems

On basis of clear modeling concepts, patterns help understand common used architectural software and system designs. Furthermore, patterns highlight to advantages and known issues by applying it, which facilitates design decisions during the development and might lead to a better system respectively software design. Choi et al. [59] introduce application patterns of the beforehand mentioned cyber framework from Kim et al. [113]. The patterns focus on the

handling and distribution of the common, unified knowledge base of the underlying distributed networked CPS.

In contrast, Frey et al. [76] discuss different interaction patterns in the context of the smart grid domain. Thereby, the authors discuss responsibilities for hierarchical, stigmergy and peer-to-peer collaborations. All of these different interaction patterns can be represented by the Deurema collaboration concept, where some instances of these interaction patterns are comprehensively discussed in the analysis chapter in Section 6.3.

In the context of adaptive systems, Weyns et al. [183] describe high level patterns by combining different variants of the MAPE activities for distributed feedback loops. Examples are the *Information Sharing*, *Regional Planning*, *Hierarchical Control*, or *Master/Slave* pattern, whereas the latter appears slightly changed in the platoon collaboration with the leader and follower role in the running example of this thesis. Unfortunately, the presented patterns in [183] are described on an informal level based on the MAPE activities. Therefore, the concrete interaction between single activities, e. g., in form of collaboration as done by the Deurema modeling language, are not described. Furthermore, the integration of knowledge as well as its distribution remains unclear in the pattern description. In contrast, the Deurema modeling language provides clear modeling concepts, which enables the specification of well-defined patterns. Although some architectural patterns such as hierarchical control or layered adaptation are discussed in the analysis chapter (cf. Section 6.3), it is not in the focus of this thesis providing a comprehensive set of different patterns for modeling adaptive SoS.

10.4. Experience from the Research Group

The former approaches of the research group that influence the design of the Deurema modeling language are summarized in Table 10.5. Hebig et al. [96] identify different problems for the development of adaptive systems and emphasize the need of an explicit modeling of the underlying control loop. Therefore, the authors propose an UML profile that enables the modeling of a so-called *controller* by means of an UML component. Furthermore, sensor components are determined to retrieve data from other components (similar to monitoring activities in Deurema feedback loop diagrams) and actuator components provide data to influence other components according to the output of the controller (similar to execute activities). Interactions between the controller, sensor and actuator components are modeled by using UML interfaces. This approach enables the explicit modeling of the control architecture by encapsulating the adaptation logic in different controller components. The internals of the controller remains as black box. Furthermore, a protocol and concrete modeling of the interaction over the UML interfaces is not presented. Multiple controller are represented by distinct UML components, where a special *sequence* concept defines a causal order between different controllers. The approach is a first step of describing and structuring the adaptation capabilities without representing a concrete execution or realization semantic.

In the context of cyber-physical systems, the Mechatronic UML (mUML) modeling language enables the description of the domain and adaptation logic of the system. Thereby, the mUML concepts were extended over time with respect to different research questions. In [51, 53], the authors present the model-driven development of mUML models by means of block diagrams for describing architectural aspects and statecharts for defining behavior aspects. Thereby, the SysML block diagram concept is extended to define *hybrid mechatronic components* that transfer data over well-defined ports. Furthermore, structural reconfiguration aspects are modeled in *hybrid reconfiguration charts*, which is basically an automaton that defines the

Table 10.5: Modeling language requirements mapping for approaches from the own research group

Feedback Loop Modeling				Collaboration				Runtime Models				Communication		Adaptation			Development			Analysis									
Explicit (R-01) Intra-Loop (R-02) Inter-Loop (R-03) Trigger (R-04)	Distribution (R-05) Delegation (R-06) Roles (R-07) Protocol (R-08)	Explicit (R-09) Management (R-10) Partial (R-11) Exchange (R-12)	Multiple (R-13) Synchronization (R-14)	Reconfiguration (R-15) Adaptation (R-16) Meta-Adaptation (R-17)	Off-/Online (R-18) Pattern (R-19) Domains (R-20)	Causality (R-21) Knowledge (R-22) Static (R-23) Runtime (R-24) Simulation (R-25)	Explicit Control Loop Modeling via an UML Profile by Hebig et al. [96]																						
							□	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
							mUML and Different Extensions by [51, 52, 53, 82, 83, 100]																						
							□	×	×	□	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
rigSoaML by [28, 29, 30]				×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×			
Adaptation and Runtime Models by [174, 176]				×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×			
Runtime Megamodels by [97, 177, 178]				×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×			
Eurema by Vogel et al. [175]				×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×			
Deurema				✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			

✓: fulfilled; □: partially fulfilled; ✗: not fulfilled

different allowed reconfigurations in form of system states as well as the transition between different system configurations as state transitions. Thus, the adaptation logic is woven into the local behavior of the system and not explicitly modeled in form of a feedback loop. Because of an underlying toolchain described in [52], mUML models can be analyzed and simulated. Another extension of mUML introduces a collaboration concept [100], which already has a notion of collaboration roles. This concept enables an interaction between components and extends the static reconfiguration capabilities to dynamic structural adaptations at runtime. Graph transformation rules are used to describe the structural changes, whereas statecharts describe the interaction behavior. Another important aspect of the mUML approach is the proposal of a reference architecture called *operator-controller-module architecture*. Thereby, two independent feedback loops decouple the hard real-time system reconfiguration logic (*reflective loop*) from the soft real-time behavior (*cognitive loop*). Considering real-time constraints is a crucial point for embedded and cyber-physical systems. One advantage of the proposed architectural decoupling is the separation of concerns with respect to timing constraints. Thus, the low level reflective loop is able to react fast on changing conditions of the system state or environment, whereas the cognitive loop realizes non-critical self-optimization capabilities. However, the real-time constraints of the underlying cyber-physical system limit the overall possibility of applying arbitrary dynamic runtime adaptations. Later work formalizes modular aspects of the mUML language [82] such as components and compositions and collaboration aspects [83] towards a verification of the modeled system as well as interaction behavior. Unfortunately, the mUML approach has no notion of integrating runtime models into the proposed reference architecture. Furthermore, there is no explicit decoupling of the domain and control logic. Finally, the language focuses basically on one cyber-physical system, which can be partially extended by the collaboration concept. However, the description of several independent systems on a multi-layered SoS architecture is not in the focus of this language.

Another research direction is the verification of structural adaptation effects on models represented by graph structures. The general idea is that the current model (graph) describes the state of the system and the modeled graph transformation rules specify the possible behavior. Becker et al. [28] propose an approach called *symbolic invariant verification* that investigates the modeled graph transformation rules (the adaptive behavior of the system) and pinpoints to unwanted adaptation effects by finding forbidden future system states. The approach is extended to coordination aspects between autonomous vehicles in [29]. Thereby, the coordination is described by means of graph transformation rules that are extended by clocks representing timing aspects as needed for embedded real-time systems. The verification approach is further extended in the context of service-oriented systems via a service role and service collaboration concept, which leads to the *rigSoaML* modeling language described in [30]. Within the language, the complete adaptive behavior is modeled by means of graph transformation rules. On the one hand, this suits for the formal verification aspects of the language, which is the main intention of this approach. On the other hand, the language does not explicitly distinguish between adaptation activities, their grouping to feedback loops, or encapsulating collaborative behavior from local system behavior. Furthermore, the graph transformation rules operate on a given system state (represented as graph), whereas the representation of runtime models is not in the focus of this modeling approach.

Focusing on system adaptation in combination with the idea of abstract runtime model representations, Vogel et al. [174] show how architectural models can be retrieved from a running system and represented as runtime models during the execution of the system. The authors used triple graph grammar rules for synchronizing system observations with the

more abstract runtime model representation. Furthermore, the authors realize the causal connection and show how changes in the runtime model are translated to the running system, which enforces corresponding changes in the domain logic. The approach is evaluated by a self-healing scenario on basis of an Enterprise Java Beans (EJB) web application. Although the approach focuses on an architectural representation of the running EJB components, it demonstrates how the causal connection of runtime models to the running system can be established and maintained during system lifetime. The runtime monitoring of this approach is further improved in [176] by synchronizing runtime observations incrementally with the runtime model representation looking only at changes in the system architecture instead of rebuilding the runtime model at every monitoring step from scratch.

As discussed in the preliminaries in Section 2.2.4, the megamodel approach targets the managing of different models as well as their relationships to other models as typical in a MDE setting. On basis of a common understanding of the megamodel approach derived in [97], Vogel et al. [177] present a first idea of combining megamodels with the runtime model approach. An advantage of this idea is to benefit from the management techniques provided by the megamodel at runtime. Thus, the authors describe how multiple runtime model types can be considered during the lifetime of the system, which is an extension of the restricted architectural view in [174], and how the megamodel can be used to maintain all these different runtime models. This idea introduces a preliminary categorization of runtime models, which is further refined in [178]. In the context of this thesis, this categorization is extended with respect to modeling collaborations and a clear separation between different runtime model types in Section 5.2. Additionally, a megamodel is used that contains all runtime models within the Deurema models as well as maintains individual views in the corresponding module templates together with the defined model operations on the runtime models as performed by the different adaptation activities. At runtime (during a simulation), the megamodel is used to manage the relationships between the different deployed runtime model instances as well as module instances in the deployed adaptive systems.

Combining the ideas of runtime megamodels and the explicit modeling of the control loop leads to the Executable Runtime Megamodels (Eurema) approach [175]. Eurema is the predecessor modeling language and the starting point of the Deurema language described in this thesis. Eurema explicitly determines feedback loops as first class entities and defines modeling concepts to describe the order of adaptation effects by means of adaptation activities and control flow concepts. Furthermore, Eurema considers multi-layered adaptive architectures that facilitates the specification of meta-adaptation concepts. Beside the adaptation effects, Eurema uses runtime models for knowledge representation, where the adaptation activities can operate on the available knowledge base. An Eurema interpreter can directly execute the defined feedback loops by following the control flow of the defined adaptation activities and the trigger dependencies between different feedback loops. Unfortunately, Eurema focuses on single self-adaptive software systems. Therefore, it does neither consider the concurrent execution of feedback loops nor provides collaboration concepts to coordinate distributed feedback loops contained in multiple system instances. Furthermore, Eurema does not support the definition of partial runtime model views that is needed for sharing runtime information via system collaborations. Thus, the Deurema modeling language extends the Eurema concepts with respect to adaptive SoS modeling, which implies the need of considering distributed knowledge and system collaborations. Additionally, Deurema considers not only feedback loops for defining the adaptation capabilities of a system but rather supports multiple domains by providing different module templates as discussed in Section 5.3. With respect

of different systems types such as embedded and cyber-physical systems, Deurema provides static reconfiguration as well as dynamic adaptation concepts. Thereby, the Deurema analysis framework is able to reason about the modeled adaptive SoS architecture (cf. Chapter 6). Finally, the Deurema simulation environment directly executes the Deurema models as well as provides concepts of integrate analysis rules supporting runtime verification.

10.5. Discussion

In summary, this chapter shows that none of the discussed approaches above introduce a systematical modeling approach for adaptive SoS with collaborations. Deurema benefits from the ideas of these existing approaches above and integrates them into a new modeling language approach. The design of the Deurema modeling language is further influenced by the following own experience and publications. In [9], a first categorization of different runtime model types is proposed together with some ideas of analyzing dependencies of interacting feedback loops in an overall layered system architecture. Furthermore, the influence of uncertainty in runtime models is discussed in [1]. The vision of combining multiple, interacting feedback loops that hold runtime models as representation of the own system state and its context is presented in [4]. In this context, different real world scenarios and state of the art research approaches are compared in [11]. A first version of the Deurema modeling language, which comprises the collaboration between feedback loops, is published in [10].

Concerning the realization of Deurema concepts, the work in [8] describes the mapping of a component-based software description to the AUTOSAR standard. Furthermore, a mapping from AUTOSAR to timed automata for formal verification of interface interaction behavior is discussed in [7]. Finally, a comprehensive discussion about an embedded toolchain for the development of robotic systems following the MDE approach can be found in [14].

11. Conclusion

This chapter summarizes the contribution of this thesis with respect to the introduced goals in Chapter 1. Furthermore, it subsumes the Deurema modeling language concepts and compares it with the derived requirements for the adaptive SoS modeling in Chapter 3. Finally, the chapter outlines open points and possible future work.

11.1. Discussing Goals and Contribution

As motivated in the introduction, a research challenge for this thesis is the specification of an adaptive SoS behavior by explicitly modeling the adaptation logic (G1), integrating the available knowledge in form of runtime models (G2), and determining system interaction by means of collaborations (G3). On basis of these three goals, the main contribution of this thesis is the Deurema modeling language approach. Deurema is developed with respect to adaptive SoS characteristics as well as subsequently derived modeling language requirements, which are retrieved from the state of the art literature. Concerning goal G1, Deurema encapsulates the adaptive behavior in modules, where the internals are specified by a corresponding template description. Thereby, the specification of the adaptation logic follows different domain specific development concepts, which are feedback loop modeling, component-based modeling, and rule-based modeling. Furthermore, Deurema supports the adaptation and reconfiguration of system behavior as first class concepts.

Looking at the knowledge with respect to goal G2, this thesis proposes a runtime model categorization that defines the purpose of the runtime information without restricting the runtime model types (metamodel). Thus, runtime models from arbitrary domains are supported. Runtime models are maintained by a megamodel, which keeps track of occurring changes due to adaptation and manipulation effects. Furthermore, runtime models can be integrated in all three supported module template types by defining a view, which is locally available for the adaptation logic. The access to runtime models is specified by model as well as module operations that are maintained by the Deurema megamodel and can be directly executed.

The thesis goal G3 targets the modeling of collaborations. The Deurema modeling language introduces a structural collaboration specification, which comprises abstract roles and collaboration types. Furthermore, Deurema allows the modeling of the interaction behavior as well as defines role interfaces. Thereby, the above mentioned runtime model concept can be seamlessly integrated in the collaboration specification and used during system interactions for the exchange of runtime information. In a first step, collaboration behavior is separately defined from the local adaptation behavior, which facilitates the concurrent development of both distinct aspects and fosters separation of concerns. Afterwards, the specified collaboration behavior can be integrated into the overall adaptive SoS architecture.

Beside the explicit description of the adaptive SoS behavior, this thesis contributes to the analysis of emergent SoS behavior as emphasized by the thesis goal G4. Therefore, the Deurema analysis framework is introduced that supports the static analysis of the modeled adaptive SoS. This thesis introduces different metrics for investigating causal dependencies, the

distribution and access to runtime models, collaboration behavior, and appearing adaptation effects by means of the defined purpose. Furthermore, these basic metrics are combined to investigate complex relationships between modules and systems. In this context, architectural patterns as well as design flaws are outlined.

Beside the analysis, the Deurema simulation framework facilitates the direct execution of the modeled SoS as required by G5. Thereby, the Deurema simulator supports different scheduling strategies and the Deurema interpreter executes the modeled adaptation effects. Furthermore, an inference engine maintains all Deurema models as well as the available runtime models during the simulation. Combining the Deurema simulation and analysis capabilities contributes to a runtime analysis of the adaptive behavior.

The goal G6 targets the realization of an adaptive SoS specification. Therefore, this thesis describes a mapping of Deurema concepts to the state of the art AUTOSAR standard applied in the embedded domain for the development of modern cars. This mapping contributes to deriving an implementation of the adaptive behavior from the Deurema models. Additionally, state of the art software tools for the simulation and analysis of the implementation are outlined. Finally, this thesis describes the application of the Deurema modeling language in two state of the art case studies as well as to an EBCS approach showing that the Deurema concepts are powerful enough to cope with current research problems.

11.2. Modeling Language Requirements and Deurema

As shown at the bottom in Table 10.5 in the former chapter, Deurema integrates several ideas from different approaches and domains into one consistent modeling approach. An overview of the thesis goals and the realized modeling language requirements by the Deurema approach is sketched in Figure 11.1. Thereby, Deurema supports the explicit modeling of the adaptation logic in form of feedback loops (R-01) and the intra-loop coordination by means of the control flow as requested by R-02. The adaptive behavior is encapsulated in modules that are placed on the layered architecture of a system template. Furthermore, systems may comprise arbitrary other systems and modules, which forms the overall adaptive SoS. Modules can interact with each other at different ways. They can send event trigger, which is related to the causal order between module execution (R-04). Thereby, modules are considered as independent and possible distributed (R-05). Therefore, the Deurema collaboration concept must be used to define interactions between modules (R-03). For collaborations, Deurema has a clear notion of roles (R-07) that is an abstract entity that defines a piece of behavior within the interaction. Those roles must be assigned to modules or systems that play (realize) the interactions. Thereby, the collaboration participants can use the Deurema message concept (R-13) for synchronizing (R-14) local adaptation behavior at dedicated points in time or invoking remote functionality by means of services (R-06). The choreography specification aggregates an arbitrary number of interaction protocols (R-08) within the corresponding collaboration.

Focusing on the knowledge in the adaptive SoS, Deurema uses runtime models (R-09) that reflect key points of interest, which comprises the context of the system or the system itself. Deurema introduces a categorization for runtime models (R-10) that defines the purpose of the contained information. Furthermore, the separation of runtime model types and arbitrary views as well as the model operation concept for accessing the runtime model views facilitate the specification of partial knowledge usage (R-11) in the module instances. Of course, runtime models can be exchanged within a collaboration interaction (R-12) using the Deurema model

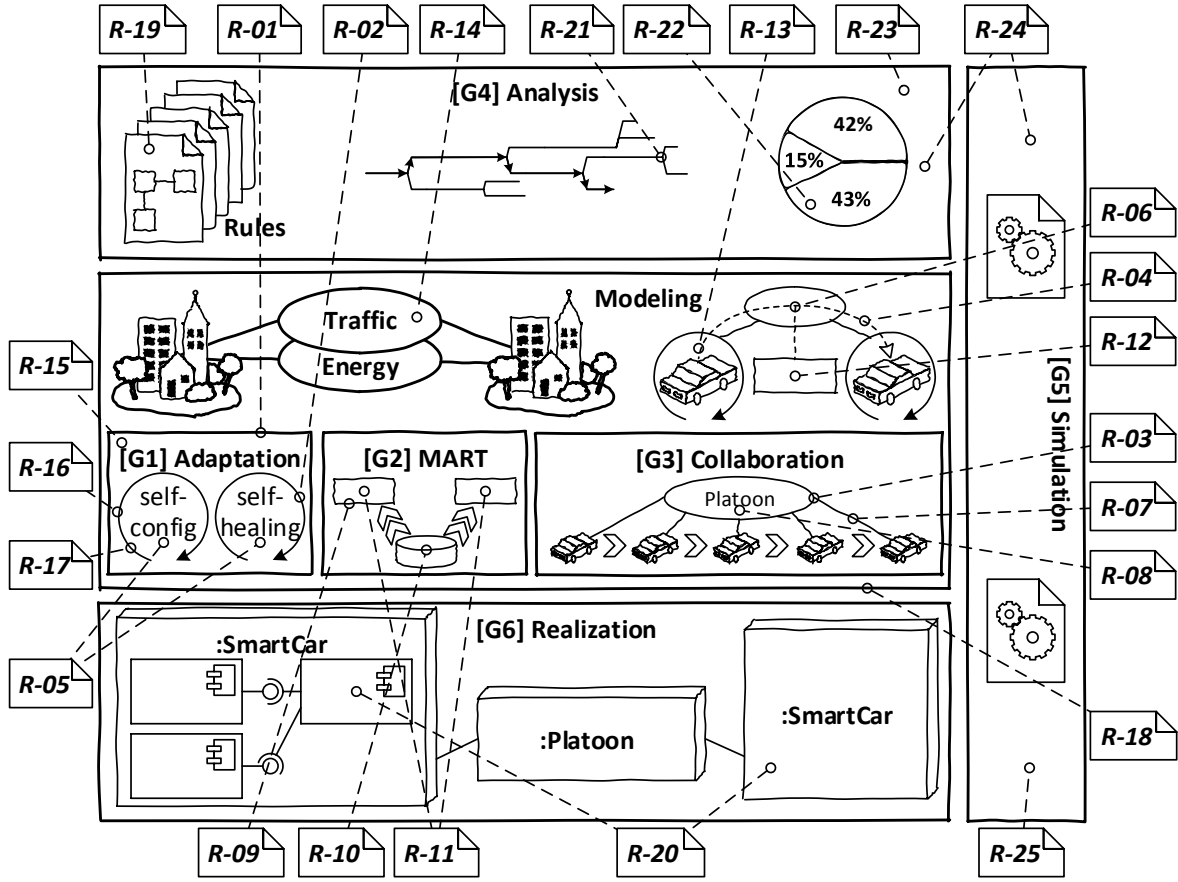


Figure 11.1: Overview of thesis goals and modeling language requirements

message concept. A runtime megamodel maintains the available knowledge base and all derived views during the simulation of the adaptive SoS.

The Deurema reflection mechanism enables the reasoning about module and system instances on different layers as well as facilitates their manipulation by affecting module operations. Therefore, this concept further enables the adaptation of the module behavior (R-16) as well as introduces meta-adaptation capabilities (R-17). Beside adaptation, the Deurema variability runtime model together with the variable concept in the module template definitions contributes to the reconfiguration modeling language requirement (R-15).

This thesis introduces different module template types that target distinct development concepts of defining the adaptation logic as it is requested by R-20. Furthermore, the discussed Deurema analysis framework introduces metrics (R-21, R-22) as well as architectural patterns (R-19) that can be statically (R-23) investigated within the modeled, adaptive SoS. Furthermore, the execution of the Deurema models enables the runtime analysis (R-24) as well as the simulation (R-25) of the adaptive SoS.

11.3. Future Work

Although the Deurema modeling language covers almost all derived requirements from Chapter 3, there are further language, analysis, and simulation extensions thinkable.

Domain Extensions

The Deurema modeling language supports three template types for the specification of the local adaptive behavior. Thereby, it covers the feedback loop modeling idea from the (self-) adaptive research domain, the component-based development paradigm, which is dominant for the development of embedded and cyber-physical systems, and the rule-based modeling by means of graph transformation rules as widely adopted in the MDE domain. Thereby, the supported module template types should not be seen as a fix, exhaustive set but rather as a meaningful subset targeting typical development paradigms for systems included in a SoS. However, the module template concept is developed in a way that further domain extensions could be introduced into the Deurema language. For example, if Deurema should support an agent-based module template type, the following steps must be applied. The abstract class `ModuleTemplate` in the Deurema metamodel (cf. Section 5.3) is the base class for all template types. Thus, a new subclass must be created, which corresponds to the new template type as for example a new `AgentModuleTemplate`. As a consequence, this new template type has automatically access to arbitrary runtime model views as defined by the super class. Therefore, the language developer has to define, which entities (e.g., agents) have access to the local runtime model views. Additionally, the language developer can inherit these new entities from the `BehaviorModel` class, which automatically refines the notion of black-gray-white box behavior that can be handled by the Deurema execution environment. Otherwise, the developer must extend the new template specific execution semantic in the interpreter and inference engine. Beside executable entities, the developer can define additional variable types that can be used for this template type. For example an agent variable may define a placeholder for one agent that can be replaced using the Deurema reconfiguration concept as comprehensively discussed in Section 5.6. For analysis, the developer must extend the given Deurema analysis rules to reason about the propagation of adaptation effects and the usage of knowledge. However, if another module template type is introduced in Deurema or not belongs to a design decision of the developer or community using this modeling language for specifying the adaptive behavior in their corresponding domain. As discussed in Chapter 9, the existing Deurema modeling language concepts are powerful enough to cope with a broad range of current systems.

Extended Tool Support and Patterns

Currently, the Deurema concepts are defined in the Eclipse Modeling Framework, which comprises an Ecore metamodel and a model-based editor. Furthermore, the analysis and simulation framework are implemented in Java on basis of the Deurema metamodel. However, using the Deurema modeling language may lead to best practices and beneficial design patterns over time. Collecting those patterns and describing advantages as well as drawbacks in a pattern catalog can improve the design of adaptive SoS architecture over time or may decrease the development time. In this context, tool extensions can offer such well-known patterns and may generate needed module, system, and template definitions, which further support the developer of specifying the adaptive behavior and the overall system architecture.

Beside the generation of initial model artifacts on basis of a pattern catalog, tool support for the realization of Deurema models, e.g., a code generator to a specific target platform, would

ease the implementation in the corresponding domain. Realizing such a model transformation from Deurema concepts to a target domain introduces overhead for the development of a code generator, but may decrease the implementation time of the target system afterwards.

Analysis and Simulation

The analysis rules discussed in Chapter 6 and enumerated in the Appendix C show how basic and complex metrics can be retrieved from the Deurema models. Depending on the underlying problem, new analysis rules can be specified to retrieve key points of interest. Conceptually, there are three kinds of analysis rules. First, the existing rule set can be extended for investigated other metrics on the Deurema models. Second, domain specific rules can investigate white box behavior, which is specified in form of graph transformation rules in Deurema. Thus, domain specific analysis rules may search for modeling guidelines or design flaws in the architecture, which depends on the underlying problem and cannot be foreseen in the context of this thesis. Third, the content of the available runtime models depends on the domain. Conceptually, Deurema can handle arbitrary runtime models, which is restricted by the implementation using the Eclipse Modeling Framework. However, because the content of the runtime models vary according to the domain, analysis rules that investigate the runtime models are not provided in the context of this thesis, but can be extended by defining new analysis rules. For example, if a domain specific runtime model represents the traffic situation of a smart city, an additional analysis rule may search for a traffic jam in the abstract runtime model representation. During a simulation, the traffic jams can be directly marked in the corresponding runtime model and reported to the user following the annotation concept as discussed in Chapter 6.

Beside the analysis, the Deurema simulation framework can be extended according different aspects. At first, a combination or round trip engineering of analysis and simulation of the adaptive SoS modeled with Deurema is thinkable. Therefore, the simulation must trace the order of executed elements. Afterwards, specialized analysis rules can investigate the simulation run to detect the source of found errors during the simulation. Thus, in this scenario, there is an interplay of running a simulation, analyzing the trace, fix errors in the model and running the simulation again.

Second, a simulation debugger similar to [6], which comprises fluently interactions with the user, would be beneficial towards the understanding the adaptive SoS behavior. A step-wise execution of interactions or the use of breakpoints directly in the Deurema models allow the direct investigation of the Deurema models, without looking on the concrete implementation. In combination with beforehand logged execution traces, a guided user simulation would ease the understanding of failures. Furthermore, the developer might stop the simulation, fix the error on the fly in the Deurema model and proceed the simulation without rerunning the complete trace, which can increase the productivity during the development process. The interpretation of Deurema models supports such a simulation debugger with dynamic changes of the underlying Deurema model during the simulation.

Third, there is a possible extension of the simulation framework by combining partial realized Deurema models in the concrete domain with a model-based simulation of other systems in the SoS. This requires a rethinking of the current simulation implementation towards a distributed simulation of systems within the SoS, where multiple simulator instances run on different hardware platforms and are coordinated for the overall simulation via well-defined interfaces. Having such a distributed simulation environment enables further combination with respect to timing properties. For example, some systems may be simulated on a host computer, where a logical simulation time is maintained, and other, already realized system

parts, can be simulated on the target platform having realistic execution times. Synchronizing such combinations of virtual and real-time simulations in an overall distributed simulation environment is very interesting for future work towards the understanding of the overall emergent and complex SoS behavior.

Bibliography

Author's References

- [1] Holger Giese, Nelly Bencomo, Liliana Pasquale, Andres J. Ramirez, Paola Inverardi, Sebastian Wätzoldt, and Siobhan Clarke. “Living with Uncertainty in the Age of Runtime Models”. In: *Models@run.time*. Vol. 8378. LNCS. Springer, 2014, pp. 47–100.
- [2] Holger Giese, Stephan Hildebrandt, Stefan Neumann, and Sebastian Wätzoldt. *Industrial Case Study on the Integration of SysML and AUTOSAR with Triple Graph Grammars*. Tech. rep. 57. Hasso Plattner Institute for IT-Systems Engineering at the University of Potsdam, Sept. 2012.
- [3] Holger Giese, Leen Lambers, Basil Becker, Stephan Hildebrandt, Stefan Neumann, Thomas Vogel, and Sebastian Wätzoldt. “Graph Transformations for MDE, Adaptation, and Models at Runtime”. In: *Proceedings of the 12th International Conference on Formal Methods for Model-Driven Engineering*. Vol. 7320. LNCS. Springer, June 2012, pp. 137–191.
- [4] Holger Giese, Thomas Vogel, and Sebastian Wätzoldt. “Towards Smart Systems of Systems”. In: *6th IPM International Conference on Fundamentals of Software Engineering*. LNCS. Springer, 2015.
- [5] Edgar Jakumeit, Sebastian Buchwald, Dennis Wagelaar, Li Dan, Ábel Hegedüs, Markus Herrmannsdörfer, Tassilo Horn, Elina Kalnina, Christian Krause, Kevin Lano, Markus Lepper, Arend Rensink, Louis Rose, Sebastian Wätzoldt, and Steffen Mazanek. “A Survey and Comparison of Transformation Tools based on the Transformation Tool Contest”. In: *Science of Computer Programming* 85.1 (June 2014), pp. 41–99.
- [6] Alexander Krasnogolowy, Stephan Hildebrandt, and Sebastian Wätzoldt. “Flexible Debugging of Behavior Models”. In: *International Conference on Industrial Technology, ICIT*. IEEE, Mar. 2012, pp. 331–336.
- [7] Stefan Neumann, Norman Kluge, and Sebastian Wätzoldt. “Automatic Transformation of Abstract AUTOSAR Architectures to Timed Automata”. In: *Proceedings of the 5th International Workshop on Model Based Architecting and Construction of Embedded Systems*. ACES-MB. ACM, 2012, pp. 55–60.
- [8] Stefan Neumann, Sebastian Wätzoldt, and Holger Giese. “From Abstract Component Descriptions to Timed I/O-Interfaces in AUTOSAR”. In: *Proceeding of the Second Analytic Virtual Integration of Cyber-Physical Systems Workshop*. AVICPS, Nov. 2011, pp. 25–32.
- [9] Sebastian Wätzoldt and Holger Giese. “Classifying Distributed Self-* Systems Based on Runtime Models and Their Coupling”. In: *Proceedings of the 9th Workshop on Models@run.time co-located with 17th International Conference on Model Driven Engineering Languages and Systems*. CEUR-WS, Sept. 2014, pp. 11–20.

- [10] Sebastian Wätzoldt and Holger Giese. “Modeling Collaborations in Adaptive Systems of Systems”. In: *Proceedings of the European Conference on Software Architecture Workshops*. ECSAW. ACM, 2015.
- [11] Sebastian Wätzoldt and Holger Giese. *Modeling Collaborations in Self-Adaptive Systems of Systems—Terms, Characteristics, Requirements, and Scenarios*. Tech. rep. 96. Hasso Plattner Institute for IT-Systems Engineering at the University of Potsdam, Apr. 2015.
- [12] Sebastian Wätzoldt, Stephan Hildebrandt, Holger Giese, and Axel Uhl. *Towards Scalable and Self-Optimizing Software for Multi-Core and Cloud Computing II*. Tech. rep. 70. Hasso Plattner Institute for IT-Systems Engineering at the University of Potsdam, 2011.
- [13] Sebastian Wätzoldt, Stephan Hildebrandt, Andreas Seibel, Gregor Gabrysiak, and Holger Giese. *Towards Scalable and Self-Optimizing Software for Multi-Core and Cloud Computing*. Tech. rep. 42. Hasso Plattner Institute for IT-Systems Engineering at the University of Potsdam, Feb. 2010.
- [14] Sebastian Wätzoldt, Stefan Neumann, Falk Benke, and Holger Giese. “Integrated Software Development for Embedded Robotic Systems”. In: *Proceedings of the 3rd International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Vol. 7628. LNCS. Springer, Oct. 2012, pp. 335–348.

Other References

- [15] Frederico Alvares de Oliveira, Remi Sharrock, and Thomas Ledoux. “Synchronization of Multiple Autonomic Control Loops: Application to Cloud Computing”. In: *Proceedings of the 14th International Conference on Coordination Models and Languages*. Vol. 7274. COORDINATION. Springer, 2012, pp. 29–43.
- [16] Jesper Andersson, Luciano Baresi, Nelly Bencomo, Rogério de Lemos, Alessandra Gorla, Paola Inverardi, and Thomas Vogel. “Software Engineering Processes for Self-Adaptive Systems”. In: *Software Engineering for Self-Adaptive Systems II*. Vol. 7475. LNCS. Springer, Jan. 2013, pp. 51–75.
- [17] Richard Anthony, Achim Rettberg, Dejiu Chen, Isabell Jahnich, Gerrit Boer, and Cecilia Ekelin. “Towards a Dynamically Reconfigurable Automotive Control System Architecture”. In: *Embedded System Design: Topics, Techniques and Trends*. Vol. 231. IFIP. Springer, 2007, pp. 71–84.
- [18] Patricia Seefelder Assis, Daniel Schwabe, and Demetrius Arraes Nunes. “ASHDM – Model-Driven Adaptation and Meta-Adaptation”. In: *Adaptive Hypermedia and Adaptive Web-Based Systems*. Vol. 4018. LNCS. Springer, June 2006, pp. 213–222.
- [19] Uwe Aßmann, Sebastian Götz, Jean-Marc Jézéquel, Brice Morin, and Mario Trapp. “A Reference Architecture and Roadmap for Models@run.time Systems”. In: *Models@run.time*. LNCS. Springer, 2014, pp. 1–18.
- [20] Luigi Atzori, Antonio Iera, and Giacomo Morabito. “The Internet of Things: A Survey”. In: *Computer Networks* 54.15 (Oct. 2010), pp. 2787–2805.
- [21] Robert Baillargeon. “Vehicle System Development: A Challenge of Ultra-Large-Scale Systems”. In: *Proceedings of the International Workshop on Software Technologies for Ultra-Large-Scale Systems*. ULS. IEEE, May 2007.

- [22] Mikael Barbero, Marcos Didonet Fabro, and Jean Bézivin. “Traceability and Provenance Issues in Global Model Management”. In: *Proceedings of 3rd Workshop on Traceability*. ECMDA-TW. SINTEF, June 2007, pp. 47–55.
- [23] Roberto Barbuti and Luca Tesei. “Timed Automata with Urgent Transitions”. In: *Acta Informatica* 40.5 (2004), pp. 317–347.
- [24] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. “Toward Open-World Software: Issue and Challenges”. In: *Computer* 39.10 (2006), pp. 36–43.
- [25] Luciano Baresi, Sam Guinea, and Adnan Shahzada. “SeSaMe: Towards a Semantic Self Adaptive Middleware for Smart Spaces”. In: *Engineering Multi-Agent Systems*. LNCS. Springer, 2013, pp. 1–18.
- [26] Luciano Baresi, Reiko Heckel, Sebastian Thöne, and Dániel Varró. “Style-based Modeling and Refinement of Service-oriented Architectures”. In: *Software and Systems Modeling* 5.2 (2006), pp. 187–207.
- [27] Jörg Bauer and Reinhard Wilhelm. “Static Analysis of Dynamic Communication Systems by Partner Abstraction”. In: *Static Analysis*. Vol. 4634. LNCS. Springer, 2007, pp. 249–264.
- [28] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. “Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation”. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE. ACM, 2006, pp. 72–81.
- [29] Basil Becker and Holger Giese. “On Safe Service-Oriented Real-Time Coordination for Autonomous Vehicles”. In: *Proceedings of the 11th International Symposium on Object Oriented Real-Time Distributed Computing*. ISORC. IEEE, May 2008, pp. 203–210.
- [30] Basil Becker and Holger Giese. *Modeling and Verifying Dynamic Evolving Service-Oriented Architectures*. Tech. rep. 75. Hasso Plattner Institute for IT-Systems Engineering at the University of Potsdam, 2013.
- [31] Basil Becker, Holger Giese, Stefan Neumann, Martin Schenck, and Arian Treffer. “Model-Based Extension of AUTOSAR for Architectural Online Reconfiguration”. In: *Models in Software Engineering*. Vol. 6002. LNCS. Springer, Oct. 2010, pp. 83–97.
- [32] Nelly Bencomo. “On the Use of Software Models during Software Execution”. In: *Proceedings of the ICSE Workshop on Modeling in Software Engineering*. MISE. IEEE, 2009, pp. 62–67.
- [33] Nelly Bencomo, Jon Whittle, Pete Sawyer, Anthony Finkelstein, and Emmanuel Letier. “Requirements Reflection: Requirements As Runtime Entities”. In: *Proceedings of the 32nd International Conference on Software Engineering*. ICSE. ACM, 2010, pp. 199–202.
- [34] Johan Bengtsson and Wang Yi. “Timed Automata: Semantics, Algorithms and Tools”. In: *Lectures on Concurrency and Petri Nets*. Vol. 3098. Lecture Notes in Computer Science. Springer, 2004, pp. 87–124.
- [35] Thomas Beyhl and Holger Giese. *Efficient and Scalable Graph View Maintenance for Deductive Graph Databases based on Generalized Discrimination Networks*. Tech. rep. Hasso Plattner Institute for IT-Systems Engineering at the University of Potsdam, 2015.

- [36] Jean Bézevin. “On the Unification Power of Models”. In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188.
- [37] Jean Bézevin, Sébastien Gérard, Pierre-Alain Muller, and Laurent Rioux. “MDA components: Challenges and Opportunities”. In: *First International Workshop on Metamodelling for MDA*. LINA, Nov. 2003, pp. 23–41.
- [38] Gordon Blair, Nelly Bencomo, and Robert B. France. “Models@run.time”. In: *Computer* 42.10 (2009), pp. 22–27.
- [39] John Boardman and Brian Sauser. “System of Systems - The Meaning of of”. In: *International Conference on System of Systems Engineering*. IEEE, Apr. 2006, pp. 1–6.
- [40] Birgit Bomsdorf, Stefan Grau, Martin Hudasch, and Jan-Torsten Milde. “Configurable Executable Task Models Supporting the Transition from Design Time to Runtime”. In: *Human-Computer Interaction. Design and Development Approaches*. Vol. 6761. LNCS. Springer, 2011, pp. 155–164.
- [41] Mélanie Bouroche, Barbara Hughes, and Vinny Cahill. “Real-time Coordination of Autonomous Vehicles”. In: *Intelligent Transportation Systems Conference*. IEEE, Sept. 2006, pp. 1232–1239.
- [42] Bruno Bouyssounouse and Joseph Sifakis. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*. Vol. 3436. LNCS. Springer, 2005.
- [43] Bart Broekman and Edwin Notenboom. *Testing Embedded Software*. Addison Wesley, 2003.
- [44] Alan W. Brown. “Model Driven Architecture: Principles and Practice”. In: *Software and Systems Modeling* 3.4 (Dec. 2004), pp. 314–327.
- [45] Manfred Broy, María Victoria Cengarle, and Eva Geisberger. “Cyber-Physical Systems: Imminent Challenges”. In: *Large-Scale Complex IT Systems. Development, Operation and Management*. Vol. 7539. LNCS. Springer, 2012, pp. 1–28.
- [46] Manfred Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. “Engineering Automotive Software”. In: *Proceedings of the IEEE* 95.2 (Feb. 2007), pp. 356–373.
- [47] Davide Brugali and Patrizia Scandurra. “Component-Based Robotic Rngineering Part I: Reusable Building Blocks”. In: *Robotics Automation Magazine* 16.4 (Dec. 2009), pp. 84–96.
- [48] Davide Brugali and Azamat Shakhimardanov. “Component-Based Robotic Engineering Part II: Systems and Models”. In: *Robotics Automation Magazine* 17.1 (Mar. 2010), pp. 100–112.
- [49] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi A. Müller, Mauro Pezzè, and Mary Shaw. “Engineering Self-Adaptive Systems through Feedback Loops”. In: *Software Engineering for Self-Adaptive Systems*. Vol. 5525. LNCS. Springer, 2009, pp. 48–70.
- [50] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. “DEECO: An Ensemble-based Component System”. In: *Proceedings of the 16th International Symposium on Component-based Software Engineering*. CBSE. ACM, 2013, pp. 81–90.

- [51] Sven Burmester and Holger Giese. “Visual Integration of UML 2.0 and Block Diagrams for Flexible Reconfiguration in Mechatronic UML”. In: *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. VL/HCC. IEEE, Sept. 2005, pp. 109–116.
- [52] Sven Burmester, Holger Giese, Eckehard Münch, Oliver Oberschelp, Florian Klein, and Peter Scheideler. “Tool Support for the Design of Self-Optimizing Mechatronic Multi-Agent Systems”. In: *International journal on Software Tools for Technology Transfer* 10.3 (June 2008), pp. 207–222.
- [53] Sven Burmester, Holger Giese, and Matthias Tichy. “Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML”. In: *Proceedings of the 2003 European Conference on Model Driven Architecture: Foundations and Applications*. MDFA. Springer, 2005, pp. 47–61.
- [54] Roland Burns. *Advanced Control Engineering*. Butterworth-Heinemann, 2001.
- [55] Radu Calinescu, Simos Gerasimou, and Alec Banks. “Self-Adaptive Software with Decentralised Control Loops”. In: *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering*. FASE. Springer, 2015, pp. 1–15.
- [56] Radu Calinescu, Lars Grunske, Marta Kwiatkowska, Raffaella Mirandola, and Giordano Tamburrelli. “Dynamic QoS Management and Optimization in Service-Based Systems”. In: *Transactions on Software Engineering* 37.3 (May 2011), pp. 387–409.
- [57] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. “Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes”. In: *Computer* 42.10 (2009), pp. 37–43.
- [58] Betty H. Cheng et al. “Software Engineering for Self-Adaptive Systems: A Research Roadmap”. In: *Software Engineering for Self-Adaptive Systems*. Vol. 5525. LNCS. Springer, 2009, pp. 1–26.
- [59] Jong-Seok Choi, T. McCarthy, M. Yadav, Minyoung Kim, Carolyn Talcott, and E. Gressier-Soudan. “Application Patterns for Cyber-Physical Systems”. In: *1st International Conference on Cyber-Physical Systems, Networks, and Applications*. CPSNA. IEEE, Aug. 2013, pp. 52–59.
- [60] Paul C. Clements. “A Survey of Architecture Description Languages”. In: *Proceedings of the 8th International Workshop on Software Specification and Design*. IWSSD. IEEE, Mar. 1996, pp. 16–25.
- [61] AUTOSAR Consortium. *AUTOSAR EXP LayeredSoftwareArchitecture.pdf*. Version 4.0, page id: 94ju5, <http://www.autosar.org/> visited: 09th of January 2016. 2015.
- [62] AUTOSAR Consortium. *AUTOSAR Specification*. Version 4.2, <http://www.autosar.org/specifications/> visited: 07th of October 2015. 2015.
- [63] World Wide Web Consortium. *SPARQL Protocol And RDF Query Language (SPARQL), Overview*. Version 1.1, <http://www.w3.org/TR/sparql11-overview/> visited: 13th of December 2015. Mar. 2013.
- [64] Y. Correa and Charles Keating. “An Approach to Model Formulation for Systems of Systems”. In: *International Conference on Systems, Man and Cybernetics*. Vol. 4. IEEE, Oct. 2003, pp. 3553–3558.

- [65] Massimo Cossentino, Nicolas Gaud, Vincent Hilaire, Stéphane Galland, and Abderrafiâa Koukam. “ASPECS: An Agent-oriented Software Process for Engineering Complex Systems”. In: *Autonomous Agents and Multi-Agent Systems* 20.2 (2010), pp. 260–304.
- [66] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel R.V. Chaudron. “A Classification Framework for Software Component Models”. In: *Transactions on Software Engineering* 37.5 (Sept. 2011), pp. 593–615.
- [67] Anind K. Dey. “Understanding and Using Context”. In: *Personal Ubiquitous Computing* 5.1 (Jan. 2001), pp. 4–7.
- [68] Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. “A Journey to Highly Dynamic, Self-Adaptive Service-Based Applications”. In: *Automated Software Engineering* 15.3 (Dec. 2008), pp. 313–341.
- [69] acatech (Ed.) *Cyber-Physical Systems: Driving force for innovation in mobility, health, energy and production*. Springer, 2011.
- [70] George Edwards, Joshua Garcia, Hossein Tajalli, Daniel Popescu, Nenad Medvidovic, Sukhat Gaurav, and Brad Petrus. “Architecture-driven Self-adaptation and Self-management in Robotics Systems”. In: *Proceedings of the Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS. IEEE, May 2009, pp. 142–151.
- [71] European Telecommunications Standards Institute. *Intelligent Transportation Systems (ITS); Vehicular Communications; Basic Set of Applications; Definitions*. Tech. rep. V1.1.1. European Telecommunications Standards Institute, June 2009.
- [72] Jean-Marie Favre. “Foundations of Model (Driven) (Reverse) Engineering : Models – Episode I: Stories of The Fidus Papyrus and of The Solarus”. In: *Language Engineering for Model-Driven Software Development*. Dagstuhl Seminar Proceedings 04101. IBFI, 2005.
- [73] Franck Fleurey and Arnor Solberg. “A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems”. In: *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*. Vol. 5795. MODELS. Springer, 2009, pp. 606–621.
- [74] Francois Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. “Kevoree Modeling Framework (KMF): Efficient Modeling Techniques for Runtime Use”. In: *CoRR* 1405.6817 (2014).
- [75] Robert B. France and Bernhard Rumpe. “Model-driven Development of Complex Software: A Research Roadmap”. In: *Future of Software Engineering*. FOSE. IEEE, 2007, pp. 37–54.
- [76] Sylvain Frey, Ada Diaconescu, David Menga, and Isabelle Demeure. “A Generic Holonic Control Architecture for Heterogeneous Multi-Scale and Multi-Objective Smart Micro-Grids”. In: *Transactions on Autonomous and Adaptive Systems*. SASO 10.2 (June 2015), pp. 1–20.
- [77] Cristina Gacek, Holger Giese, and Ethan Hadar. “Friends or Foes?: A Conceptual Analysis of Self-Adaptation and IT Change Management”. In: *Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS. ACM, May 2008, pp. 121–128.

- [78] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 34th ed. Addison-Wesley, Mar. 2007.
- [79] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. “Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure”. In: *Computer* 37.10 (2004), pp. 46–54.
- [80] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. “Self-Organising Software Architectures for Distributed Systems”. In: *Proceedings of the First Workshop on Self-healing Systems*. WOSS. ACM, 2002, pp. 33–38.
- [81] T. Gezgin, C. Etzien, Stefan Henkler, and Achim Rettberg. “Towards a Rigorous Modeling Formalism for Systems of Systems”. In: *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. ISORCW. IEEE, Apr. 2012, pp. 204–211.
- [82] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. “Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration”. In: *Proceedings of the 12th International Symposium on Foundations of Software Engineering*. SIGSOFT/FSE. ACM, Nov. 2004, pp. 179–188.
- [83] Holger Giese and Wilhelm Schäfer. “Model-Driven Development of Safe Self-Optimizing Mechatronic Systems with MechatronicUML”. In: *Assurances for Self-Adaptive Systems*. Vol. 7740. LNCS. Springer, Jan. 2013, pp. 152–186.
- [84] Didac Gil de la Iglesia and Danny Weyns. “Guaranteeing Robustness in a Mobile Learning Application Using Formally Verified MAPE Loops”. In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS. IEEE, 2013, pp. 83–92.
- [85] Heather J. Goldsby, Betty H. Cheng, and Ji Zhang. “AMOEBa-RT: Run-Time Verification of Adaptive Software”. In: *Models in Software Engineering*. Springer, 2008, pp. 212–224.
- [86] Sebastian Götz, Ilias Gerostathopoulos, Filip Krikava, Adnan Shahzada, and Romina Spalazzese. “Adaptive Exchange of Distributed Partial Models@Run.Time for Highly Dynamic Systems”. In: *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS. IEEE, 2015, pp. 64–70.
- [87] Object Management Group. *Unified Modeling Language (UML), Superstructure*. Version 2.4.1, <http://www.omg.org/spec/UML/2.4.1/> visited: 08th of December 2014. 2011.
- [88] Object Management Group. *Service oriented architecture Modeling Language (SoaML)*. Version 1.0.1, <http://www.omg.org/spec/SoaML/1.0.1/> visited: 08th of December 2014. 2012.
- [89] Object Management Group. *Systems Modeling Language (SysML)*. Version 1.3, <http://www.omg.org/spec/SysML/> visited: 19th of February 2015. 2012.
- [90] Object Management Group. *MDA Guide revision 2.0*. <http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf> visited: December 2014. June 2014.
- [91] Object Management Group. *Foundational Unified Modeling Language (fUML)*. Version 1.2.1, <http://www.omg.org/spec/FUML/1.2.1/> visited: 03rd of February 2016. Jan. 2016.

- [92] Ning Gui and Vincenzo De Florio. “Towards Meta-Adaptation Support with Reusable and Composable Adaptation Components”. In: *6th International Conference on Self-Adaptive and Self-Organizing Systems*. SASO. IEEE, Sept. 2012, pp. 49–58.
- [93] Ning Gui, Vincenzo De Florio, Hong Sun, and Chris Blondia. “A Hybrid Real-time Component Model for Reconfigurable Embedded Systems”. In: *Proceedings of the Symposium on Applied Computing*. SAC. ACM, 2008, pp. 1590–1596.
- [94] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. “Dynamic Software Product Lines”. In: *Computer* 41.4 (Apr. 2008), pp. 93–95.
- [95] Bo Han, Weijia Jia, Ji Shen, and Man-Ching Yuen. “Context-Awareness in Mobile Web Services”. In: *Parallel and Distributed Processing and Applications*. Vol. 3358. LNCS. Springer, 2005, pp. 519–528.
- [96] Regina Hebig, Holger Giese, and Basil Becker. “Making Control Loops Explicit when Architecting Self-adaptive Systems”. In: *Proceedings of the Second International Workshop on Self-organizing Architectures*. SOAR. ACM, June 2010, pp. 21–28.
- [97] Regina Hebig, Andreas Seibel, and Holger Giese. “On the Unification of Megamodels”. In: *Proceedings of the 4th International Workshop on Multi-Paradigm Modeling*. Vol. 42. Electronic Communications of the EASST. EASST, 2011.
- [98] Thomas A. Henzinger and Joseph Sifakis. “The Embedded Systems Design Challenge”. In: *14th International Symposium on Formal Methods*. Vol. 4085. LNCS. Springer, 2006, pp. 1–15.
- [99] Jamie Hillman and Ian Warren. “Meta-Adaptation in Autonomic Systems”. In: *Proceedings of the 10th International Workshop on Future Trends of Distributed Computing Systems*. FTDCS. IEEE, May 2004, pp. 292–298.
- [100] Martin Hirsch, Stefan Henkler, and Holger Giese. “Modeling Collaborations with Dynamic Structural Adaptation in Mechatronic UML”. In: *Proceedings of the International Workshop on Software Engineering for Adaptive and Self-managing Systems*. SEAMS. ACM, 2008, pp. 33–40.
- [101] John H. Holland. *Hidden Order: How Adaptation Builds Complexity*. Addison Wesley, 1996.
- [102] Matthias Hölzl and Martin Wirsing. “Towards a System Model for Ensembles”. In: *Formal Modeling: Actors, Open Systems, Biological Systems*. Vol. 7000. LNCS. Springer, 2011, pp. 241–261.
- [103] M. Usman Iftikhar and Danny Weyns. “ActivFORMS: Active Formal Models for Self-adaptation”. In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS. ACM, 2014, pp. 125–134.
- [104] European Research Cluster on the Internet of Things. *Internet of Things: IoT Semantic Interoperability: Research Challenges, Best Practices, Recommendations and Next Steps*. Mar. 2015.
- [105] Ethan K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. “Components, Platforms and Possibilities: Towards Generic Automation for MDA”. In: *Proceedings of the 10th International Conference on Embedded Software*. EMSOFT. ACM, 2010, pp. 39–48.

-
- [106] Mohammad Jamshidi, ed. *System of Systems Engineering: Innovations for the 21st Century*. Systems Engineering and Management. Wiley, Nov. 2008.
 - [107] Xiong Jian, Ge Bing-feng, Zhang Xiao-ke, Yang Ke-wei, and Chen Ying-Wu. “Evaluation Method of System-of-Systems Architecture using Knowledge-based Executable Model”. In: *International Conference on Management Science and Engineering*. ICMSE. IEEE, Nov. 2010, pp. 141–147.
 - [108] YC Jiang, ZY Xia, YP Zhong, and SY Zhang. “An Adaptive Adjusting Mechanism for Agent Distributed Blackboard Architecture”. In: *Microprocessors and Microsystems* 29.1 (Feb. 2005), pp. 9–20.
 - [109] Jin Jing, Abdelsalam Sumi Helal, and Ahmed Elmagarmid. “Client-Server Computing in Mobile Environments”. In: *Computing Surveys* 31.2 (June 1999), pp. 117–157.
 - [110] Jeffrey O. Kephart and David Chess. “The Vision of Autonomic Computing”. In: *Computer* 36.1 (Jan. 2003), pp. 41–50.
 - [111] Nil Kilicay-Ergin and Cihan Dagli. “Executable Modeling for System of Systems Architecting: An Artificial Life Framework”. In: *Proceedings of the 2nd Systems Conference*. IEEE, Apr. 2008, pp. 1–5.
 - [112] Kyoung-Dae Kim and P.R. Kumar. “Cyber-Physical Systems: A Perspective at the Centennial”. In: *Proceedings of the IEEE* 100.1 (May 2012), pp. 1287–1308.
 - [113] Minyoung Kim, Mark-Oliver Stehr, Jinwoo Kim, and Soonhoi Ha. “An Application Framework for Loosely Coupled Networked Cyber-Physical Systems”. In: *8th International Conference on Embedded and Ubiquitous Computing*. IEEE, Dec. 2010, pp. 144–153.
 - [114] Florian Klein and Matthias Tichy. “Building Reliable Systems based on Self-Organizing Multi-Agent Systems”. In: *Proceedings of the 5th Workshop on Software Engineering for Large-Scale Multi-Agent Systems*. SELMAS. ACM, May 2006, pp. 51–58.
 - [115] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.
 - [116] Verena Klös, Thomas Göthel, and Sabine Glesner. “Adaptive Knowledge Bases in Self-Adaptive System Design”. In: *Proceedings of the 41st Euromicro Conference on Software Engineering and Advanced Applications*. SEAA. IEEE, Aug. 2015, pp. 472–478.
 - [117] Mieczyslaw M. Kokar, Kenneth Baclawski, and Yonet A. Eracar. “Control Theory-Based Foundations of Self-Controlling Software”. In: *Intelligent Systems* 14.3 (May 1999), pp. 37–45.
 - [118] Birgit Korherr and Beate List. “A UML 2 Profile for Variability Models and their Dependency to Business Processes”. In: *Proceedings of the 18th International Workshop on Database and Expert Systems Applications*. DEXA. IEEE, Sept. 2007, pp. 829–834.
 - [119] Jeff Kramer and Jeff Magee. “Self-Managed Systems: an Architectural Challenge”. In: *Proceedings of Future of Software Engineering*. FOSE. IEEE, May 2007, pp. 259–268.
 - [120] Fatma Krichen, Brahim Hamid, Bechir Zalila, and Mohamed Jmaiel. “Towards a Model-based Approach for Reconfigurable DRE Systems”. In: *Proceedings of the 5th European Conference on Software Architecture*. ECSA. Springer, 2011, pp. 295–302.
-

- [121] Anette J. Krygiel. *Behind the Wizard's Curtain: An Integration Environment for a System of Systems*. National Defense University Press, June 1999.
- [122] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.
- [123] Edward A. Lee. "Cyber-Physical Systems - Are Computing Foundations Adequate?" In: *Position Paper for NSF Workshop on Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*. National Science Foundation, Oct. 2006.
- [124] Edward A. Lee. "Cyber Physical Systems: Design Challenges". In: *11th International Symposium on Object Oriented Real-Time Distributed Computing*. ISORC. IEEE, May 2008, pp. 363–369.
- [125] Edward A. Lee. "CPS Foundations". In: *Proceedings of the 47th Design Automation Conference*. DAC. ACM, 2010, pp. 737–742.
- [126] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. <http://leeseshia.org>, 2011.
- [127] Rogério de Lemos et al. "Software Engineering for Self-Adaptive Systems: A second Research Roadmap". In: *Software Engineering for Self-Adaptive Systems II*. Vol. 7475. LNCS. Springer, Jan. 2013, pp. 1–32.
- [128] Jochen Ludewig. "Models in Software Engineering: An Introduction". In: *Software and Systems Modeling 2.1* (2003), pp. 5–14.
- [129] Pattie Maes. "Concepts and Experiments in Computational Reflection". In: *Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA. ACM, 1987, pp. 147–155.
- [130] Pedro Maia, Everton Cavalcante, Porfírio Gomes, Thais Batista, Flavia C. Delicato, and Paulo F. Pires. "On the Development of Systems-of-Systems Based on the Internet of Things: A Systematic Mapping". In: *Proceedings of the European Conference on Software Architecture Workshops*. ECSAW. ACM, 2014, pp. 1–8.
- [131] Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. "A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems". In: *Component Deployment*. Vol. 3798. LNCS. Springer, 2005, pp. 99–114.
- [132] Radu Manuca, Yi Li, Rick Riolo, and Robert Savit. *The Structure of Adaptive Competition in Minority Games*. Tech. rep. PSCS-98-11-001. Cornell University, 1998.
- [133] A. Marconi, Antonio Bucchiarone, K. Bratanis, Antonio Brogi, Javier Camara, D. Dranidis, Holger Giese, R. Kazhamiakink, Rogério de Lemos, C.C. Marquezan, and Andreas Metzger. "Research Challenges on Multi-layer and Mixed-initiative Monitoring and Adaptation for Service-based Systems". In: *Workshop on European Software Services and Systems Research - Results and Challenges (S-Cube)*. IEEE, June 2012, pp. 40–46.
- [134] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. "Composing Adaptive Software". In: *Computer* 37.7 (July 2004), pp. 56–64.
- [135] Andreas Metzger and Klaus Pohl. "Software Product Line Engineering and Variability Management: Achievements and Challenges". In: *Proceedings of the on Future of Software Engineering*. FOSE. ACM, 2014, pp. 70–84.

- [136] Ronald Miller and Qingfeng Huang. “An Adaptive Peer-to-Peer Collision Warning System”. In: *Proceedings of the 55th Vehicular Technology Conference*. Vol. 1. IEEE, 2002, pp. 317–321.
- [137] Saurabh Mittal and José Luis Risco Martín. “Model-driven Systems Engineering for Netcentric System of Systems with DEVS Unified Process”. In: *Proceedings of the Simulation Conference*. WSC. IEEE, Dec. 2013, pp. 1140–1151.
- [138] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. “Models@Run.Time to Support Dynamic Adaptation”. In: *Computer* 42.10 (Oct. 2009), pp. 44–51.
- [139] Ron Morrison, Graham Kirby, Dharini Balasubramaniam, Kath Mickan, Flavio Oquendo, Sorana Cîmpan, Brian Warboys, Bob Snowdon, and R. Mark Greenwood. “Support for Evolving Software Architectures in the ArchWare ADL”. In: *Proceedings of the 4th Working Conference on Software Architecture*. WICSA. IEEE, June 2004, pp. 69–78.
- [140] Georgios Moschoglou, Timothy Eveleigh, Thomas Holzer, and Shahryar Sarkani. “An Approach to Semantic Interoperability in Federations of Systems”. In: *International Journal of System of Systems Engineering* 4.1 (2013), pp. 79–97.
- [141] David J. Musliner, Robert P. Goldman, Michael J. Pelican, and Kurt D. Krebsbach. “Self-Adaptive Software for Hard Real-Time Environments”. In: *Intelligent Systems and their Applications* 14.4 (July 1999), pp. 23–29.
- [142] Linda Northrop, Peter H. Feiler, Richard P. Gabriel, Rick Linger, Tom Longstaff, Rick Kazman, Markus Klein, and Douglas Schmidt. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University, 2006.
- [143] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. “An Architecture-Based Approach to Self-Adaptive Software”. In: *Intelligent Systems* 14.3 (May 1999), pp. 54–62.
- [144] Charles L. Ortiz, Régis Vincent, and Benoit Morisset. “Task Inference and Distributed Task Management in the Centibots Robotic System”. In: *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS. ACM, 2005, pp. 860–867.
- [145] H. Van Dyke Parunak. “A Survey of Environments and Mechanisms for Human-Human Stigmergy”. In: *Environments for Multi-Agent Systems II*. Vol. 3830. LNCS. Springer, 2006, pp. 163–186.
- [146] H. Van Dyke Parunak, Sven Brueckner, Mitch Fleischer, and James Odell. “Co-X: Defining what Agents Do Together”. In: *Workshop on Team and Coalition Formation*. AAMAS. ACM, July 2002.
- [147] H. Van Dyke Parunak, Sven Brueckner, Mitch Fleischer, and James Odell. “A Preliminary Taxonomy of Multi-Agent Interactions”. In: *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS. ACM, 2003, pp. 1090–1091.

- [148] H. Van Dyke Parunak, Sven Brueckner, Mitch Fleischer, and James Odell. “A Design Taxonomy of Multi-Agent Interactions”. In: *Agent-Oriented Software Engineering IV*. Vol. 2935. LNCS. Springer, 2004, pp. 123–137.
- [149] Christian Piechnick, Sebastian Richly, Thomas Kühn, Sebastian Götz, Georg Püschel, and Uwe Aßmann. “ContextPoint: An Architecture for Extrinsic Meta-Adaptation in Smart Environments”. In: *The 6th International Conference on Adaptive and Self-Adaptive Systems and Applications*. ADAPTIVE. IARIA, May 2014, pp. 121–128.
- [150] Klaus Pohl, Günter Böckl, and Frank van der Linden. *Software Product Line Engineering. Foundations, Principles, and Techniques*. Springer, 2005.
- [151] Akshay Rajhans, Ajinkya Bhawe, Ivan Ruchkin, Bruce H. Krogh, David Garlan, André Platzer, and Bradley Schmerl. “Supporting Heterogeneity in Cyber-Physical Systems Architectures”. In: *Transactions on Automatic Control* 59.12 (Dec. 2014), pp. 3178–3193.
- [152] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working With Objects - The OOram Software Engineering Method*. Prentice Hall, 1996.
- [153] Dirk Riehle and Thomas Gross. “Role Model Based Framework Design and Integration”. In: *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA. ACM, 1998, pp. 117–133.
- [154] Grzegorz Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing, 1997.
- [155] Ramzi Ben Salah, Marius Bozga, and Oded Maler. “On Timed Components and Their Abstraction”. In: *Proceedings of the Conference on Specification and Verification of Component-based Systems*. SAVCBS. ACM, 2007, pp. 63–71.
- [156] Mazeiar Salehie and Ladan Tahvildari. “Self-Adaptive Software: Landscape and Research Challenges”. In: *Transactions on Autonomous and Adaptive Systems* 4.2 (2009), pp. 1–42.
- [157] Richard Torbjørn Sanders, Humberto Nicolás Castejón, Frank Kraemer, and Rolv Bræk. “Using UML 2.0 Collaborations for Compositional Service Specification”. In: *Model Driven Engineering Languages and Systems*. Vol. 3713. LNCS. Springer, 2005, pp. 460–475.
- [158] Pete Sawyer, Nelly Bencomo, Jon Whittle, Emmanuel Letier, and Anthony Finkelstein. “Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems”. In: *International Conference on Requirements Engineering*. IEEE, 2010, pp. 95–103.
- [159] Wilhelm Schäfer and Heike Wehrheim. “The Challenges of Building Advanced Mechatronic Systems”. In: *Future of Software Engineering*. FOSE. IEEE, 2007, pp. 72–84.
- [160] Klaus-Dieter Schewe and Bernhard Thalheim. “Development of Collaboration Frameworks for Web Information Systems”. In: *Proceedings of the 20th International Conference on Artificial Intelligence*. EMC. ACM, 2007, pp. 27–32.
- [161] Douglas C. Schmidt. “Model-Driven Engineering”. In: *Computer* 39.2 (Feb. 2006).
- [162] John A. Stankovic, Insup Lee, Aloysius Mok, and Raj Rajkumar. “Opportunities and Obligations for Physical Computing Systems”. In: *Computer* 38.11 (Nov. 2005), pp. 23–31.

- [163] Mark-Oliver Stehr, Minyoung Kim, and Carolyn Talcott. “Toward Distributed Declarative Control of Networked Cyber-Physical Systems”. In: *Ubiquitous Intelligence and Computing*. Vol. 6406. LNCS. Springer, Oct. 2010, pp. 397–413.
- [164] Daniel Sykes, Jeff Magee, and Jeff Kramer. “FlashMob: Distributed Adaptive Self-assembly”. In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS. ACM, 2011, pp. 100–109.
- [165] Janos Sztipanovits, Gabor Karsai, and Ted Baptý. “Self-Adaptive Software for Signal Processing”. In: *Communication* 41.5 (1998), pp. 66–73.
- [166] Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic. “PLASMA: A Plan-based Layered Architecture for Software Model-driven Adaptation”. In: *Proceedings of the International Conference on Automated Software Engineering*. ASE. ACM, 2010, pp. 467–476.
- [167] Guy Theraulaz and Eric Bonbeau. “A Brief History of Stigmergy”. In: *Artificial Life* 5.2 (Apr. 1999), pp. 97–116.
- [168] Mario Trapp, Rasmus Adler, Marc Förster, and Janosch Junger. “Runtime Adaptation in Safety-Critical Automotive Systems”. In: *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering*. SE. ACTA Press, 2007, pp. 308–315.
- [169] Frank Trollman, Grzegorz Lehmann, and Sahin Albayrak. “Separating Local and Global Aspects of Runtime Model Reconfiguration”. In: *Proceedings of the 5th Workshop on Models@run.time*. Vol. 641. CEUR Workshop Proceedings. CEUR-WS, 2010, pp. 72–83.
- [170] Wolfgang Trumler, Markus Helbig, Andreas Pietzowski, Benjamin Satzger, and Theo Ungerer. “Self-Configuration and Self-Healing in AUTOSAR”. In: *Proceedings of the 14th Asia Pacific Conference on Automotive Engineering*. APAC. SAE International, Aug. 2007, pp. 1–12.
- [171] Zoltán Ujhelyi, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, Benedek Izsó, István Ráth, Zoltán Szatmári, and Dániel Varró. “EMF-IncQuery: An Integrated Development Environment for Live Model Queries”. In: *Science of Computer Programming* 98.1 (Feb. 2015).
- [172] Ricardo Valerdi, Elliot Axelband, Thomas Baehren, Barry Boehm, Dave Dorenbos, Scott Jackson, Azad Madni, Gerald Nadler, Paul Robitaille, and Stan Settles. “A Research Agenda for Systems of Systems Architecting”. In: *International journal of System of Systems Engineering* 1.1 (2008), pp. 171–188.
- [173] Emil Vassev and Mike Hinchey. “The Challenge of Developing Autonomic Systems”. In: *Computer* 43.12 (2010), pp. 93–96.
- [174] Thomas Vogel and Holger Giese. “Adaptation and Abstract Runtime Models”. In: *Proceedings of the 5th Workshop on Software Engineering for Adaptive and Self-Managing Systems at the 32nd International Conference on Software Engineering*. SEAMS. ACM, May 2010, pp. 39–48.
- [175] Thomas Vogel and Holger Giese. “Model-Driven Engineering of Self-Adaptive Software with EUREMA”. In: *Transactions on Autonomous and Adaptive Systems* 8.4 (Jan. 2014), pp. 1–33.

- [176] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. “Incremental Model Synchronization for Efficient Run-time Monitoring”. In: *Proceedings of the International Conference on Models in Software Engineering*. Vol. 6002. LNCS. Springer, Apr. 2010, pp. 124–139.
- [177] Thomas Vogel, Andreas Seibel, and Holger Giese. “Toward Megamodels at Runtime”. In: *Proceedings of the 5th International Workshop on Models@run.time at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems*. Vol. 641. CEUR Workshop Proceedings. CEUR-WS, Oct. 2010, pp. 13–24.
- [178] Thomas Vogel, Andreas Seibel, and Holger Giese. “The Role of Models and Megamodels at Runtime”. In: *Proceedings of the International Conference on Models in Software Engineering*. Vol. 6627. LNCS. Springer, May 2011, pp. 224–238.
- [179] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. “On Interacting Control Loops in Self-Adaptive Systems”. In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS. ACM, 2011, pp. 202–207.
- [180] Paul Ward, Mariusz Pelc, James Hawthorne, and Richard Anthony. “Embedding Dynamic Behaviour into a Self-configuring Software System”. In: *Proceedings of the 5th International Conference on Autonomic and Trusted Computing*. ATC. Springer, 2008, pp. 373–387.
- [181] Danny Weyns, Sam Malek, and Jesper Andersson. “On Decentralized Self-adaptation: Lessons from the Trenches and Challenges for the Future”. In: *Proceedings of the Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS. ACM, 2010, pp. 84–93.
- [182] Danny Weyns, Sam Malek, and Jesper Andersson. “FORMS: Unifying Reference Model for Formal Specification of Distributed Self-adaptive Systems”. In: *Transactions on Autonomous and Adaptive Systems* 7.1 (May 2012), pp. 1–61.
- [183] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaella Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. “On Patterns for Decentralized Control in Self-Adaptive Systems”. In: *Software Engineering for Self-Adaptive Systems II*. Vol. 7475. LNCS. Springer, 2013, pp. 76–107.
- [184] Martin Wirsing and European Research Consortium for Informatics and Mathematics and National Science Foundation (U.S.) *Report on the EU/NSF Strategic Workshop on Engineering Software-Intensive Systems*. ERCIM, May 2004.
- [185] Wayne Wolf. “The Good News and the Bad News”. In: *Computer* 40.11 (2007), pp. 104–105.
- [186] Peter T. Wood. “Query Languages for Graph Databases”. In: *SIGMOD Record* 41.1 (Apr. 2012), pp. 50–60.
- [187] Peter R. Wurman, Raffaello D’Andrea, and Mick Mountz. “Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses”. In: *AI Magazine* 29.1 (2008), pp. 9–20.
- [188] Pamela Zave and Michael Jackson. “Four Dark Corners of Requirements Engineering”. In: *Transaction Software Engineering Methodology* 6.1 (Jan. 1997), pp. 1–30.

- [189] Marc Zeller and Christian Prehofer. “A Multi-layered Control Approach for Self-adaptation in Automotive Embedded Systems”. In: *Advances in Software Engineering* 2012.10 (Jan. 2012), pp. 1–15.
- [190] Dandan Zhang, Guangming Xie, Junzhi Yu, and Long Wang. “Adaptive Task Assignment for Multiple Mobile Robots via Swarm Intelligence Approach”. In: *Robotics and Autonomous Systems* 55.7 (July 2007), pp. 572–588.
- [191] Jian Zhang, Heather J. Goldsby, and Betty H.C. Cheng. “Modular Verification of Dynamically Adaptive Systems”. In: *Proceedings of the 8th International Conference on Aspect-oriented Software Development*. AOSD. ACM, 2009, pp. 161–172.

Appendix A. Deurema Metamodel

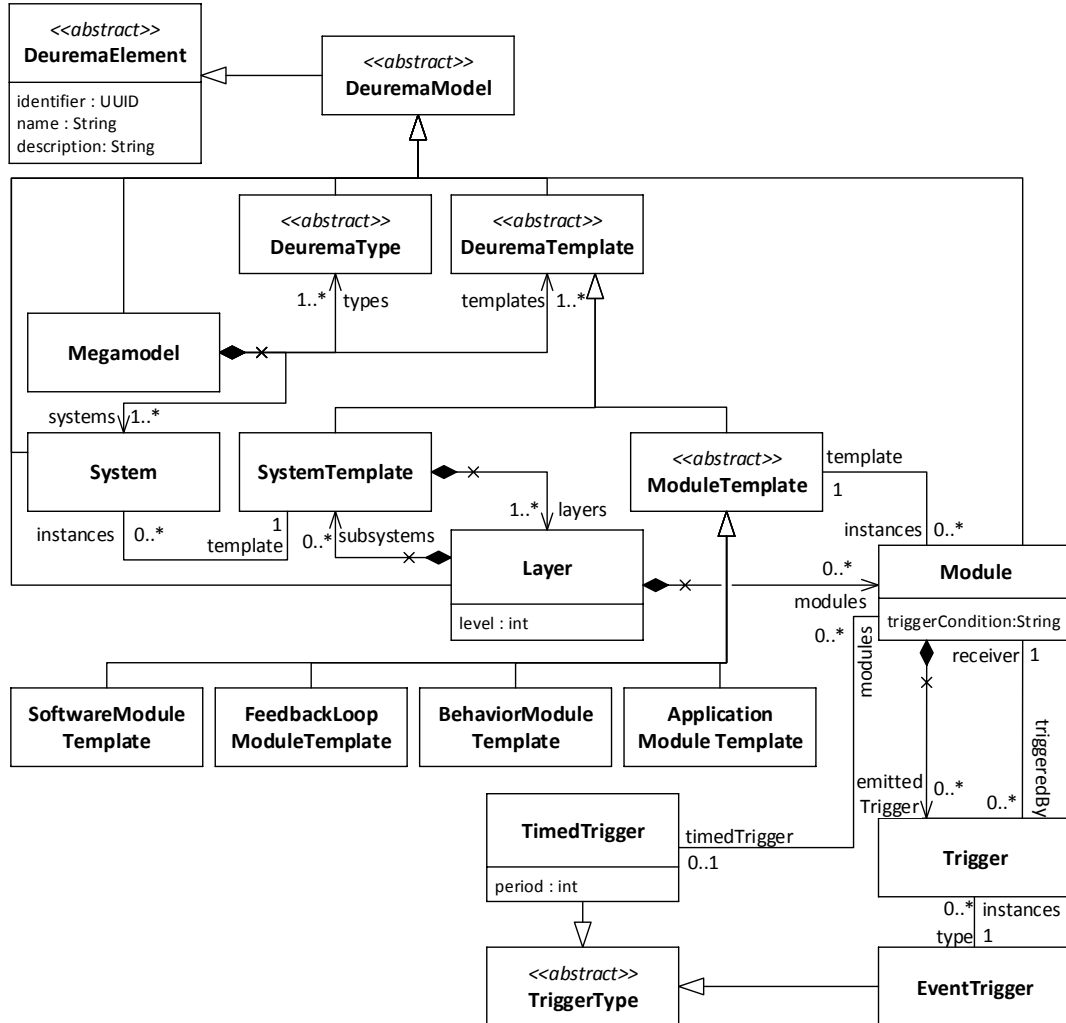


Figure A.1: This figure shows the excerpt of the Deurema metamodel according to the megamodel, system and module concept. The megamodel maintains all Deurema model elements as introduced in Section 5.1. Systems define a layered, adaptive architecture and contain modules. Modules encapsulate the local adaptation logic, which is specified in a corresponding template description. Deurema supports four module template types as discussed in Section 5.3. Furthermore, Deurema supports the triggering between modules as comprehensively discussed in Section 5.4.

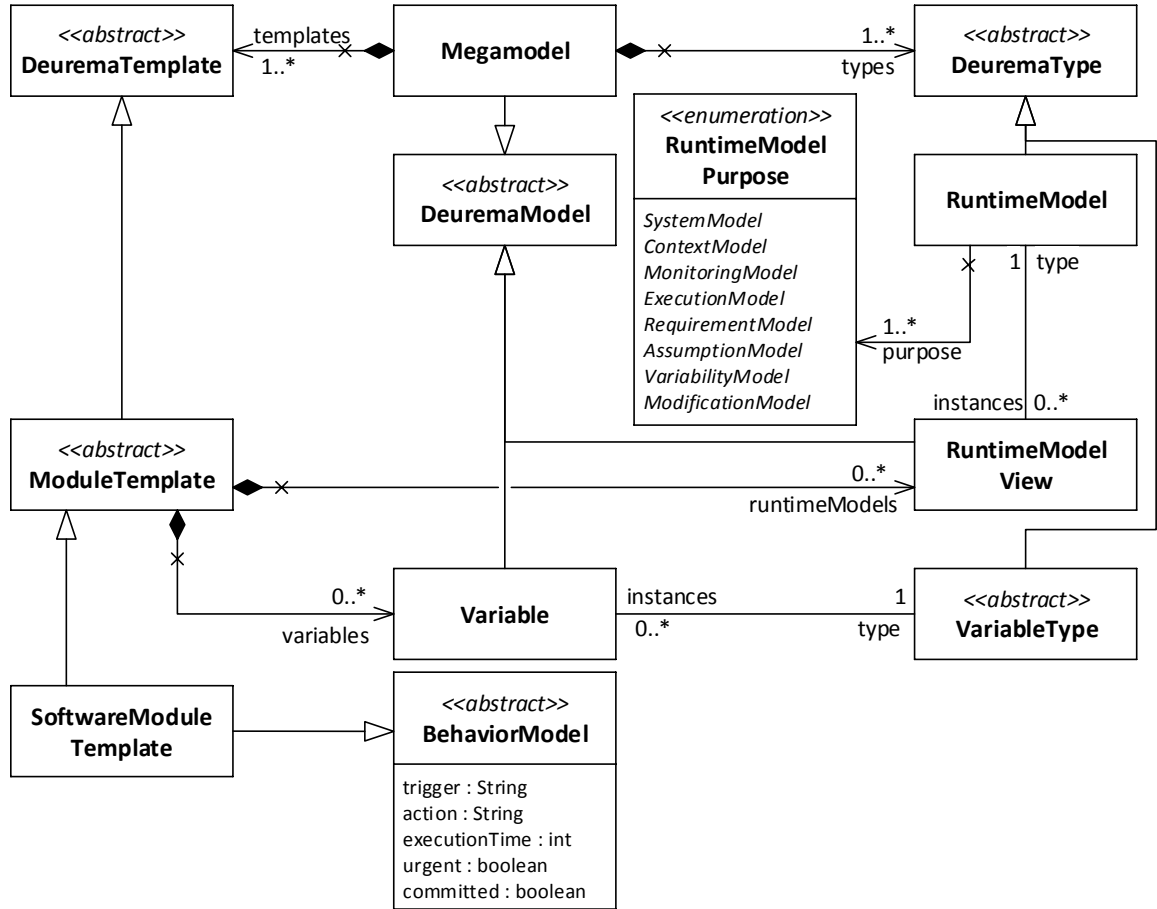


Figure A.2: The Deurema metamodel excerpt defines the responsibilities of a module template with respect to the runtime model view handling (cf. Section 5.2) and variables (cf. Section 5.6). The supported runtime model purposes follow the runtime model categorization in Section 5.2.

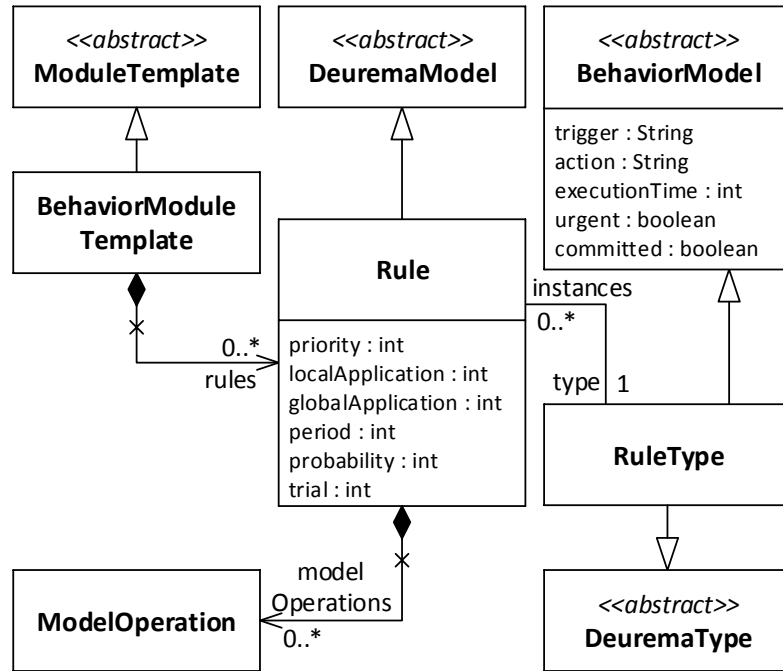


Figure A.4: The Deurema metamodel excerpt defining the behavior module template concept of using declarative graph transformation rules for specifying local adaptation effects. The adaptation rules have access to runtime models via Deurema model operations. Furthermore, rule properties such as a priority or probability further restrict the application of the local adaptation effects as comprehensively discussed in Section 5.3.5.

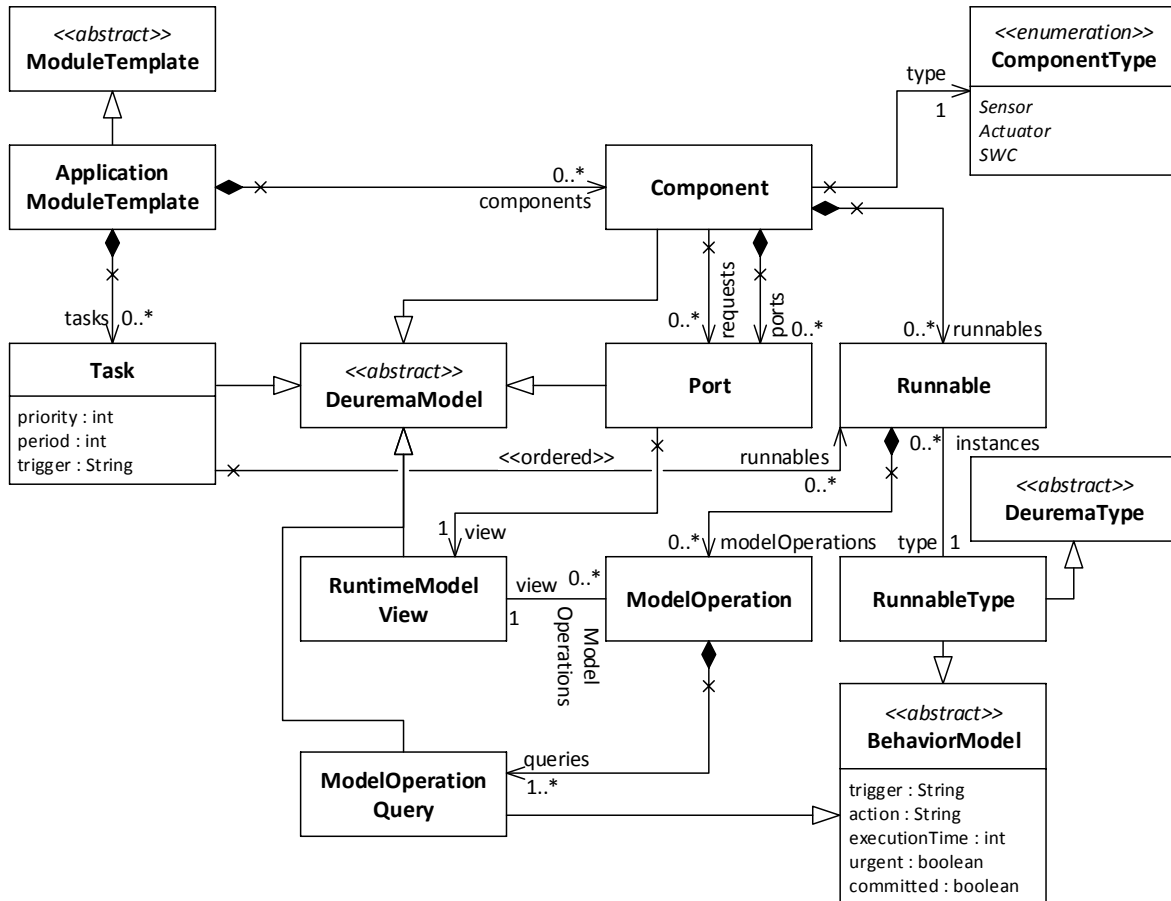


Figure A.5: The Deurema metamodel defining the application module template concepts. Deurema supports the component-based specification of adaptive behavior, where components and ports define the static software architecture. Runnables and tasks realize the dynamic (behavioral) part of the component architecture, where runnables can manipulate the runtime model data. Application module templates are comprehensively discussed in Section 5.3.4.

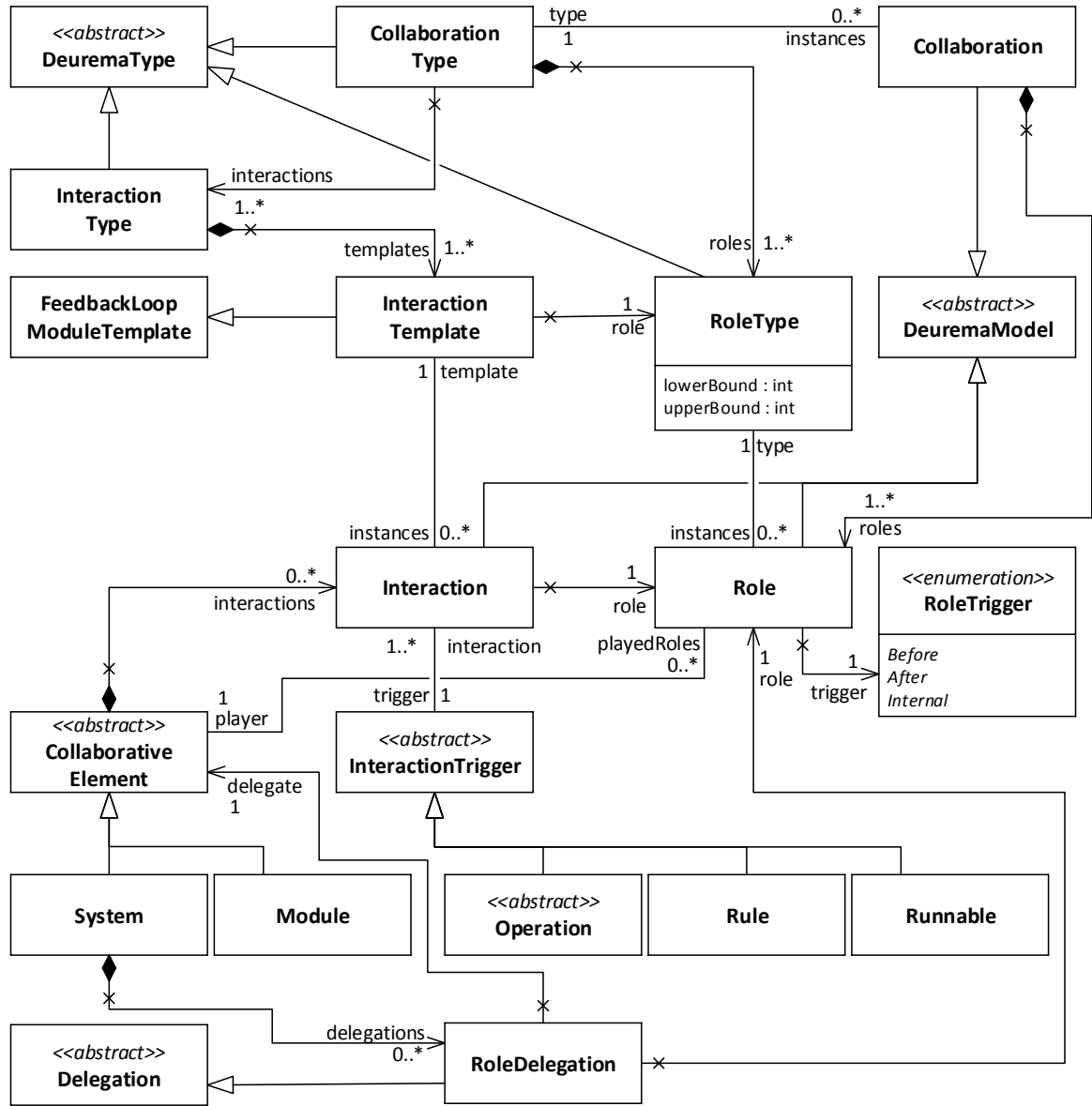


Figure A.6: This figure shows the Deurema metamodel class definition with respect to the collaboration concept. Structural aspects are defined in the collaboration structure definition, which comprises the collaboration type and the role types. Behavioral aspects are defined in the collaboration choreography specification by means of interactions that are specialized feedback loops. The collaboration mapping defines the systems and modules that play the corresponding roles defined by the collaboration structure. Systems must delegate the role to an inner system or module, which realized the interaction behavior. According to the module template type, an operation, rule, or runnable can trigger the mapped collaboration starting the interaction with another module or system. The Deurema collaboration concept is comprehensively discussed in Section 5.5.

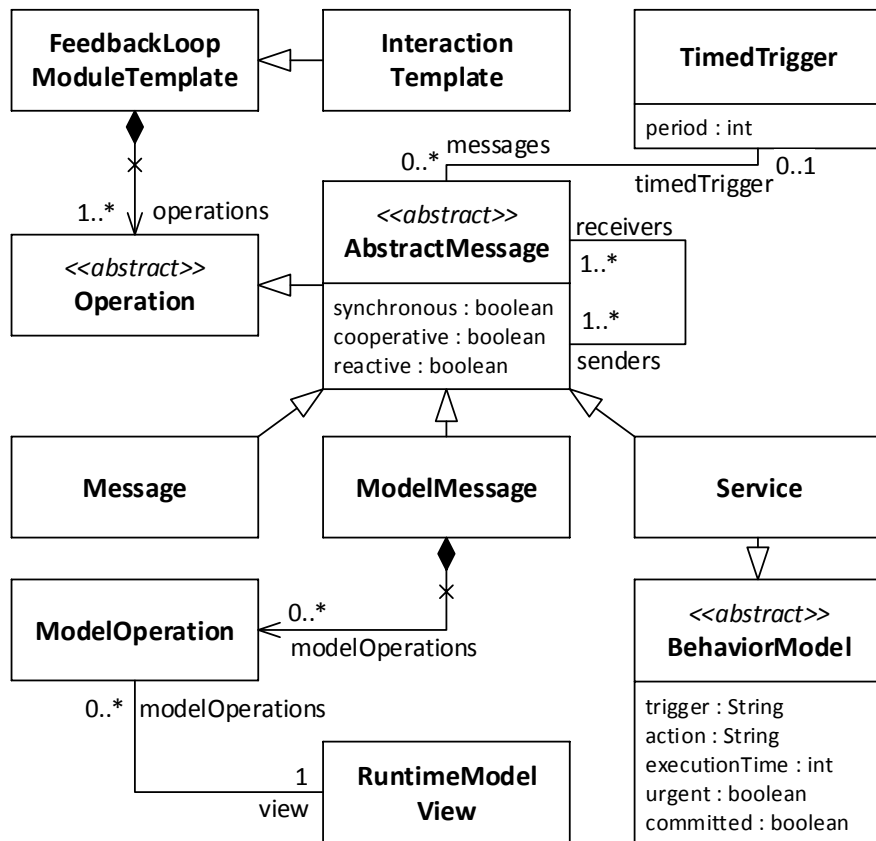


Figure A.7: The metamodel excerpt defines the Deurema message concept within interactions. Deurema supports synchronization messages, model messages and services, where each type is comprehensively discussed in Section 5.5.3.

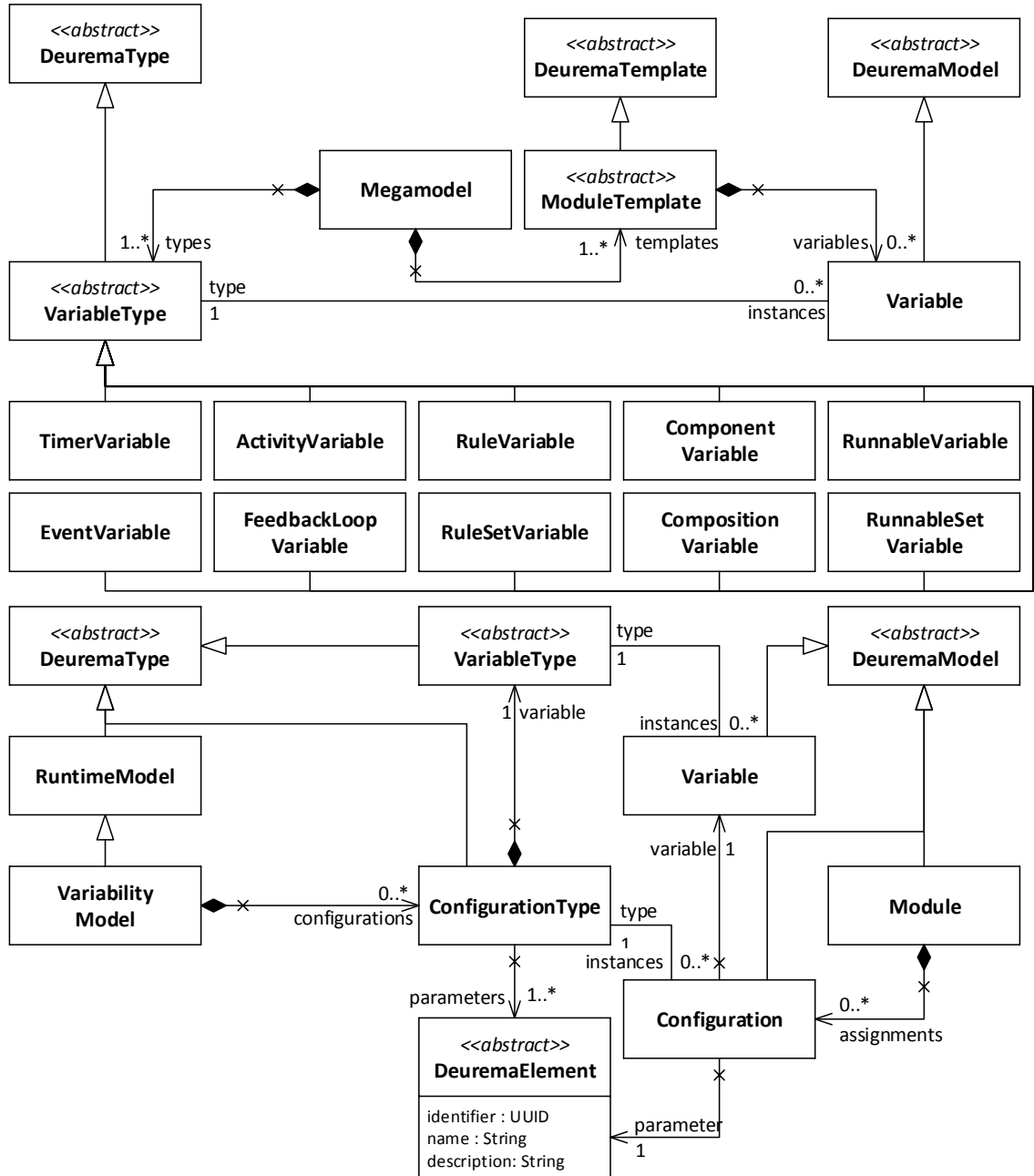


Figure A.8: This metamodel excerpt shows the supported variable types of the Deurema modeling language. The variable types are introduced along with the module template description. Variables and the corresponding variability Deurema runtime model enable the Deurema reflection mechanism supporting system reconfiguration. The Deurema reflection mechanism, reconfiguration and adaptation capabilities are comprehensively discussed in Section 5.6.

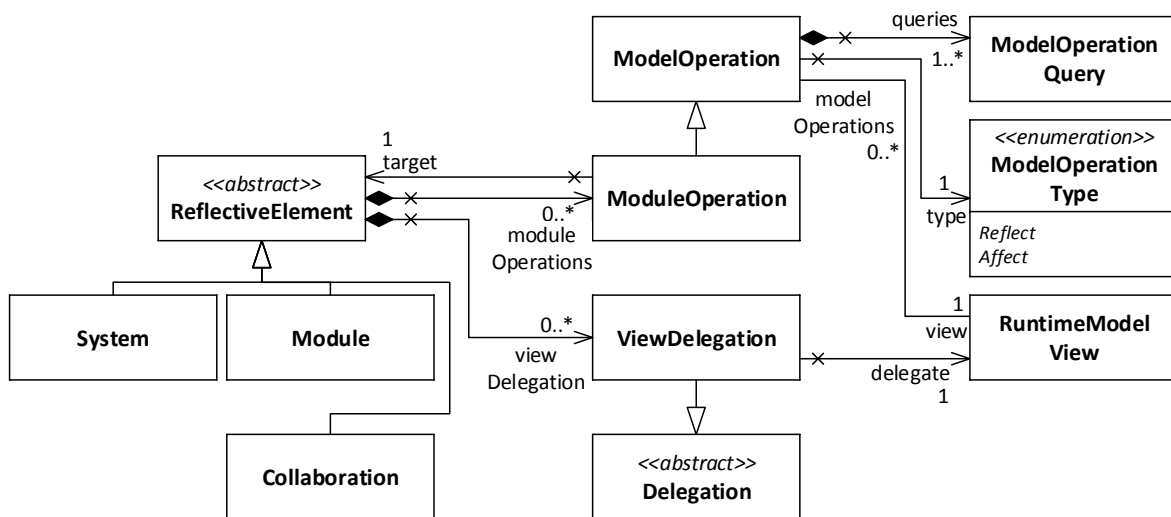


Figure A.9: This figure shows the metamodel excerpt of the Deurema view delegation concept. Systems, modules, and collaborations (short reflective elements) can reflect respectively affect underlying reflective elements. The reflected runtime information is delegated to a runtime model view in the corresponding module respectively interaction template. The view delegation concept is introduced in Section 5.6.

Appendix B. Interaction Message

As motivated in Section 5.5.3, a message in an interaction has a synchronous, cooperative, as well as reactive property, which influence the execution semantic of this message. Figure B.1 enumerates all four possible combinations concerning the cooperative and reactive property. The semantic of these properties is already discussed in Section 5.5.3. In the following, concrete scenarios of sending and receiving messages over time are discussed.

<<abstract>> AbstractMessage		
	Reactive (Sender)	Cooperative (Receiver)
synchronous : boolean cooperative : boolean reactive : boolean	(1) first (true)	or (false)
	(2) first (true)	and (true)
	(3) all (false)	or (false)
	(4) all (false)	and (true)

Figure B.1: Combinations of the reactive and cooperative message properties

Figure B.2 shows the example collaboration definition on the left and the scenarios enumerated from I to V on the right. The collaboration structure defines two multi-roles, named Sender and Receiver. Furthermore, the choreography specification defines one interaction, where the sender role sends a single message M to the receiver role. On the bottom, the collaboration deployment is shown, where two software modules realize the sender role and two software modules realize the receiver role. Thus, there are two senders and two receivers in the collaboration example.

On the right, Figure B.2 enumerates five scenarios of sending and receiving the message M at different point in times (t_1 to t_4). The rectangle labeled with the name s_1M denotes the sending of the message M by the sender playing the role s_1 as shown for the software module sm_1 in the collaboration deployment specification on the left. The sender always emits the message immediately after it is executed. Furthermore, the sender waits (synchronous case) two time steps for the reception of the message by one or both receivers. After waiting two time steps, the sender cancels its waiting and aborts execution. Straight forward, the rectangle labeled with the name r_1M denotes the receiving of the message M by the receiver playing the role r_1 . The receiver is able to process the message directly after it is executed and waits for two time steps for an incoming message, if no message was already sent. For example, the scenario I describes the parallel execution of three software modules sm_1 , sm_3 , and sm_4 from time t_0 to t_2 . Afterwards, the parallel sending respectively receiving of the message is performed from time t_2 to t_4 . In all scenarios, the interaction is performed after the black box behavior of the corresponding player software module is executed. Table B.1 shows the causal execution order for all five scenarios in Figure B.2 for all four combinations of the message properties in Figure B.1.

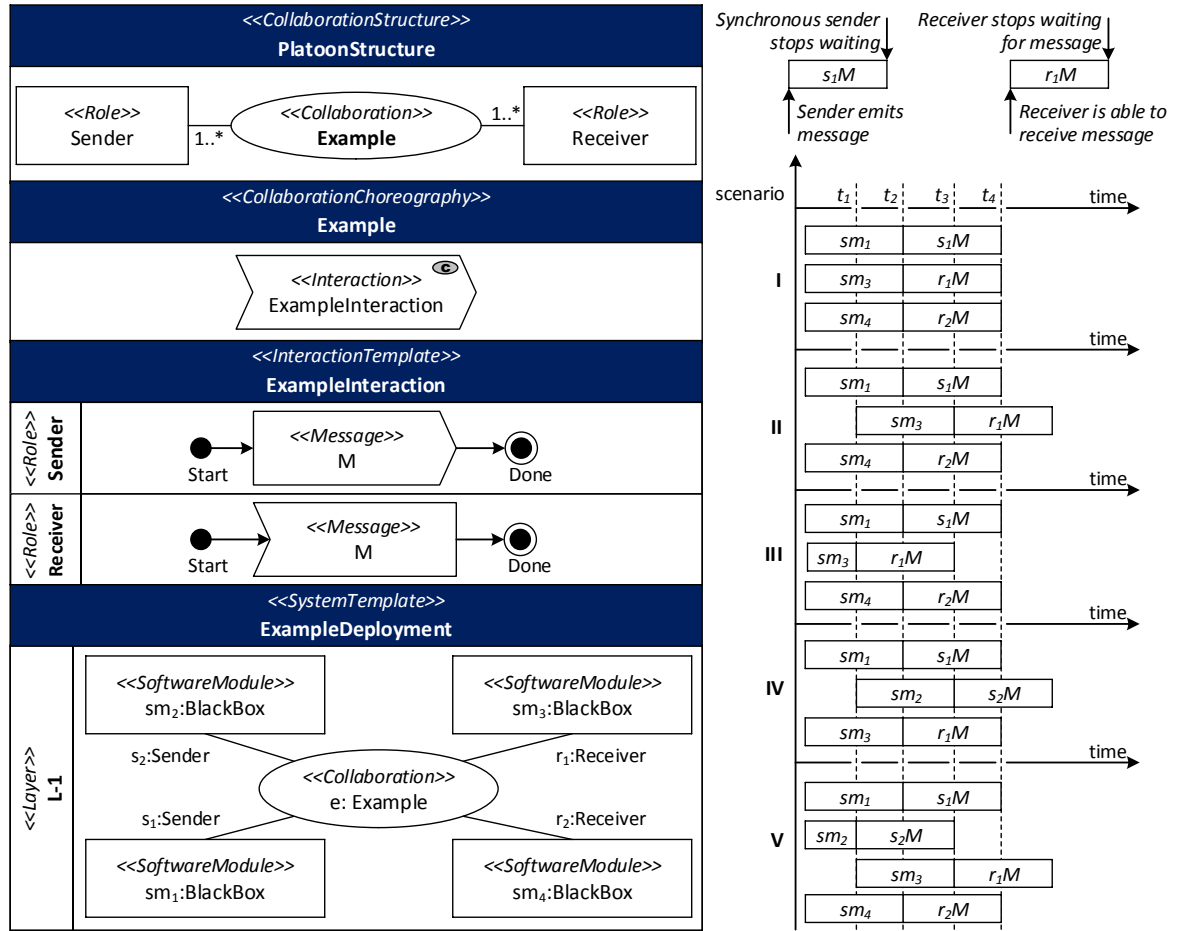


Figure B.2: Interaction scenarios with multiple senders and receivers

Table B.1: Execution order for the scenarios (Sc) from Figure B.2 together with the message property combinations (MPC) in Figure B.1

Sc	MPC	Causal Execution Order
I	(1)	$\xrightarrow{0} sm_1 \parallel sm_3 \parallel sm_4 \xrightarrow{2} s_1M \rightarrow r_1M \parallel r_2M$
I	(2)	as I (1)
I	(3)	as I (1)
I	(4)	as I (1)
II	(1)	$\xrightarrow{0} sm_1 \parallel sm_4 \xrightarrow{1} sm_1 \parallel sm_3 \parallel sm_4 \xrightarrow{2} sm_3 \parallel s_1M \rightarrow r_2M$
II	(2)	$\xrightarrow{0} sm_1 \parallel sm_4 \xrightarrow{1} sm_1 \parallel sm_3 \parallel sm_4 \xrightarrow{2} sm_3 \parallel s_1M \xrightarrow{3} r_1M \parallel r_2M$
II	(3)	as II (1)
II	(4)	as II (2)
III	(1)	$\xrightarrow{0} sm_1 \parallel sm_3 \parallel sm_4 \xrightarrow{1} sm_1 \parallel sm_4 \xrightarrow{2} s_1M \rightarrow r_1M \parallel r_2M$
III	(2)	as III (1)
III	(3)	as III (1)
III	(4)	as III (1)
IV	(1)	$\xrightarrow{0} sm_1 \parallel sm_3 \xrightarrow{1} sm_1 \parallel sm_2 \parallel sm_3 \xrightarrow{2} sm_2 \parallel s_1M \rightarrow r_1M$
IV	(2)	as IV (1)
IV	(3)	$\xrightarrow{0} sm_1 \parallel sm_3 \xrightarrow{1} sm_1 \parallel sm_2 \parallel sm_3 \xrightarrow{2} sm_2 \parallel s_1M \xrightarrow{3} s_2M \rightarrow r_1M$
IV	(4)	as IV (3)
V	(1)	$\xrightarrow{0} sm_1 \parallel sm_2 \parallel sm_4 \xrightarrow{1} sm_1 \parallel sm_2 \parallel s_2M \xrightarrow{2} s_1M \parallel sm_3 \parallel r_2M$
V	(2)	$\xrightarrow{0} sm_1 \parallel sm_2 \parallel sm_4 \xrightarrow{1} sm_1 \parallel sm_2 \parallel s_2M \xrightarrow{2} s_1M \parallel sm_3 \xrightarrow{3} r_1M \parallel r_2M$
V	(3)	as V (1)
V	(4)	as V (3)

$sm_1 \parallel sm_2$: The software module sm_1 runs in parallel with the software module sm_2 ;
 \rightarrow : Causal execution dependency; $\xrightarrow{3}$: A simulation time step to global time 3 (cf. Section 7.3);
 s_1M : The sender s_1 of message M ; r_1M : The receiver r_1 of message M

Note, for the scenario V together with the message property combination (1) in Table B.1, the execution after r_2M is non-deterministic depending on the execution of sender s_1M . Because sender s_1M and receiver r_2M run in parallel, the receiver can process both messages (from s_1M and s_2M), if the sender s_1M is executed before. Otherwise, if the receiver r_2M is executed before sender s_1M , the receiver r_1M will be enabled. Both execution traces are possible and depend on the scheduling of the simulation run.

Appendix C. Analysis Rules

The Deurema analysis framework uses the deductive inference engine from Beyhl et al. [35]. Both, the analysis framework and the inference engine are implemented in Java using the Eclipse Modeling Framework. In the following, the basic concepts of the inference engine are subsumed. Afterwards, all specified annotation types are enumerated in Section C.1. Finally, all modeled Deurema analysis rules are depicted in the concrete syntax of the inference engine in Section C.2.

Figure C.1 shows an exemplary analysis rule in concrete syntax of the inference engine tool, which is used to explain the core concept of the inference engine. As discussed in Chapter 6, the analysis rules are modeled within the inference engine tool on basis of the Deurema metamodel. An analysis rule consists of a graph pattern, which is the information of interest that is searched in the Deurema model. Graph pattern objects are modeled as rectangles. Furthermore, each object has a name and a type, whereas the type refers to a class type in the Deurema metamodel. Pattern objects can be linked over references as defined by the metamodel. If the modeled pattern is found, the inference engine creates an annotation, which is directly stored in the Deurema model and maintained by the Deurema megamodel during simulation (e.g., for a runtime analysis). The rule pattern can be further restricted by an OCL constraint as shown at the top in the example rule in Figure C.1. A created annotation is denoted as green rounded, dashed rectangle. Furthermore, the ++ sign denotes the creation of an annotation. Beside new annotations, analysis rules can depend on results of other analysis rules. In this case, the rule pattern is enriched with already retrieved annotations from the former rules. Needed annotations are modeled as rounded, dashed rectangles as shown at the bottom in the graph pattern in Figure C.1. Annotations link to pattern objects via role links. Each role link has a name and is defined by a role type, which is specified in the inference engine. Annotations are also typed, which refers to the desired analysis fact. Finally, annotations can include arbitrary additional attributes, which contain further information belonging to the retrieved analysis pattern.

In summary, the analysis rule in Figure C.1 describes a pattern, which consists of three Operation objects and two CausalDependency annotations. Furthermore, the pattern is restricted by an additional OCL constraint. If the pattern is found in the Deurema model and if the two annotations are found before, a ClosureDependency annotation is created. Additionally, a level attribute is set for the closure dependency annotation. Each analysis rule specified in the inference engine has incoming ports denoting the needed, beforehand retrieved annotations (e.g., CausalDependency annotations in the example in Figure C.1) and outgoing ports denoting the retrieved analysis fact in form of the created annotation (e.g., the ClosureDependency annotation in the example). A comprehensive discussion about the inference engine tool can be found in [35].

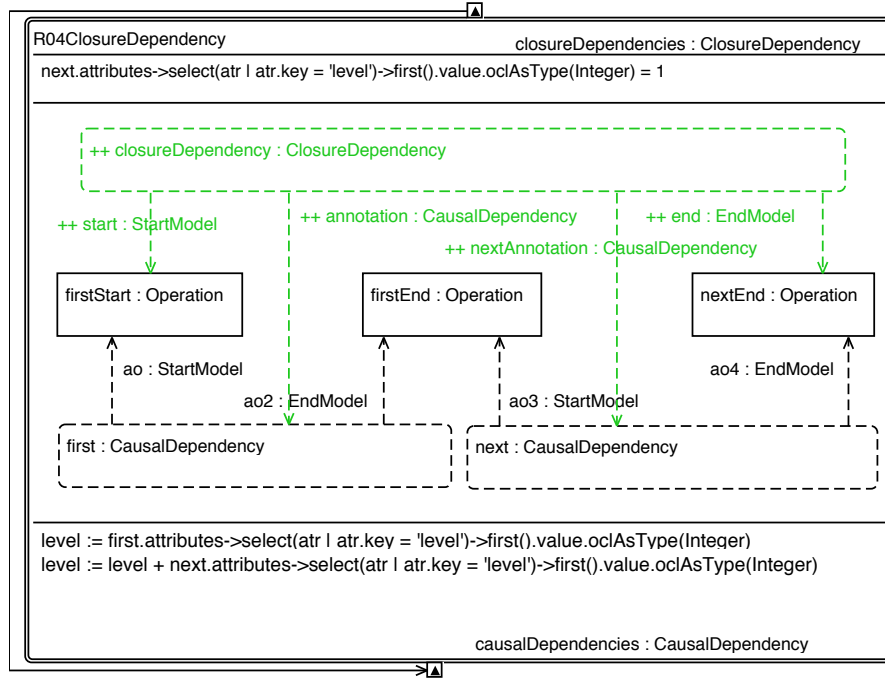


Figure C.1: Inference engine rule example searching for closure dependencies in a feedback loop module template on basis of two causal dependencies.

C.1. Annotation Types

All realized metrics and patterns for analyzing the modeled adaptive SoS architecture with the Deurema approach are comprehensively discussed in Chapter 6. This section enumerates all necessary annotation types as they are created in the inference engine for the introduced analysis metrics accordingly.

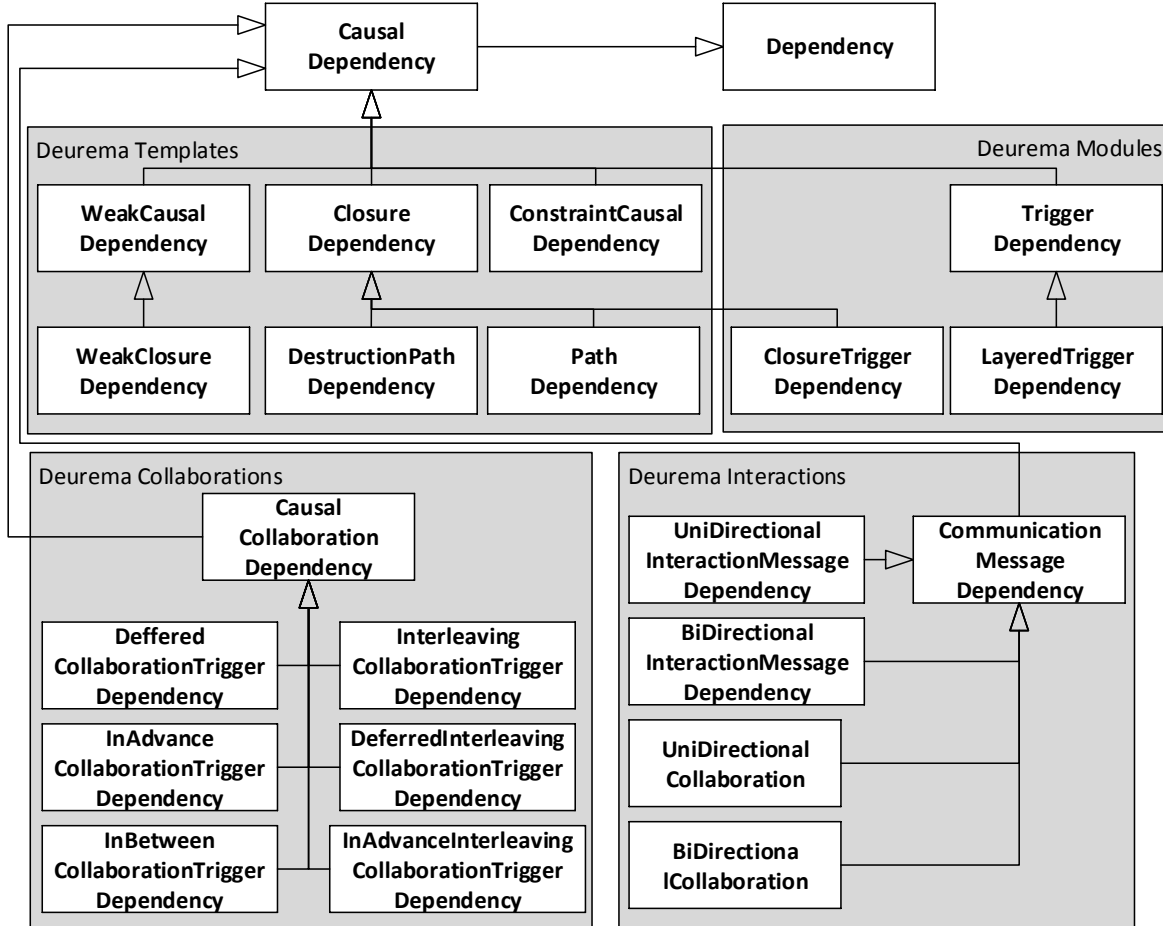


Figure C.2: Overview of annotation types for causal dependencies

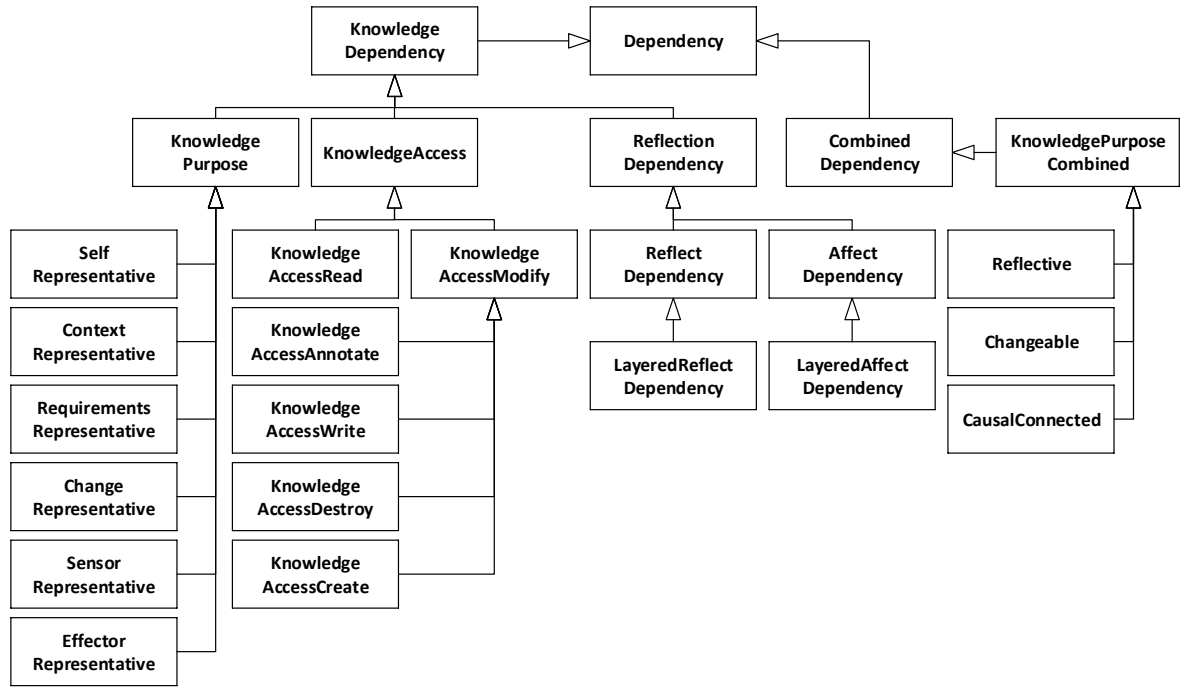


Figure C.3: Overview of annotation types for knowledge dependencies

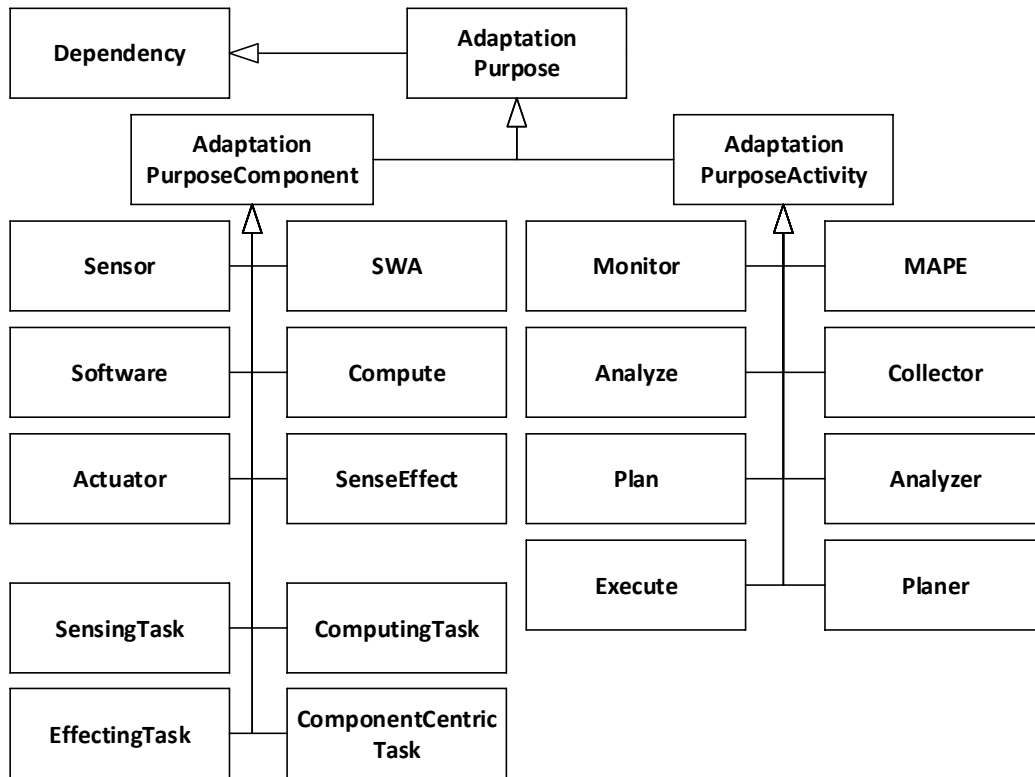


Figure C.4: Overview of annotation types for the adaptation purpose

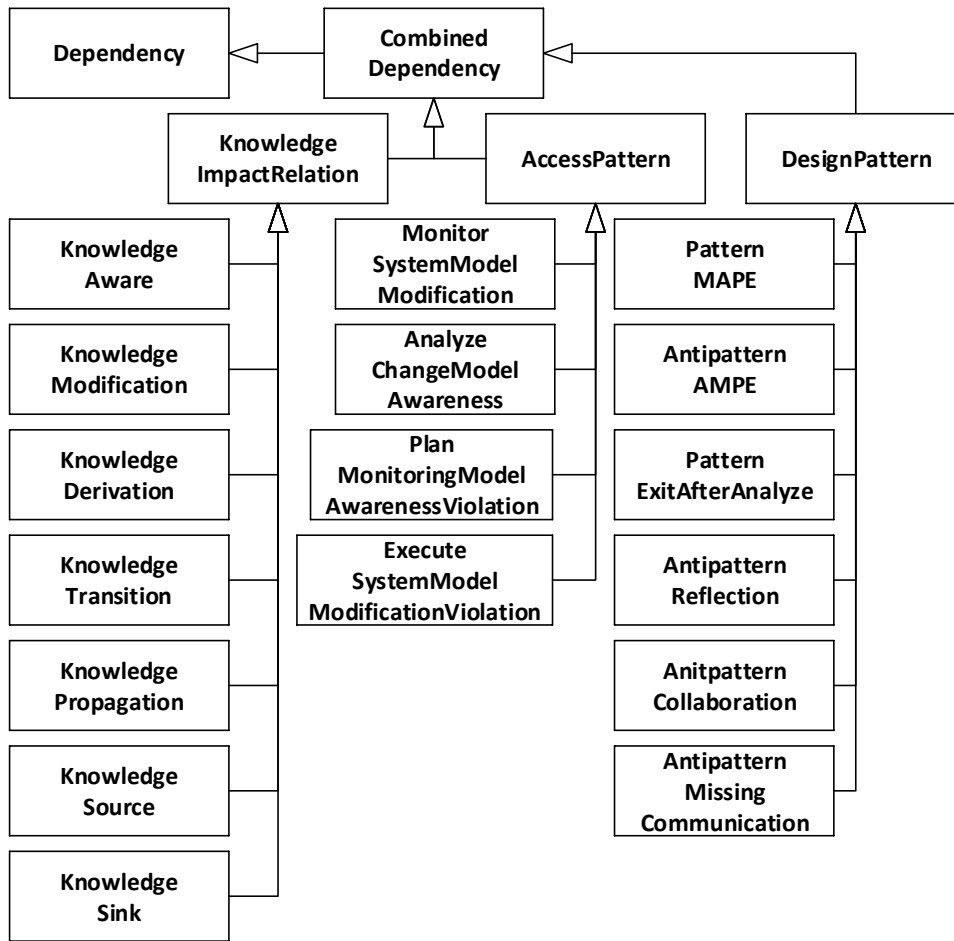


Figure C.5: Overview of annotation types for combined dependencies and patterns

C.2. Analysis Rules

In total, the Deurema analysis framework realizes 83 analysis rules to retrieve important metrics and patterns as discussed in Chapter 6. This thesis does not claim that these analysis rules are complete nor that all important metrics are covered. The realized rules are a proof of concept and can be individually extended towards important key aspects of the concrete domain or underlying problem. All depicted rules in this section are realized within the inference engine. In this context, thanks to the student Paul Geppert, who technically modeled the rules in the inference engine tool.

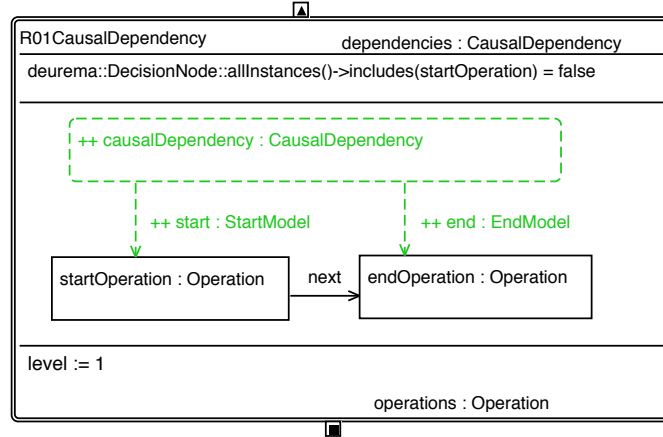


Figure C.6: This rule detects a causal dependency between operations in a feedback loop module template by determining the control flow.

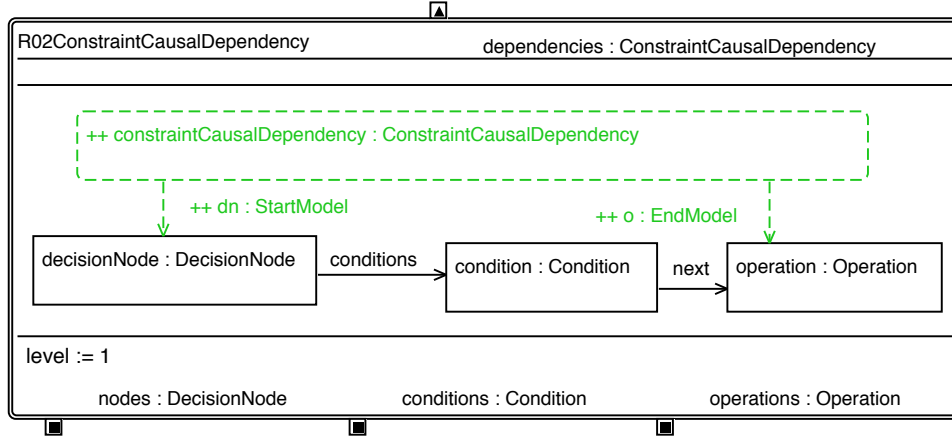


Figure C.7: This rule detects a constraint causal dependency between a decision node and an arbitrary operation in a feedback loop module template by determining the control flow.

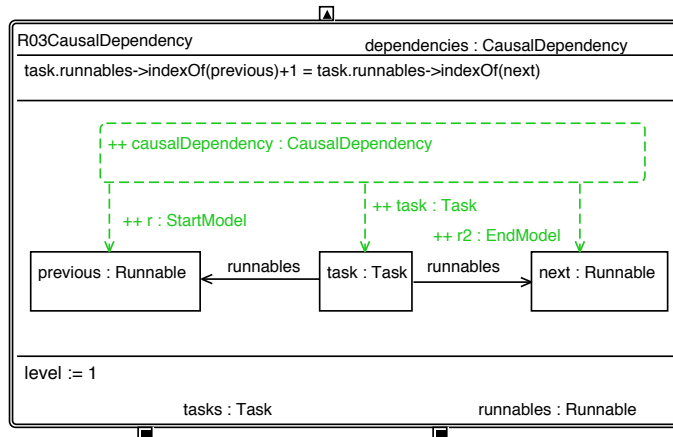
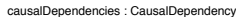


Figure C.8: This rule detects a causal dependency between two runnables in a behavior rule module template by determining the runnable task mapping.



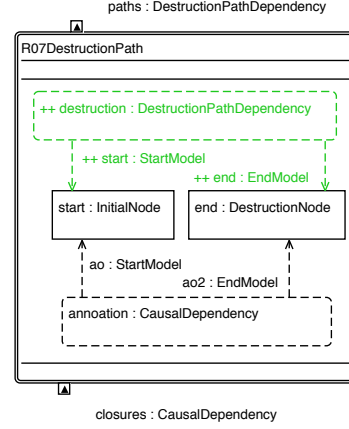


Figure C.12: The inference rule retrieves all possible destruction paths in a feedback loop module templates by looking at the initial and destruction nodes.

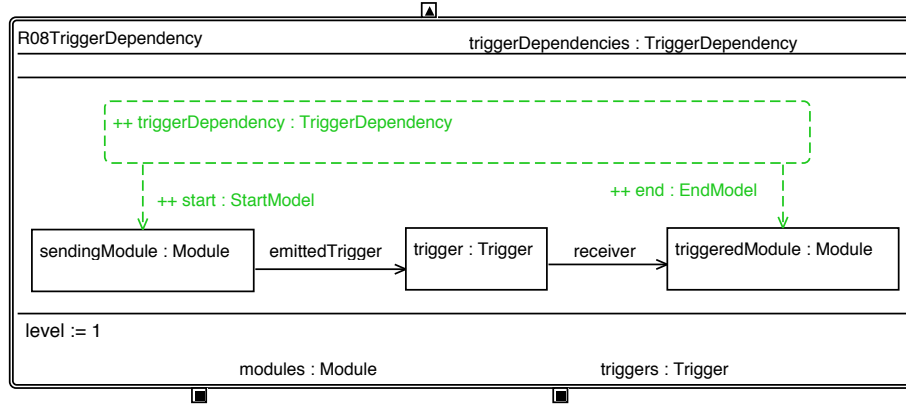


Figure C.13: The rule infers direct trigger dependencies between modules, which is a causal dependency on level of the corresponding system template specification.

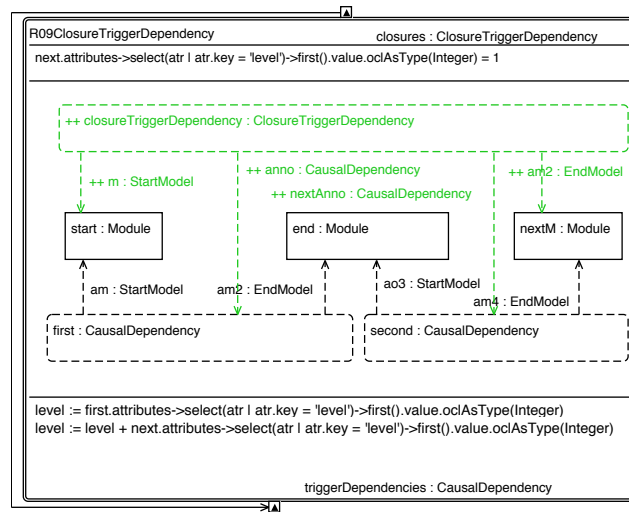


Figure C.14: The rule recursively infers all trigger dependency closures between module instances. The indirection level is annotated in the corresponding closure annotation.

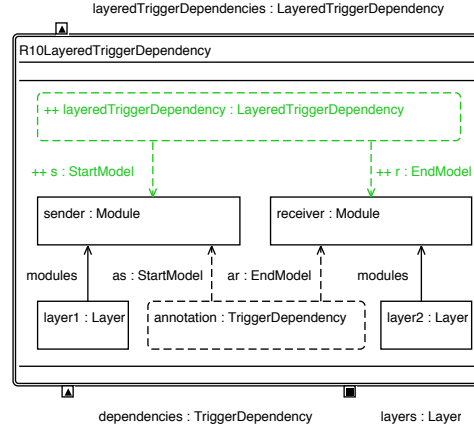


Figure C.15: This inference rule refines a beforehand retrieved trigger dependency, if modules are located on different layers.

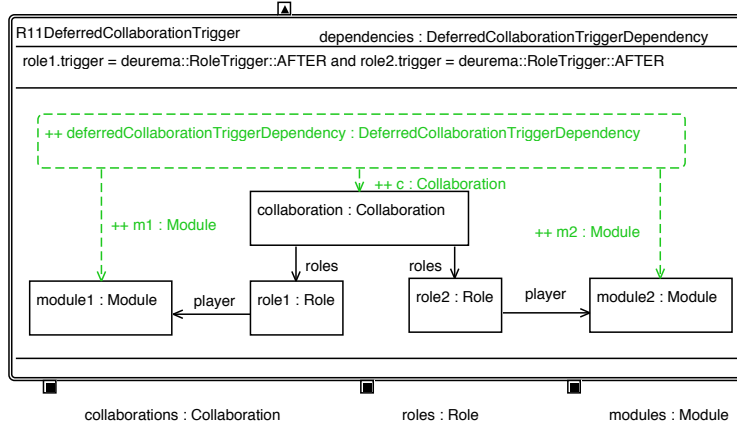


Figure C.16: The rule determines the causal order of interactions played by a pair of modules. The corresponding role trigger defines when the interactions in the corresponding collaboration instance must be executed.

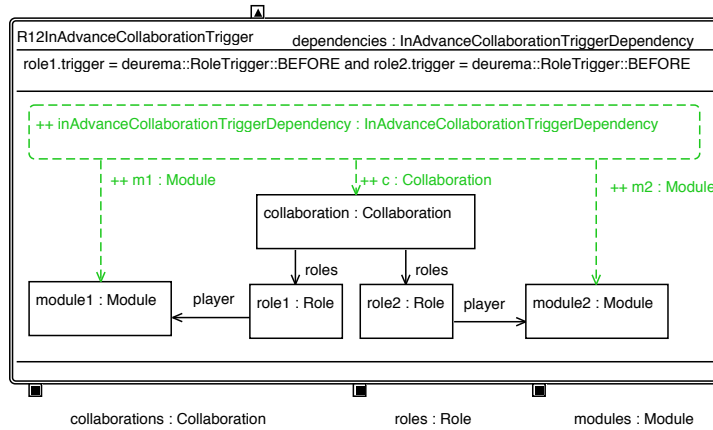


Figure C.17: The rule determines the causal order of interactions played by a pair of modules. The corresponding role trigger defines when the interactions in the corresponding collaboration instance must be executed.

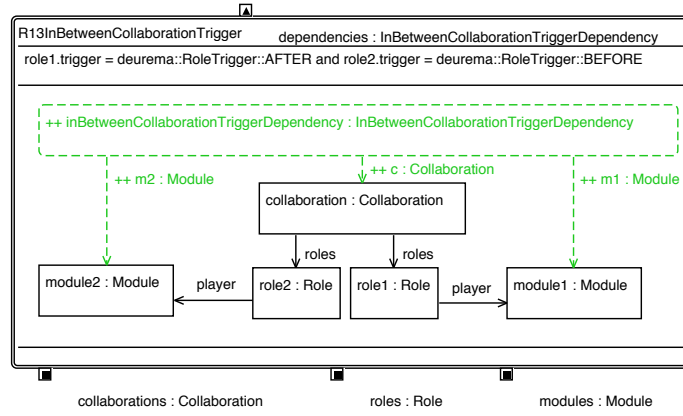


Figure C.18: The rule determines the causal order of interactions played by a pair of modules. The corresponding role trigger defines when the interactions in the corresponding collaboration instance must be executed.

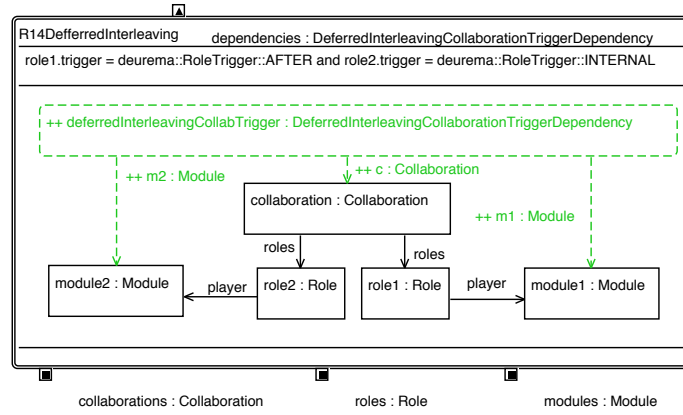


Figure C.19: The rule determines the causal order of interactions played by a pair of modules. The corresponding role trigger defines when the interactions in the corresponding collaboration instance must be executed.

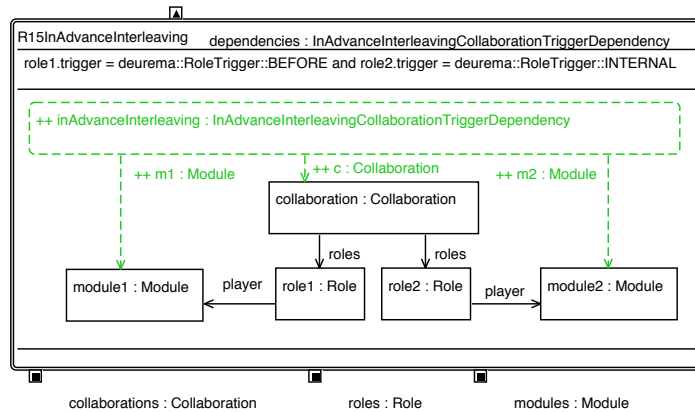


Figure C.20: The rule determines the causal order of interactions played by a pair of modules. The corresponding role trigger defines when the interactions in the corresponding collaboration instance must be executed.

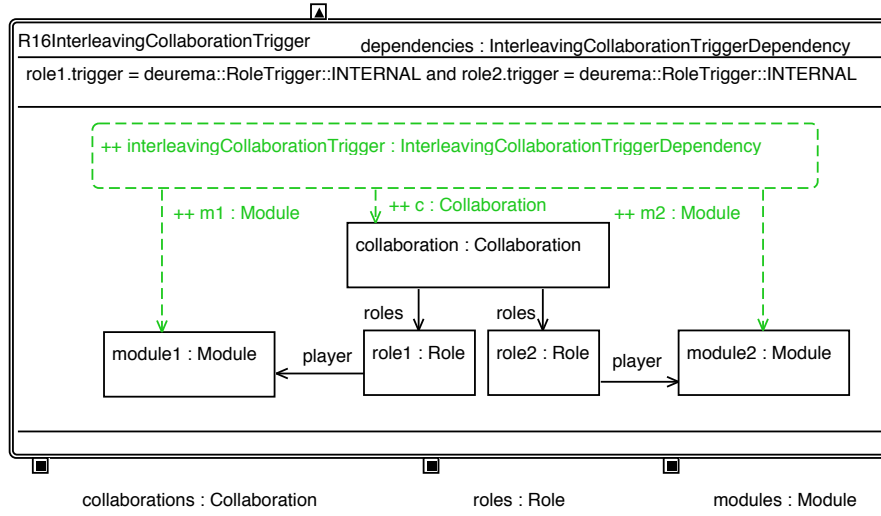


Figure C.21: The rule determines the causal order of interactions played by a pair of modules. The corresponding role trigger defines when the interactions in the corresponding collaboration instance must be executed.

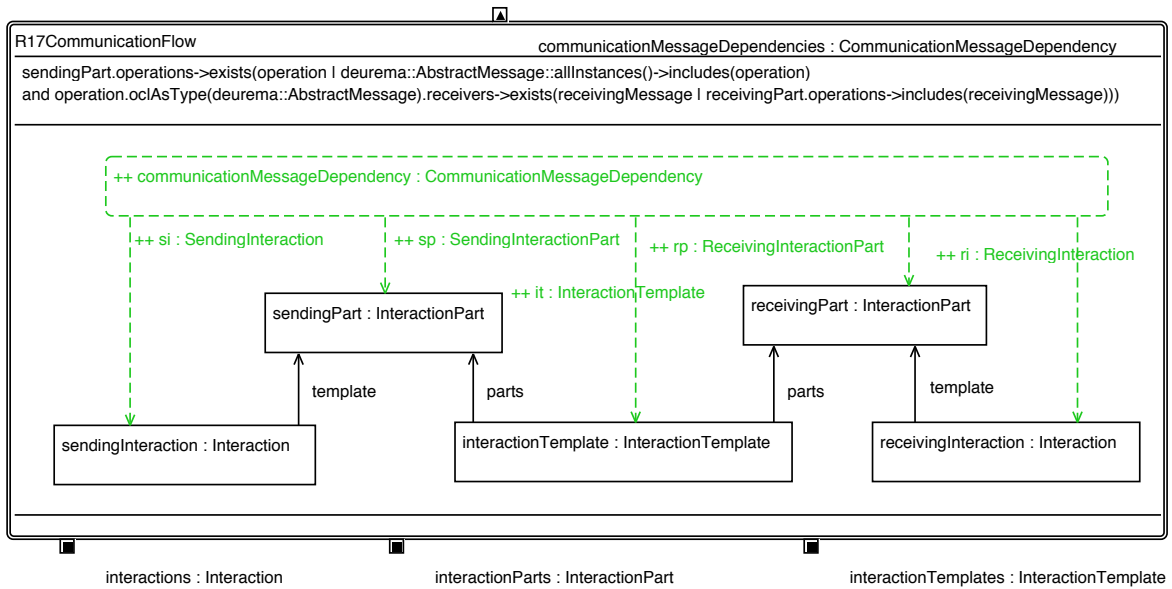


Figure C.22: This inference rule retrieves occurring communication flows between two roles. Thereby a communication can be a send/receive of a synchronization message, a model message, or a service invocation.

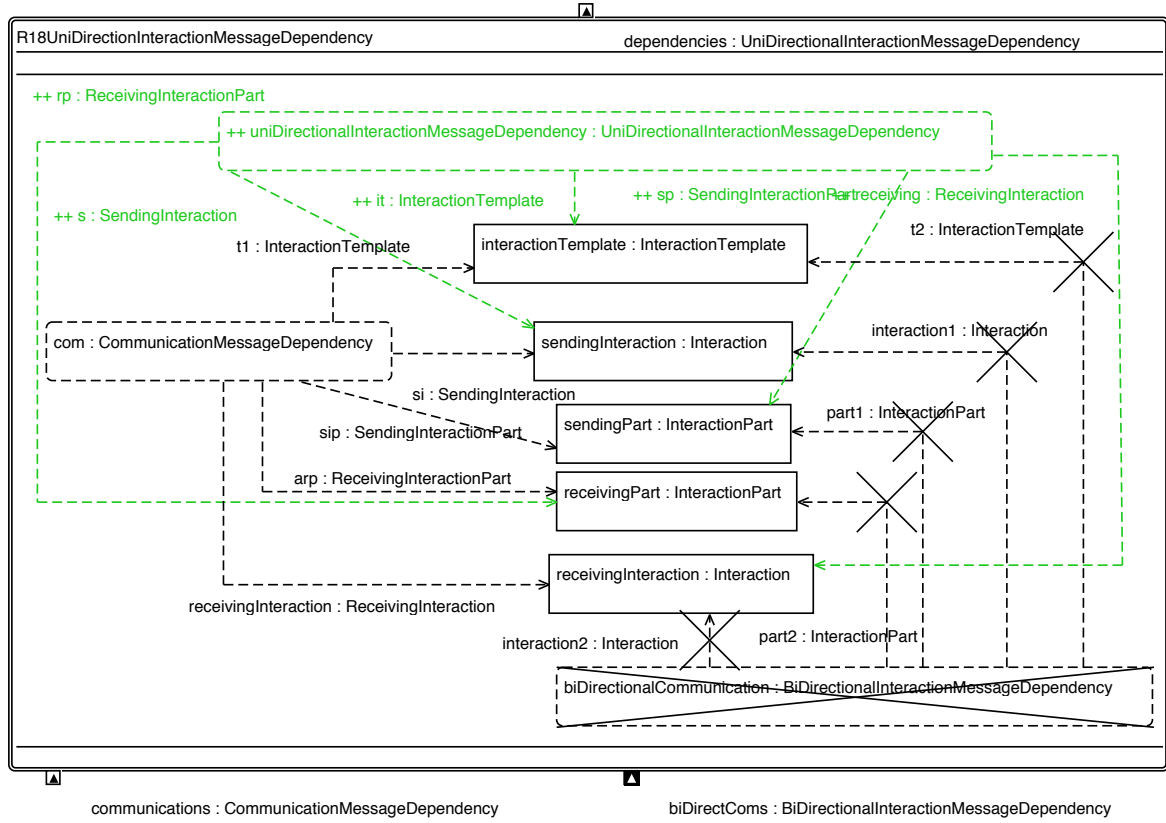


Figure C.23: Based on the communication flow, this rule detects unidirectional interactions.

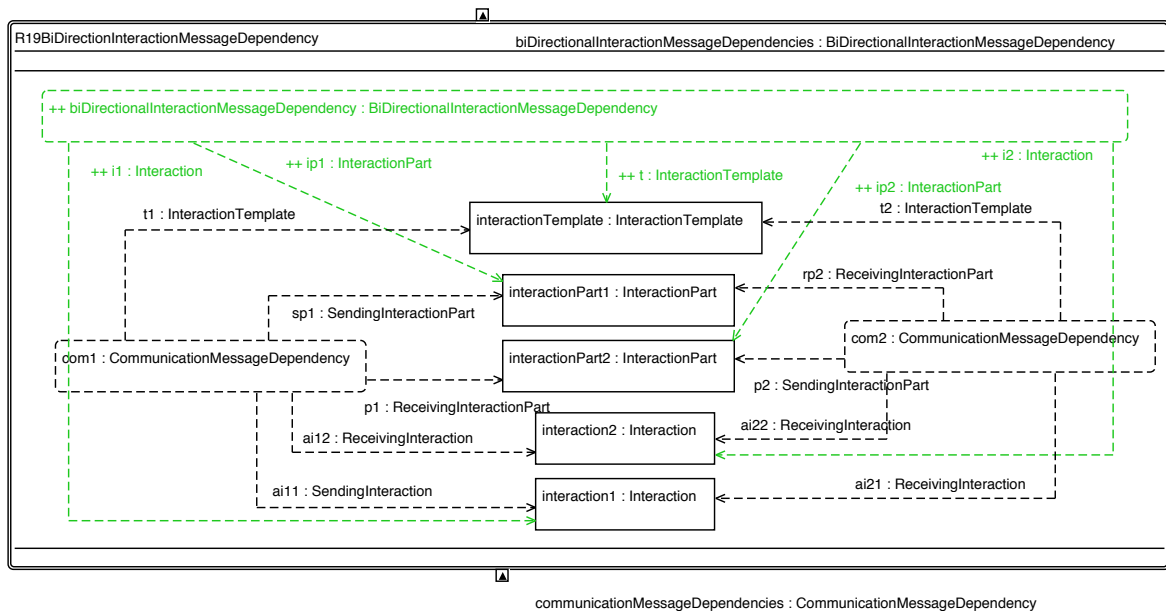


Figure C.24: Based on the communication flow, this rule detects bidirectional interactions.

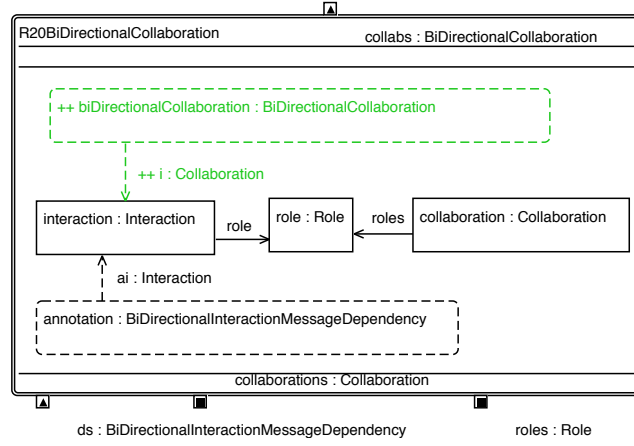


Figure C.25: A collaboration that contains a bidirectional interaction is seen as bidirectional, too.

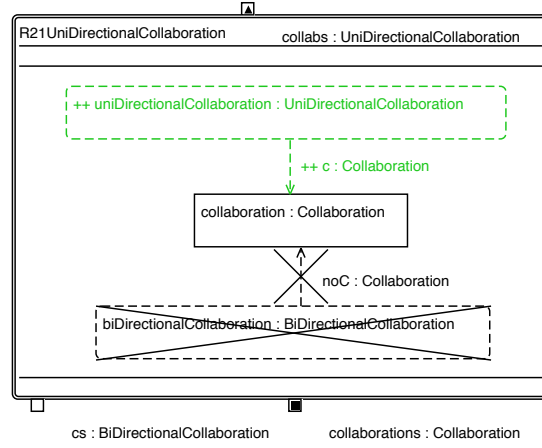


Figure C.26: A collaboration that contains only unidirectional interaction is considered as unidirectional, too.

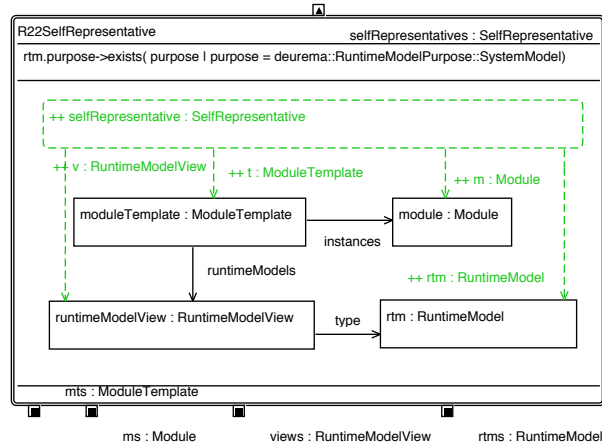


Figure C.27: This rule infers the self-representative property by looking on the available local knowledge and the defined purposes. The runtime model view must specify a *SystemModel* purpose.

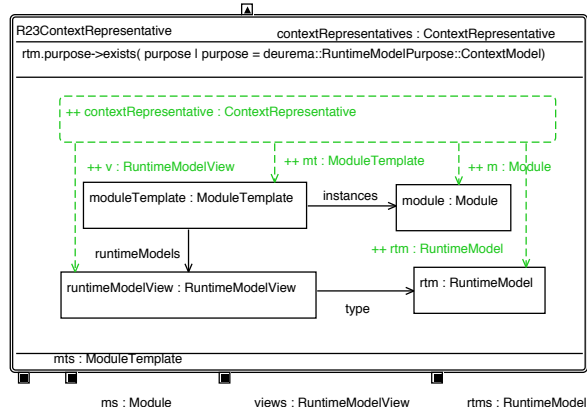


Figure C.28: This rule infers the context-representative property by looking on the available local knowledge and the defined purposes. The runtime model view must specify a ContextModel purpose.

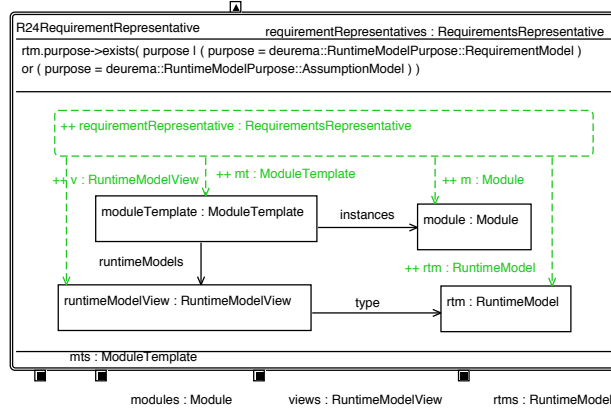


Figure C.29: This rule infers the requirement-representative property by looking on the available local knowledge and the defined purposes. The runtime model view must specify a RequirementModel or AssumptionModel purpose.

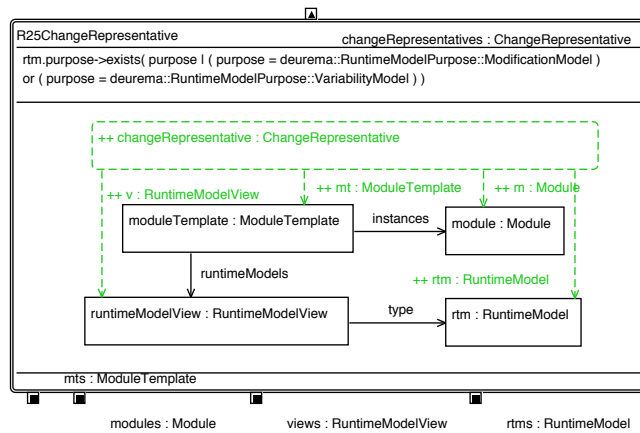


Figure C.30: This rule infers the change-representative property by looking on the available local knowledge and the defined purposes. The runtime model view must specify a ModificationModel or VariabilityModel purpose.

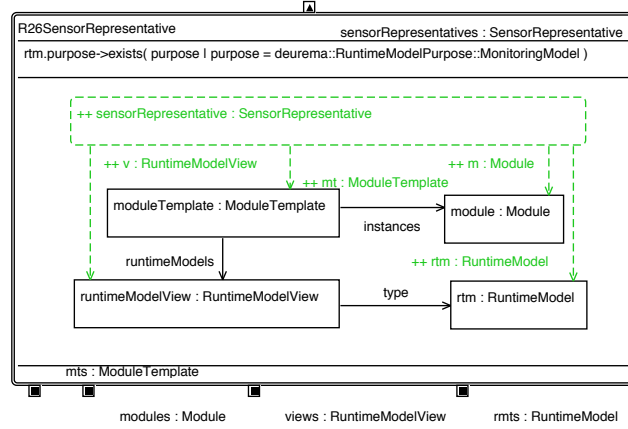


Figure C.31: This rule infers the sensor-representative property by looking on the available local knowledge and the defined purposes. The runtime model view must specify a `MonitoringModel` purpose.

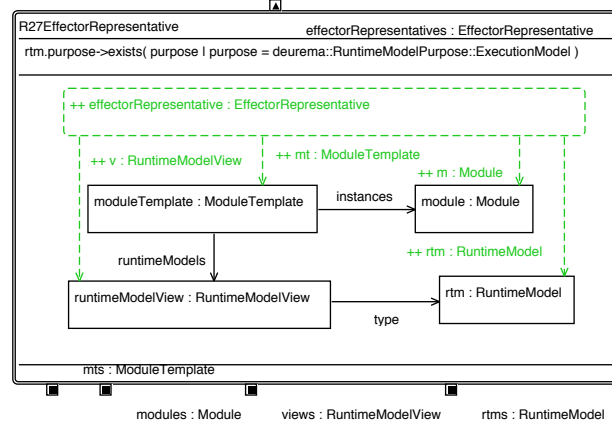


Figure C.32: This rule infers the effector-representative property by looking on the available local knowledge and the defined purposes. The runtime model view must specify a `ExecutionModel` purpose.

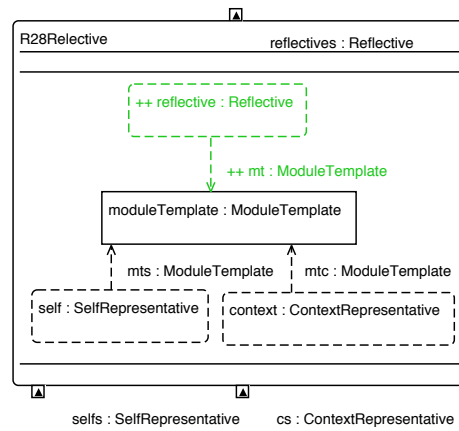


Figure C.33: A module template is considered as reflective, if it is self-representative and context-representative.

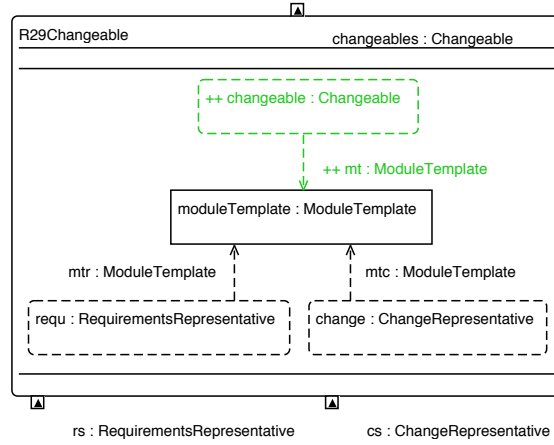


Figure C.34: A module template is considered as changeable, if it is requirements-representative and change-representative.

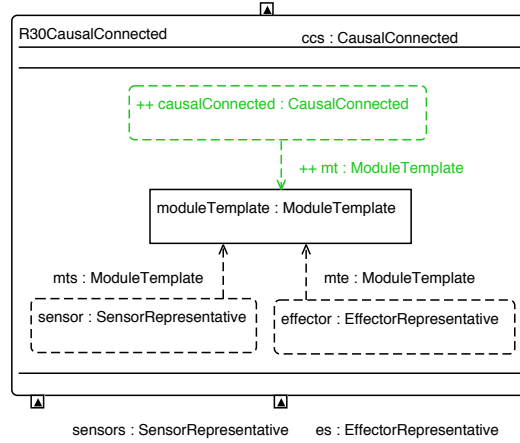


Figure C.35: A module template is considered as causal connected, if it is sensor-representative and effector-representative.

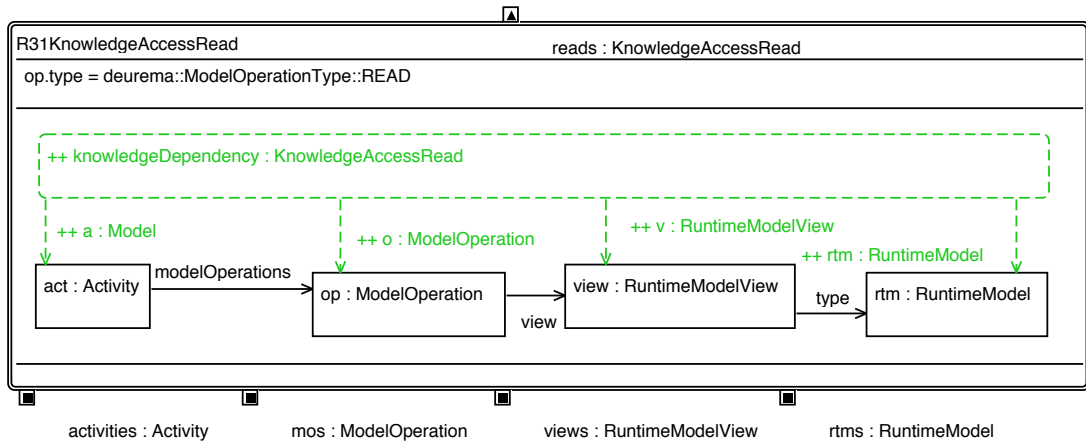


Figure C.36: This rule retrieves all read accesses on the local knowledge base for feedback loop templates.

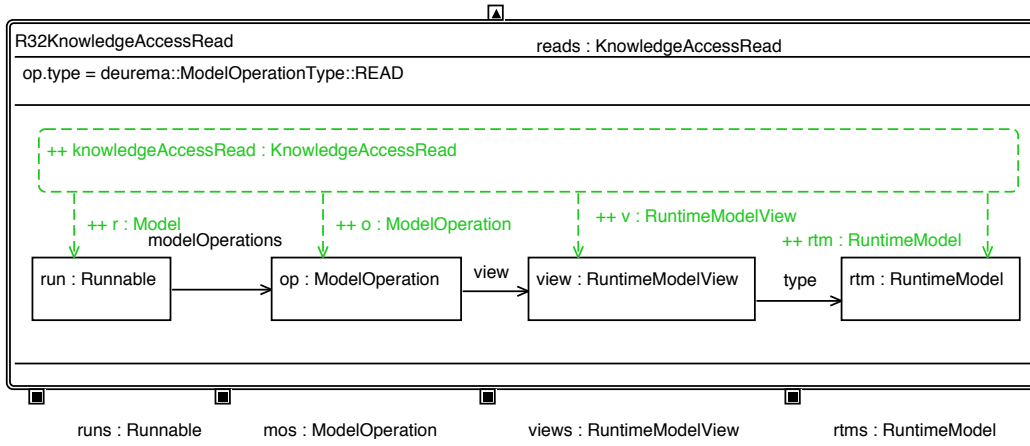


Figure C.37: This rule retrieves all read accesses on the local knowledge base for application module templates.

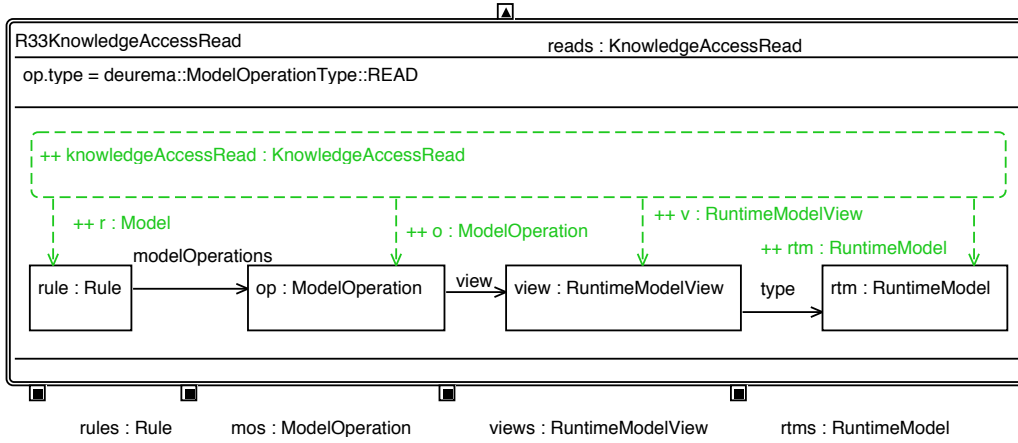


Figure C.38: This rule retrieves all read accesses on the local knowledge base for behavior module templates.

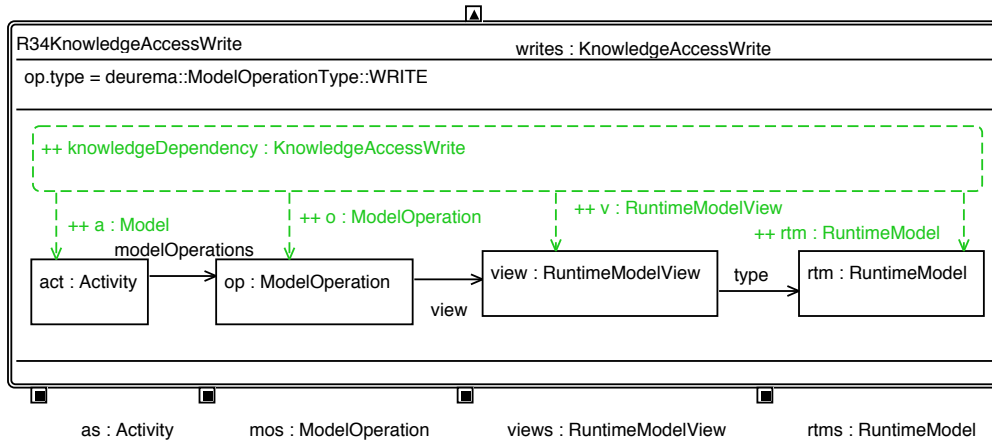


Figure C.39: This rule retrieves all write accesses on the local knowledge base for feedback loop module templates. The analysis rule for behavior rules and runnables follows the same principle.

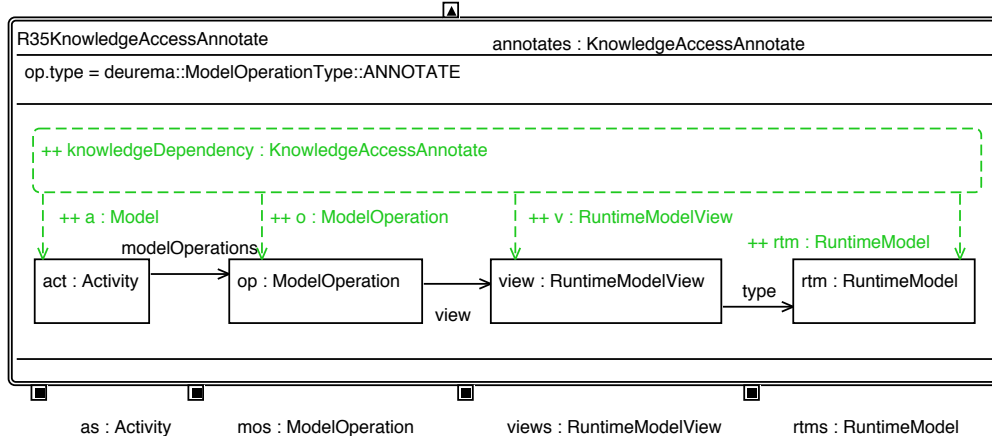


Figure C.40: This rule retrieves all annotate accesses on the local knowledge base for feedback loop module templates.

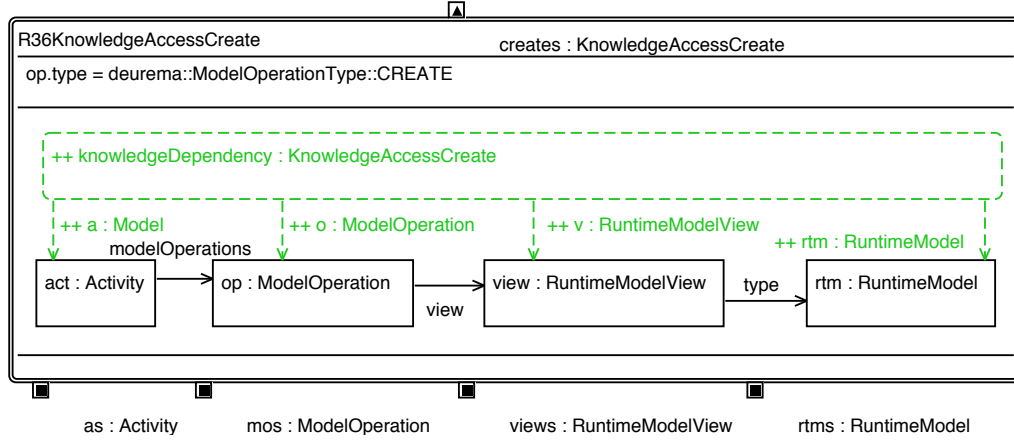


Figure C.41: This rule retrieves all create accesses on the local knowledge base for feedback loop module templates.

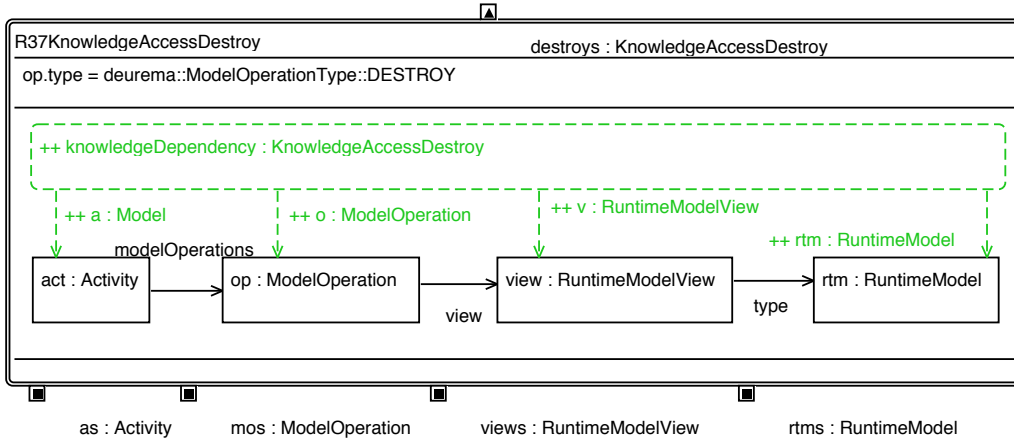


Figure C.42: This rule retrieves all destroy accesses on the local knowledge base for feedback loop module templates.

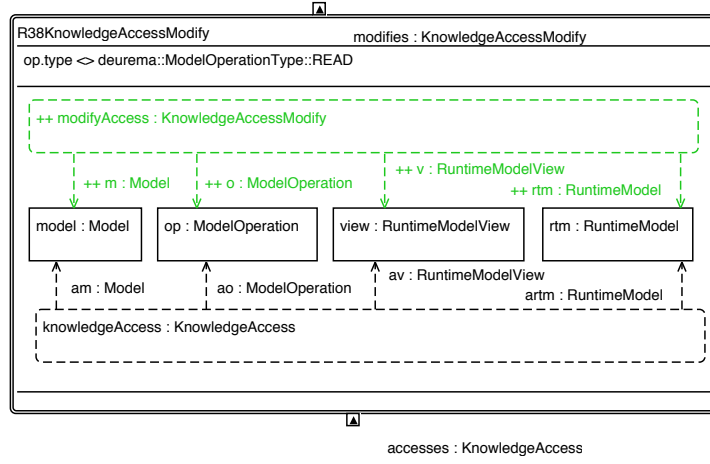


Figure C.43: This rule subsumes all non-read accesses (annotate, write, create, destroy) to a modify access on the corresponding runtime model view of the local knowledge base.

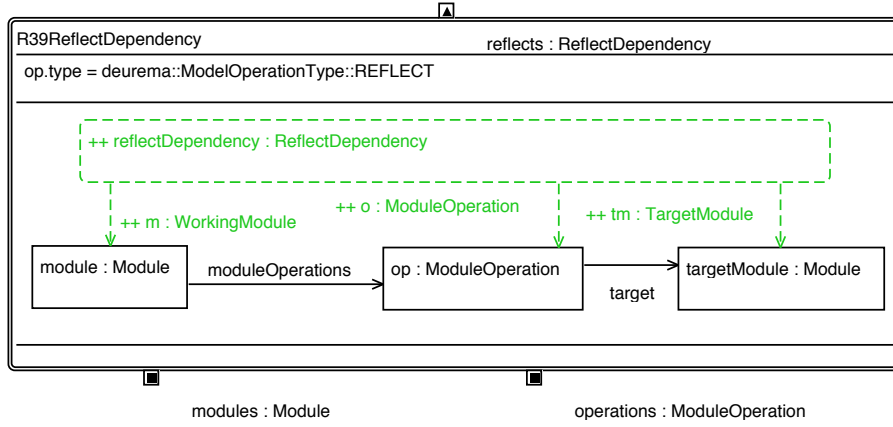


Figure C.44: This rule retrieves reflection dependencies between module instances.

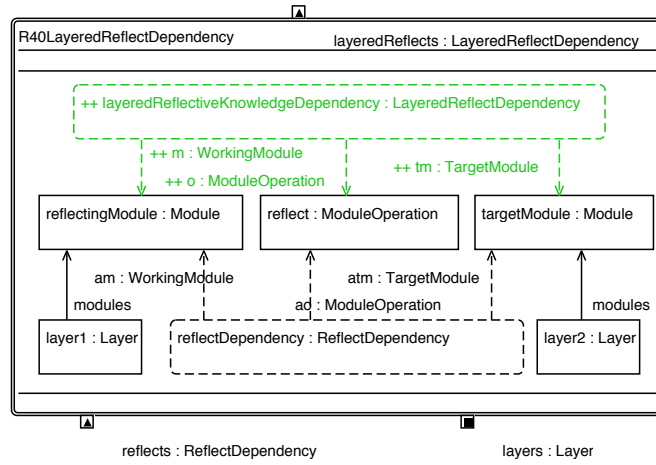


Figure C.45: This rule refines beforehand retrieved reflection dependencies between module instances, if there are two independent layers involved.

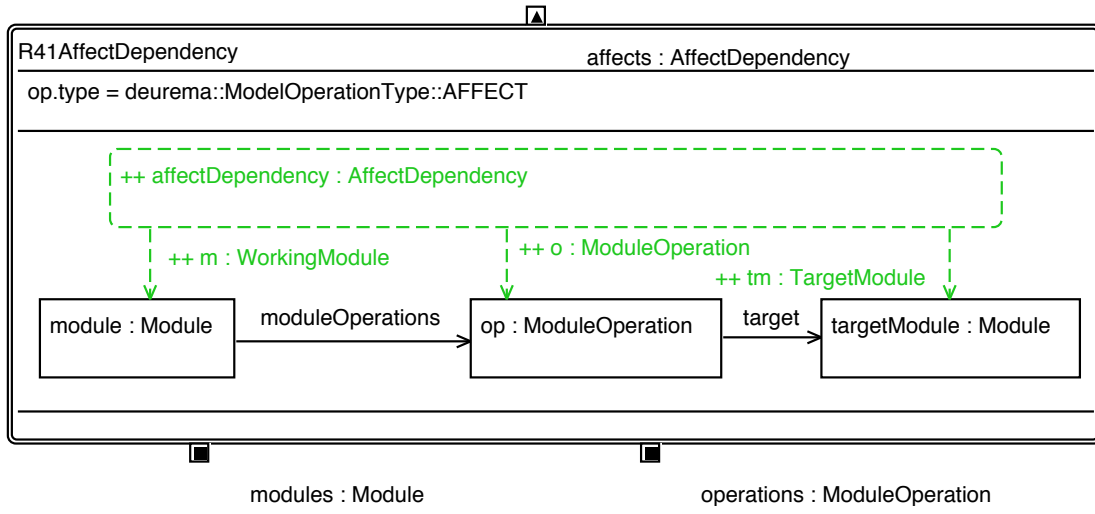


Figure C.46: This rule retrieves affect dependencies between module instances.

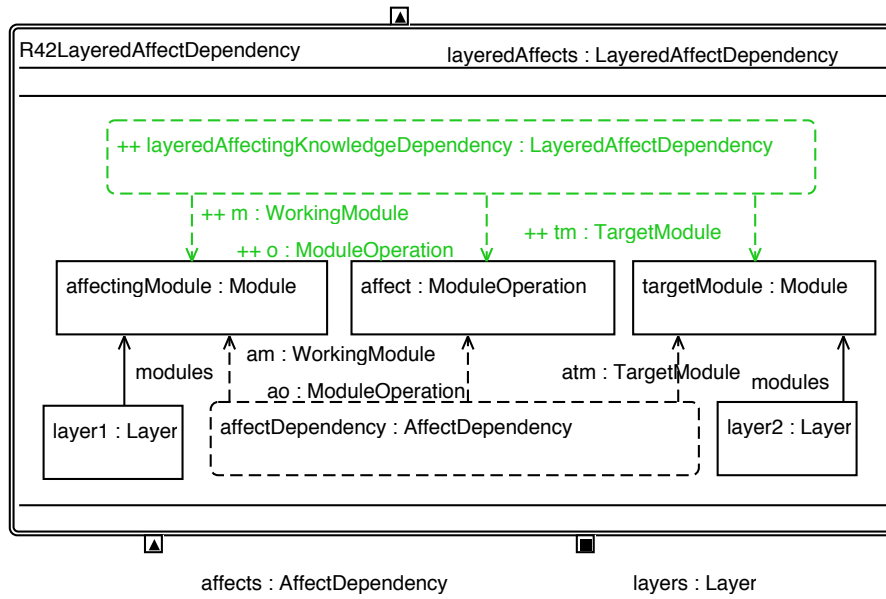


Figure C.47: This rule refines beforehand retrieved affect dependencies between module instances, if there are two independent layers involved.

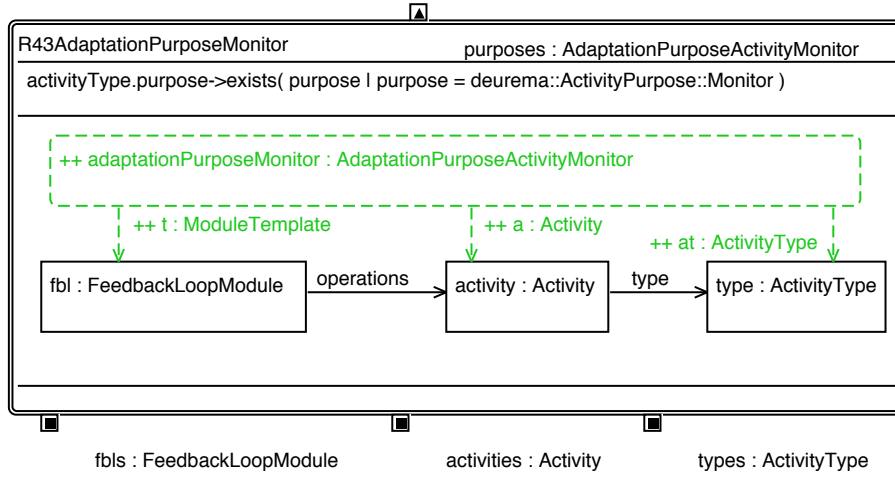


Figure C.48: This rule detects a monitoring activity.

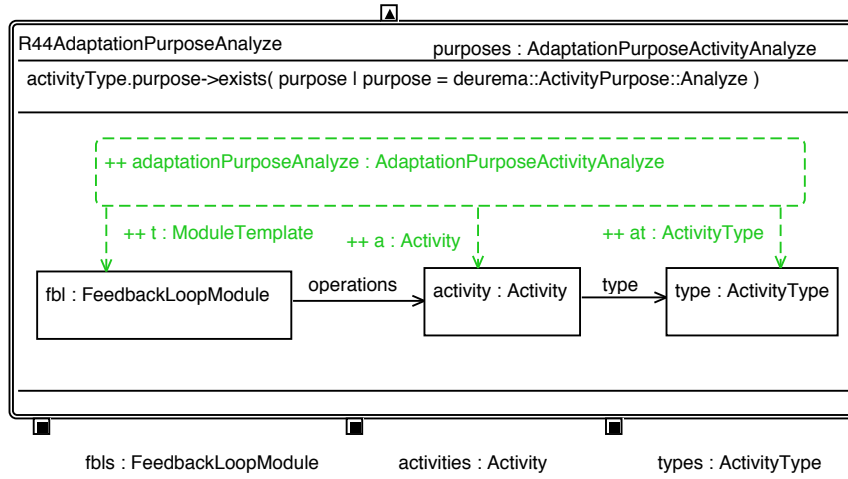


Figure C.49: This rule detects an analyzing activity.

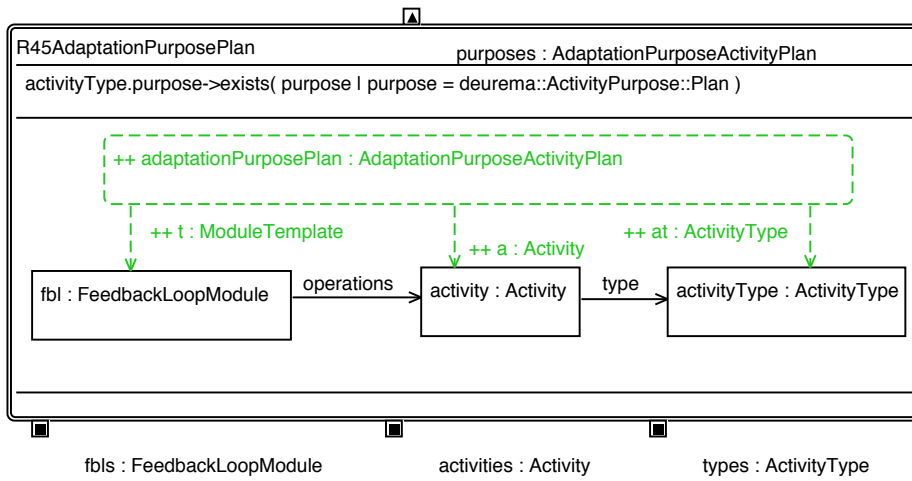


Figure C.50: This rule detects a planning activity.

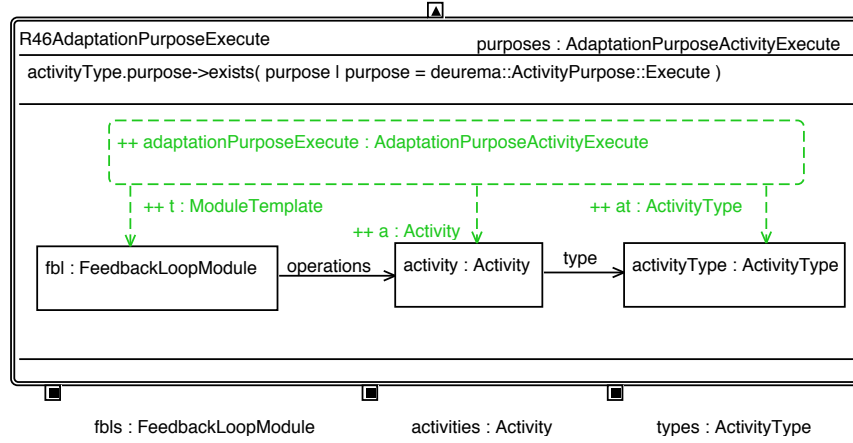


Figure C.51: This rule detects an executing activity.

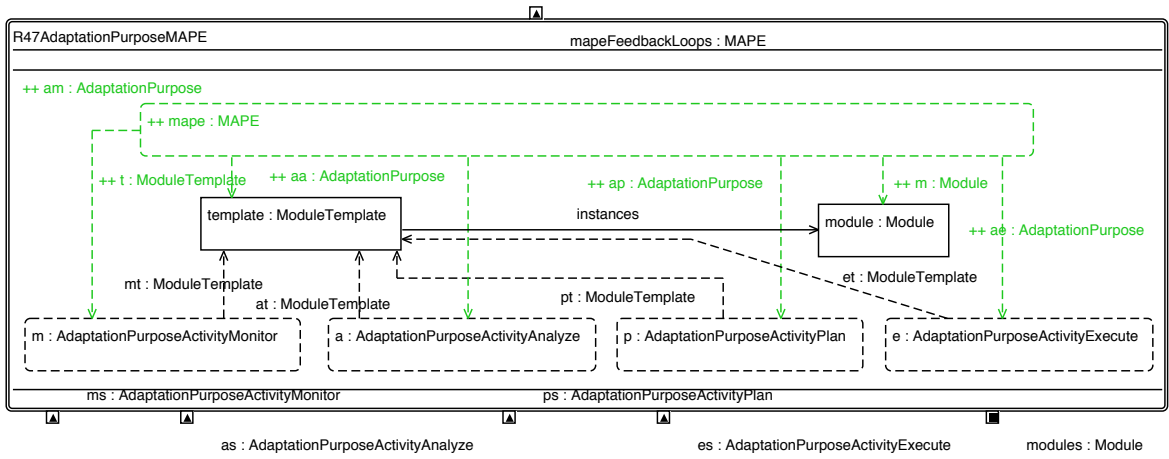


Figure C.52: This rule detects the MAPE feedback loop pattern by means of the adaptation purposes only.

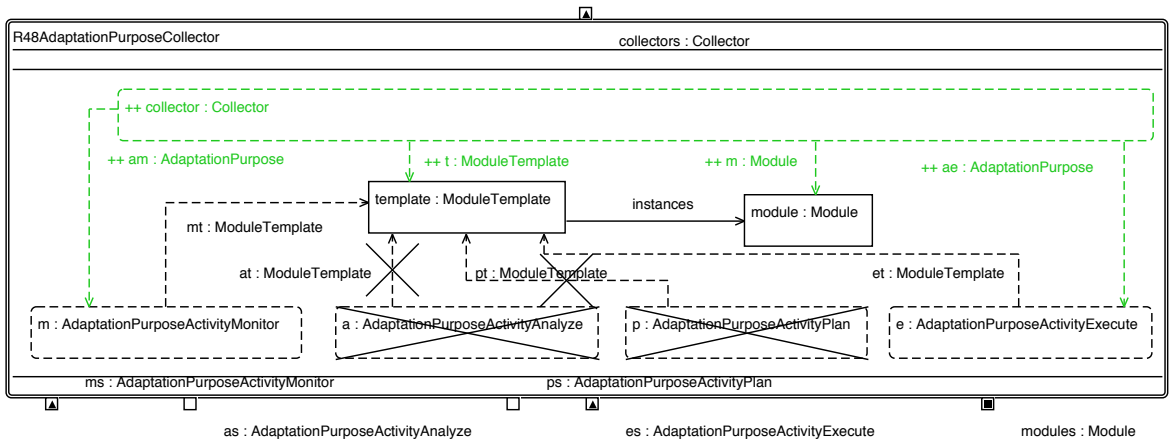


Figure C.53: This rule detects the collector feedback loop pattern by means of the adaptation purposes only.

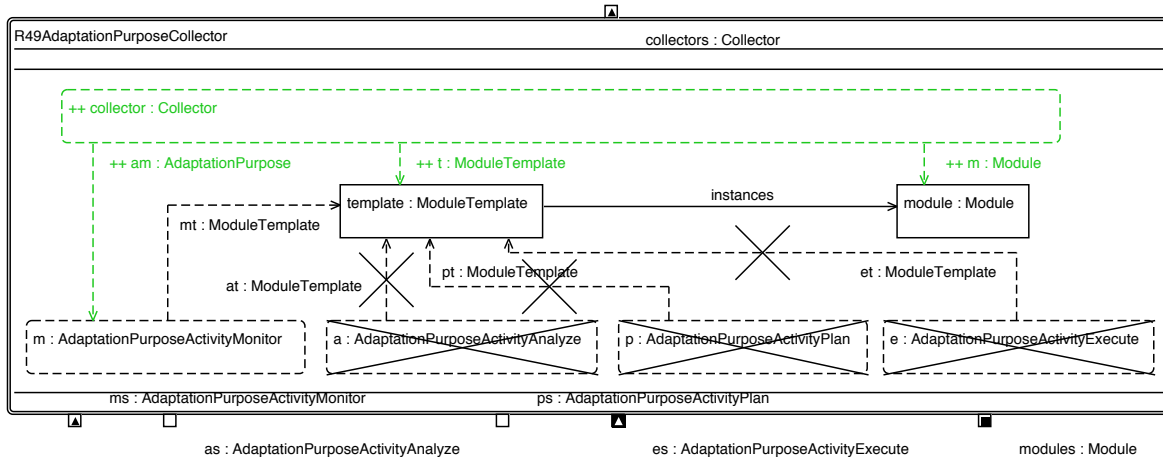


Figure C.54: This rule detects the collector feedback loop pattern by means of the adaptation purposes only.

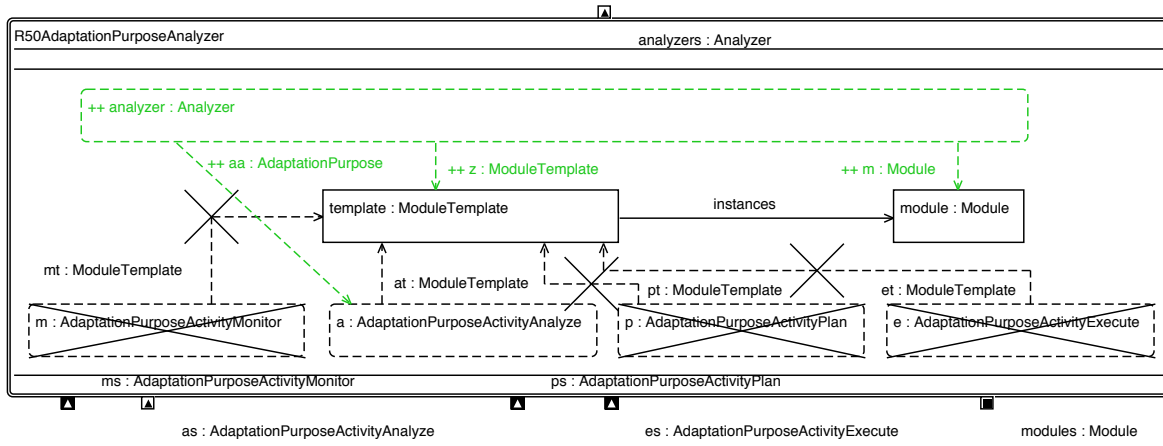


Figure C.55: This rule detects the analyzer feedback loop pattern by means of the adaptation purposes only.

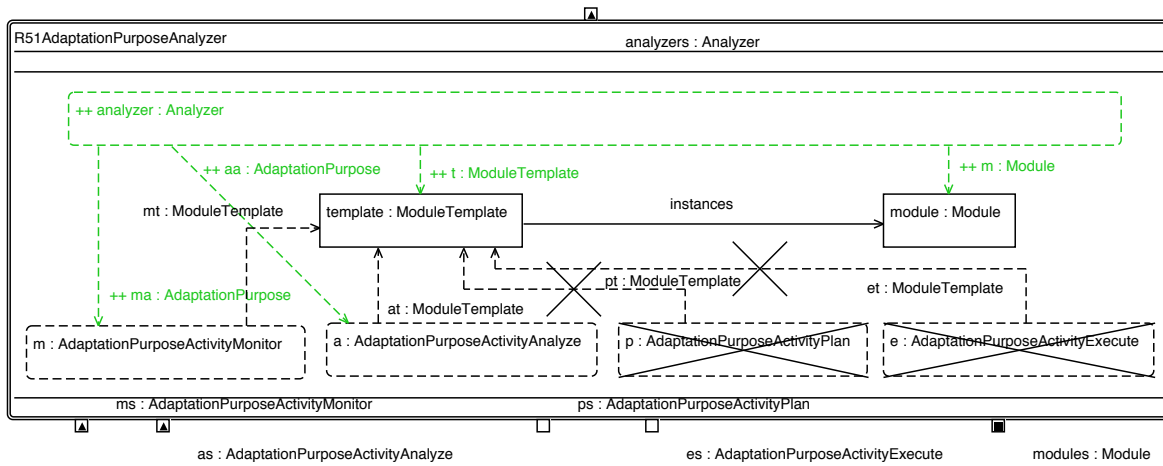


Figure C.56: This rule detects the analyzer feedback loop pattern by means of the adaptation purposes only.

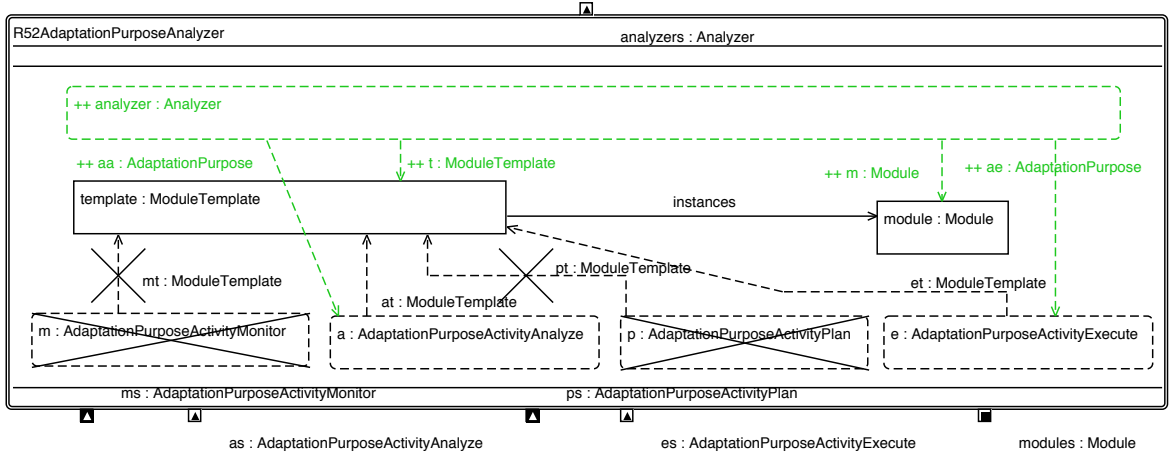


Figure C.57: This rule detects the analyzer feedback loop pattern by means of the adaptation purposes only.

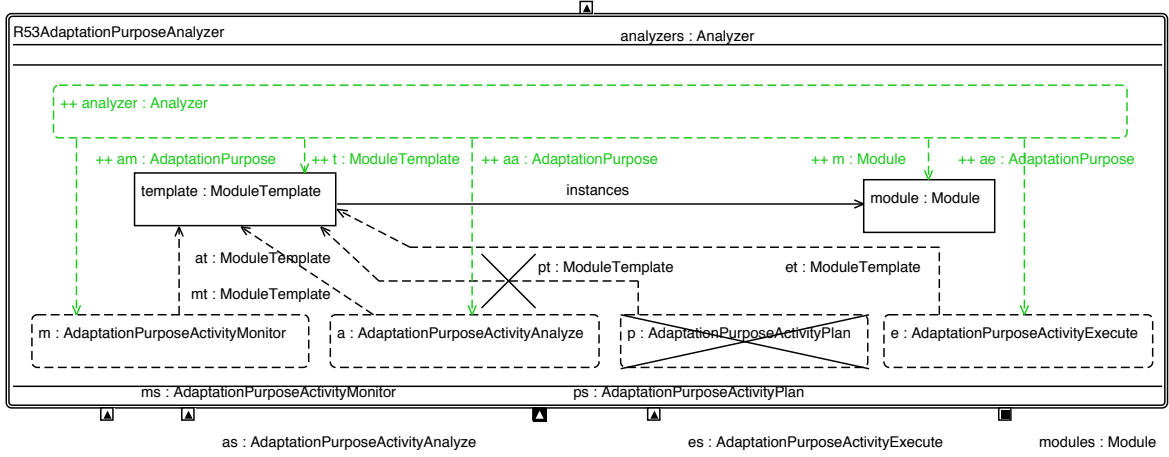


Figure C.58: This rule detects the analyzer feedback loop pattern by means of the adaptation purposes only.

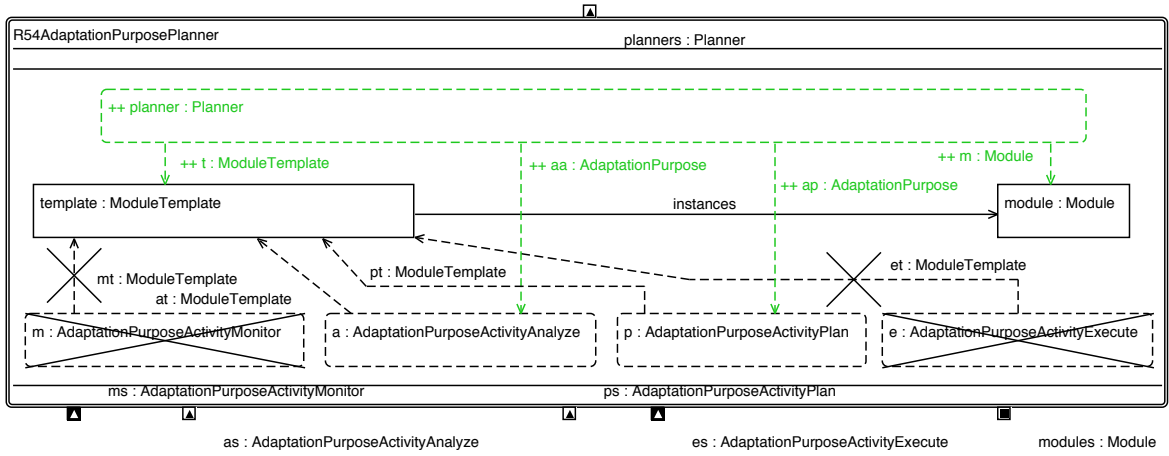


Figure C.59: This rule detects the planner feedback loop pattern by means of the adaptation purposes only.

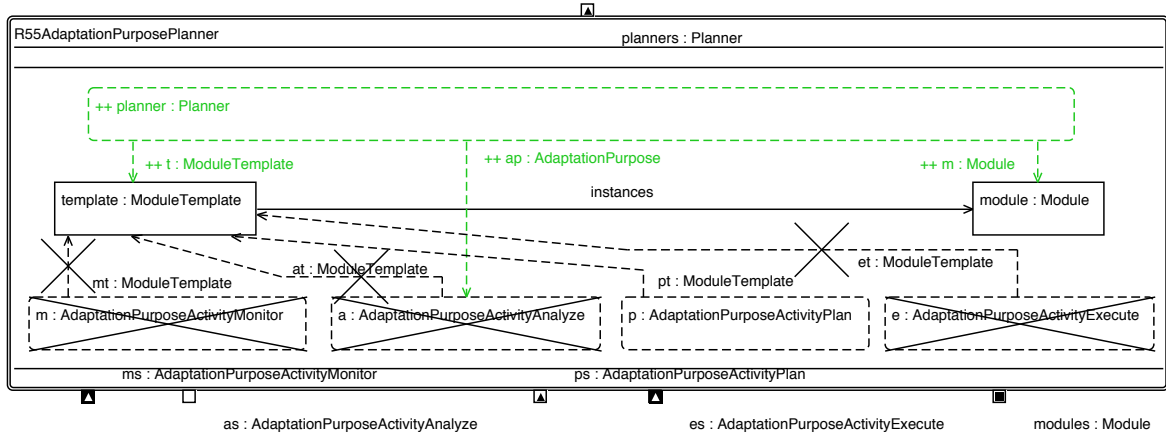


Figure C.60: This rule detects the planner feedback loop pattern by means of the adaptation purposes only.

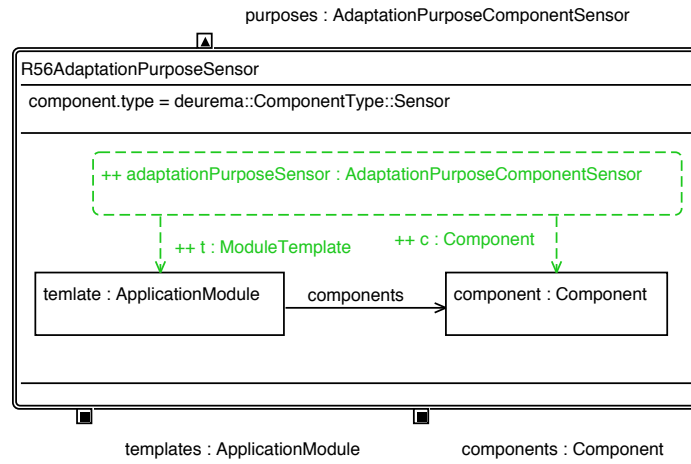


Figure C.61: This rule searches for sensor components in application module templates by looking at the modeled purpose.

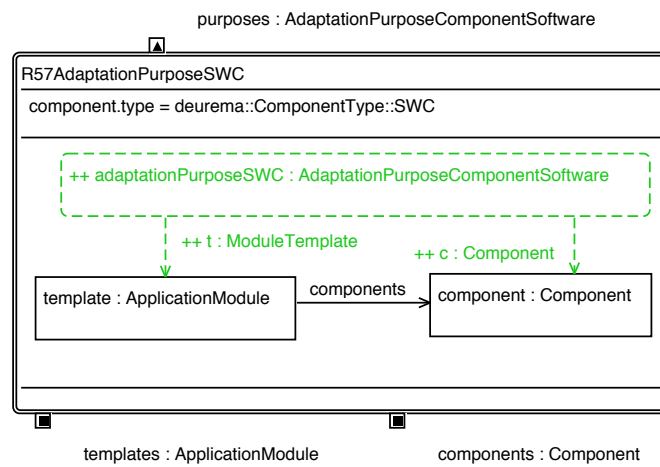


Figure C.62: This rule searches for software components in application module templates by looking at the modeled purpose.

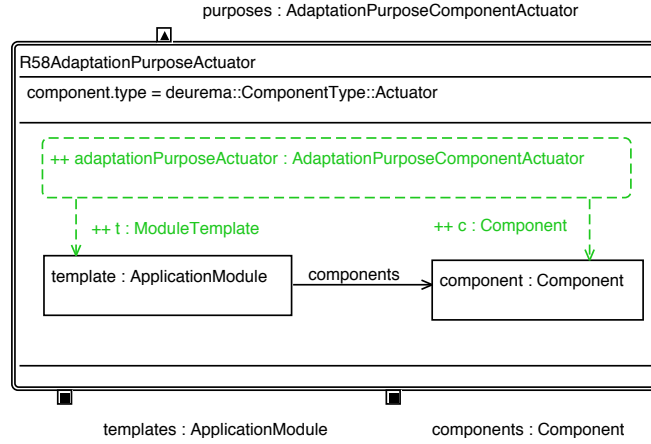


Figure C.63: This rule searches for actuator components in application module templates by looking at the modeled purpose.

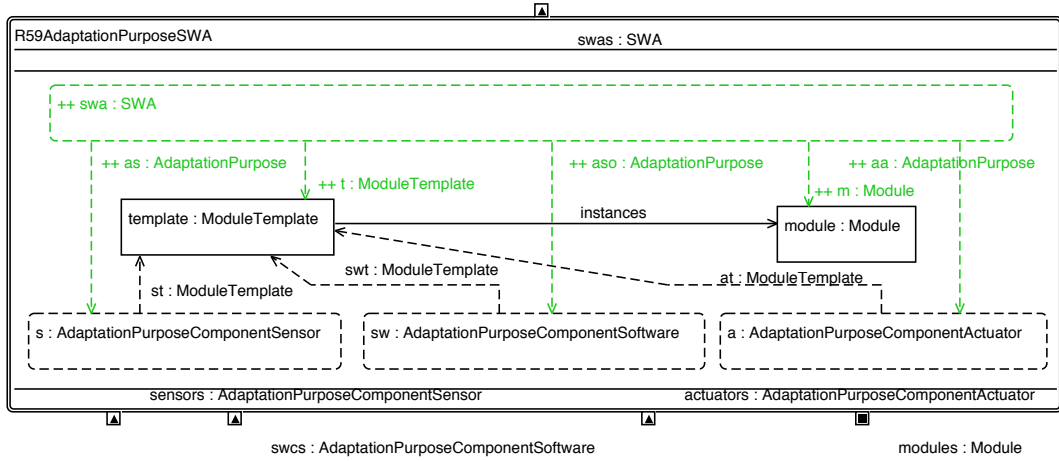


Figure C.64: This rule retrieves the SWA pattern for application module templates by looking on the available component purposes only.

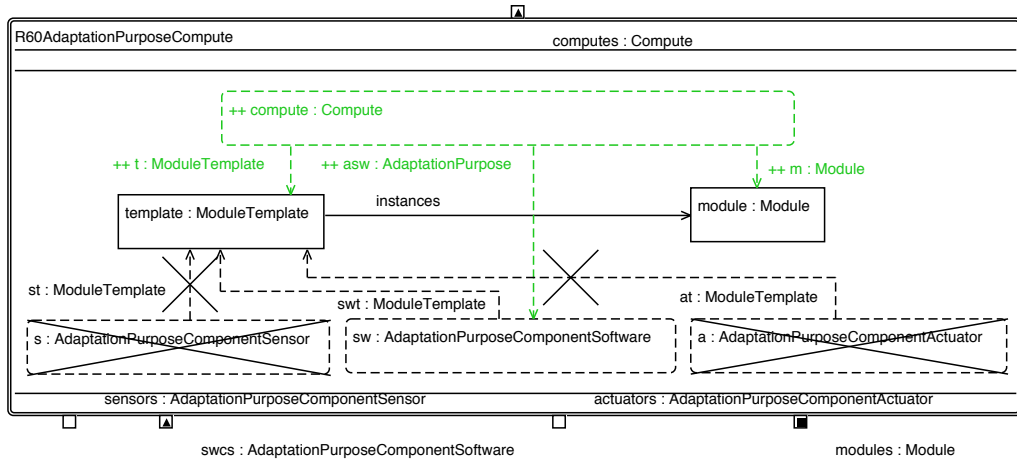


Figure C.65: This rule retrieves the compute pattern for application module templates by looking on the available component purposes only.

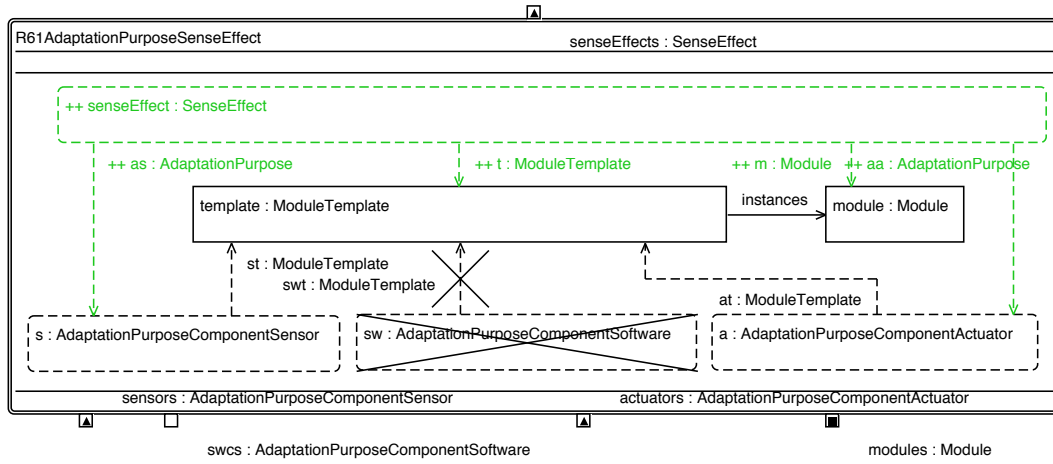


Figure C.66: This rule retrieves the sense-effect pattern for application module templates by looking on the available component purposes only.

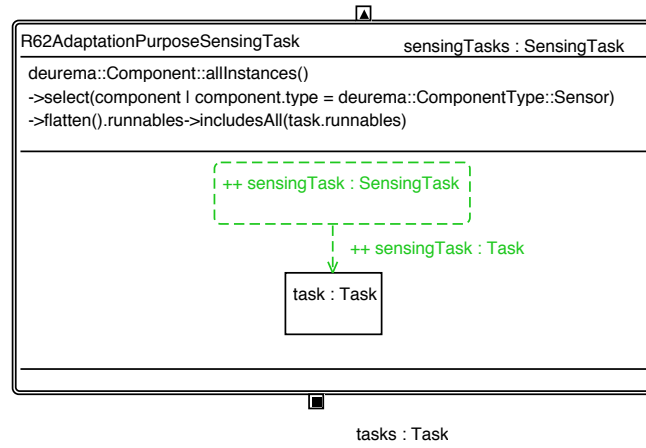


Figure C.67: This rule determines sensing tasks, where each mapped runnable is contained in a sensor component.

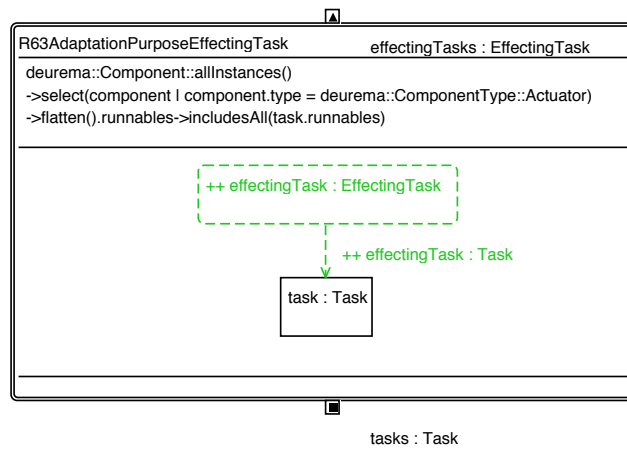


Figure C.68: This rule determines effecting tasks, where each mapped runnable is contained in an actuator component.

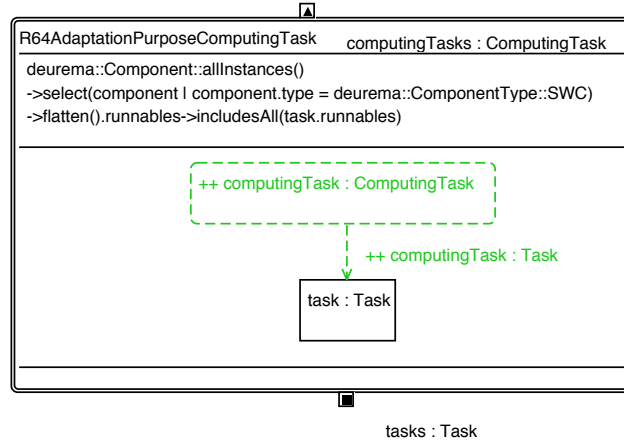


Figure C.69: This rule determines computing tasks, where each mapped runnable is contained in a software component.

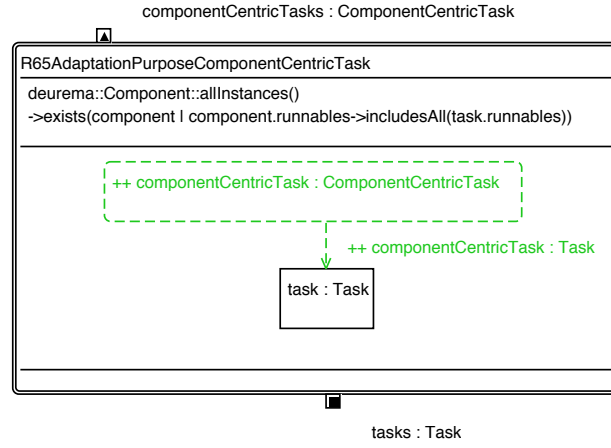


Figure C.70: This rule determines component-centric tasks, where each mapped runnable is contained in the same component.

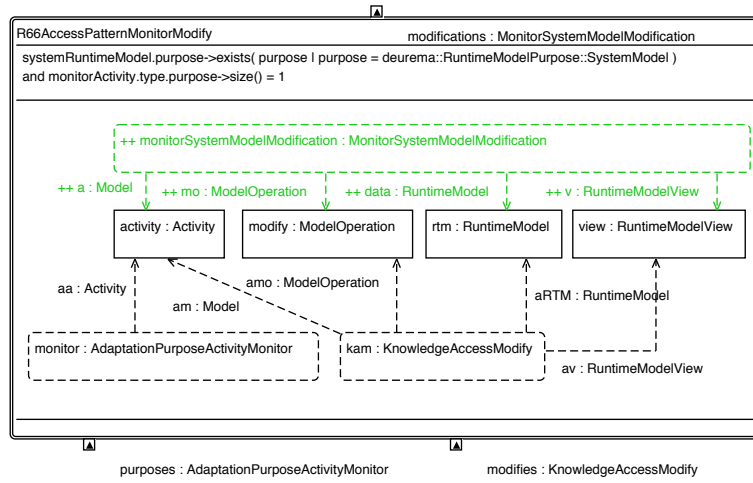


Figure C.71: An exemplary access pattern for an advised modify model operation of a monitor activity.

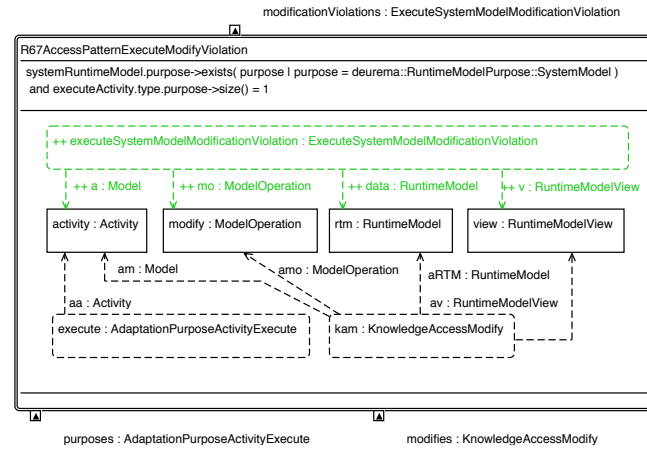


Figure C.72: An exemplary access pattern violation for a modify model operation of an execute activity.

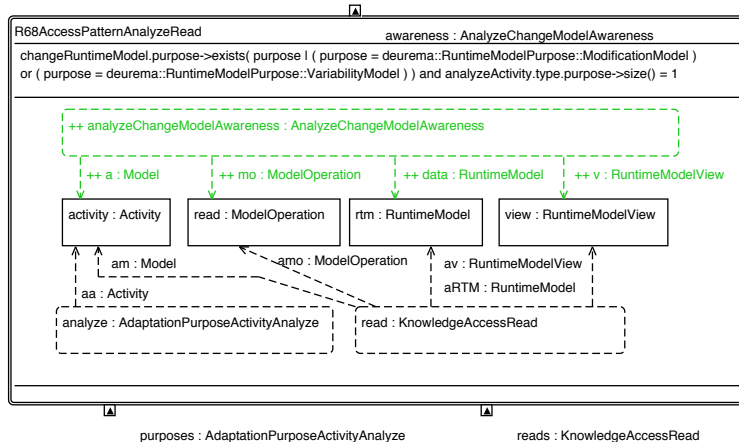


Figure C.73: An exemplary access pattern violation for a modify model operation of an execute activity.

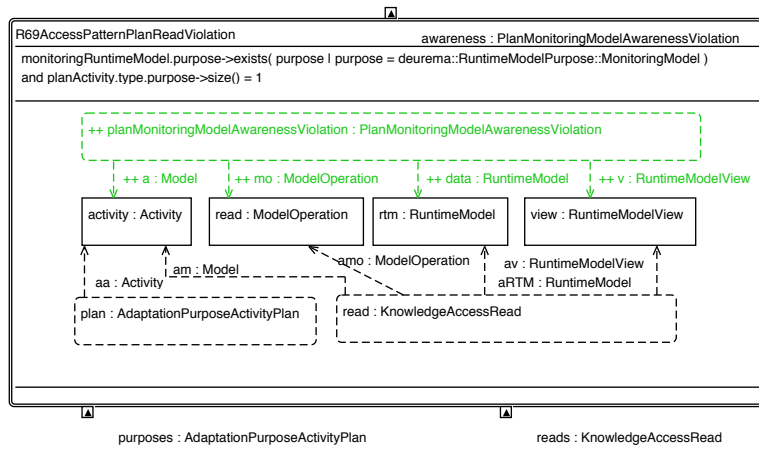


Figure C.74: An exemplary access pattern violation for a read model operation of a plan activity.

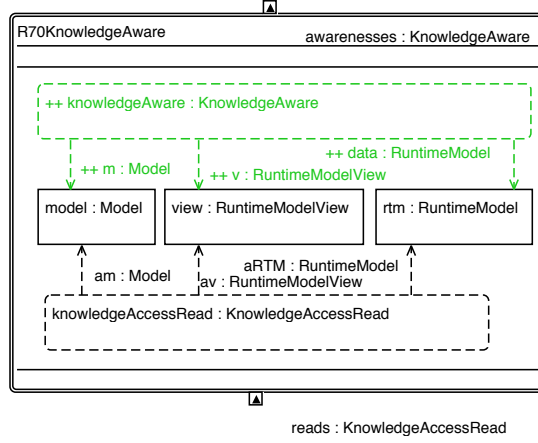


Figure C.75: This rule retrieves the knowledge-awareness property.

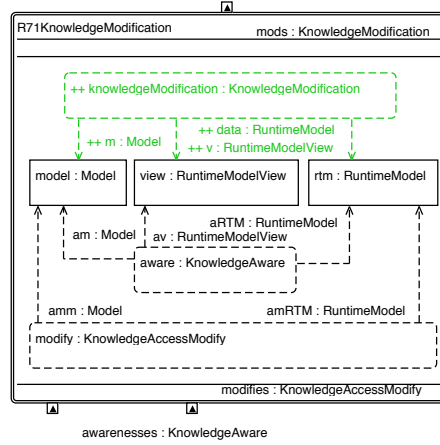


Figure C.76: This rule retrieves the knowledge modifications on basis of the local available runtime model views.

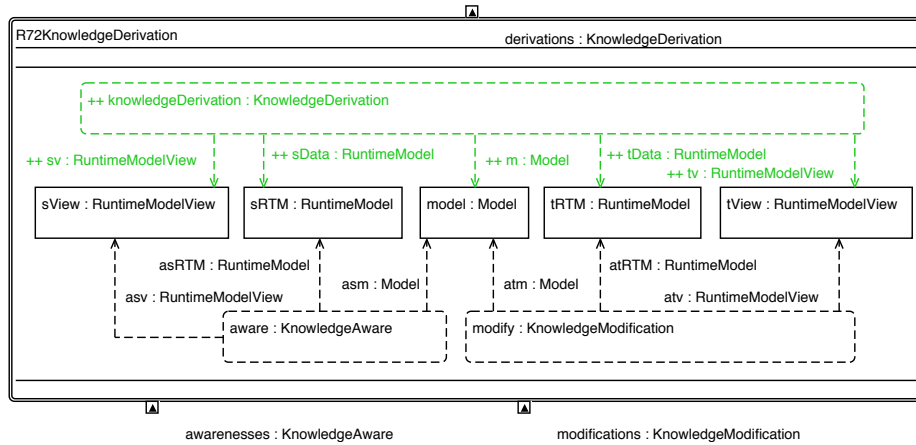


Figure C.77: This rule retrieves the knowledge derivations on basis of the local available runtime model views.

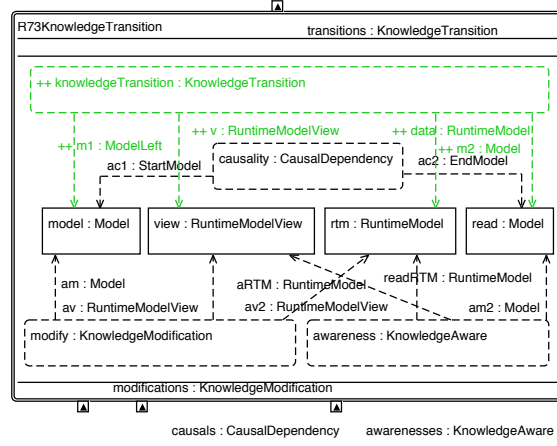


Figure C.78: This rule retrieves the knowledge transitions on basis of the local available runtime model views.

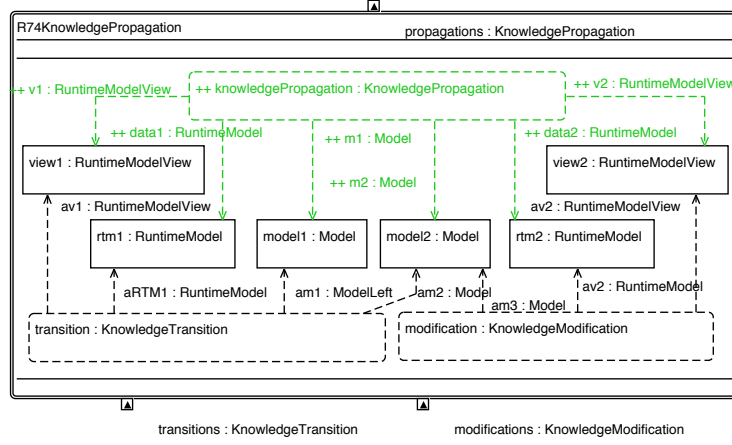


Figure C.79: This rule retrieves the knowledge propagations on basis of the local available runtime model views.

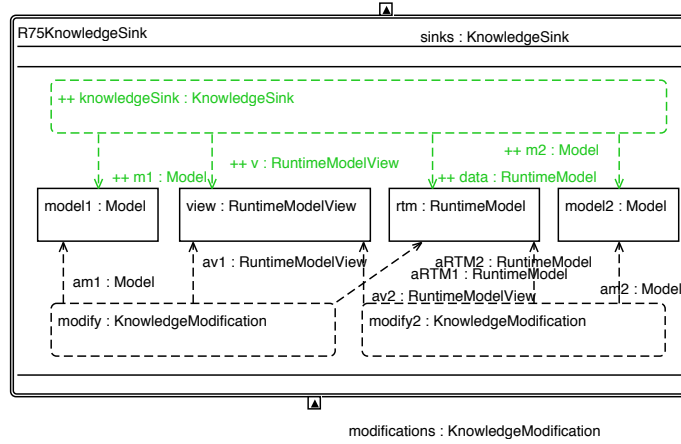


Figure C.80: This rule retrieves the knowledge sinks on basis of the local available runtime model views.

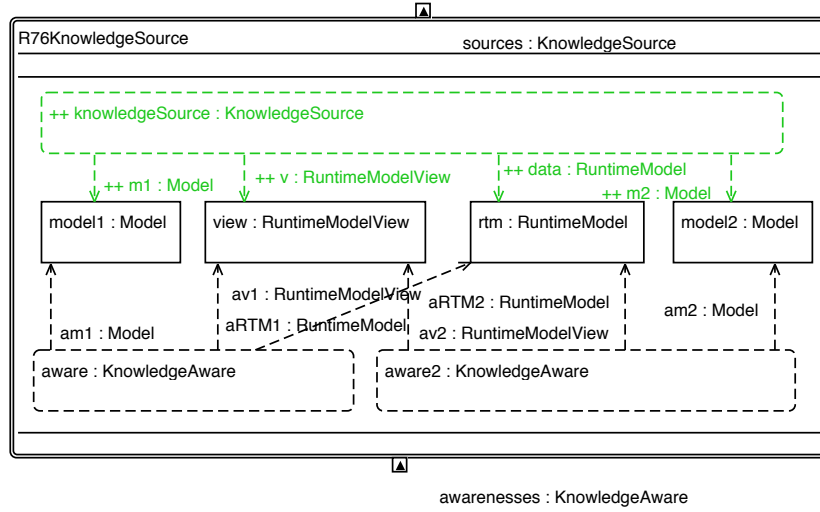


Figure C.81: This rule retrieves the knowledge sources on basis of the local available runtime model views.

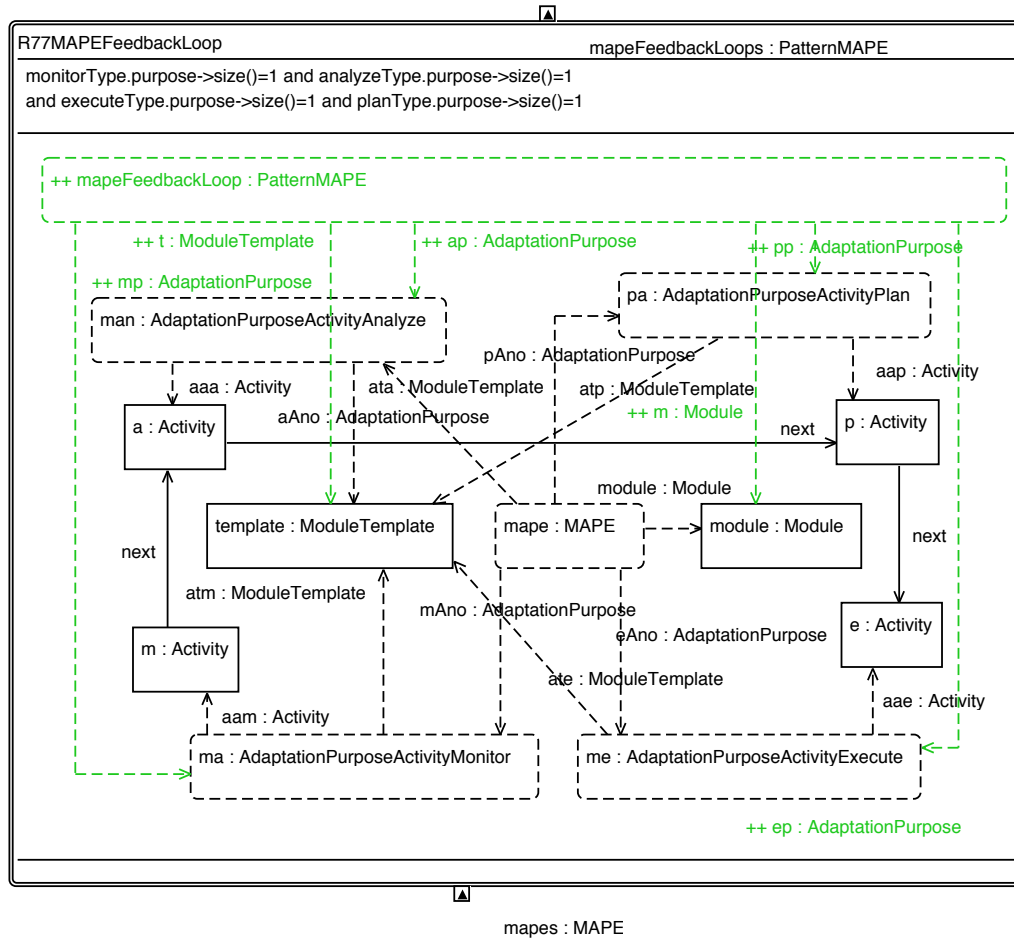


Figure C.82: The rule detects a feedback loop that follows the MAPE blueprint by looking at the adaptation purposes and the control flow.

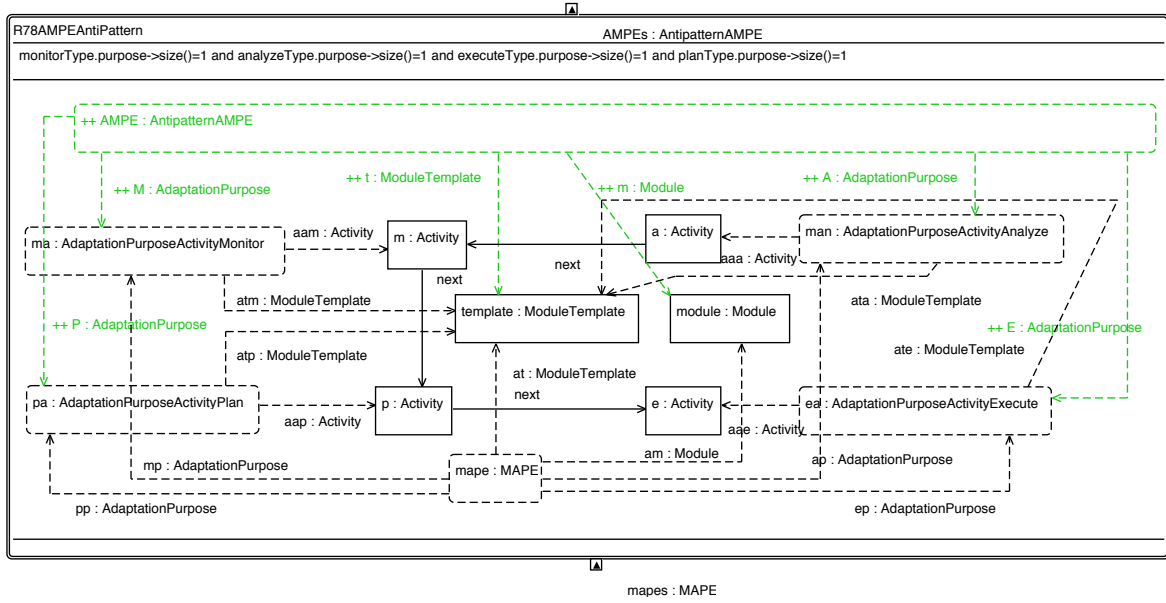


Figure C.83: An example rule for retrieving anti-patterns of modeled feedback loops. In this case a AMPE combination is retrieved.

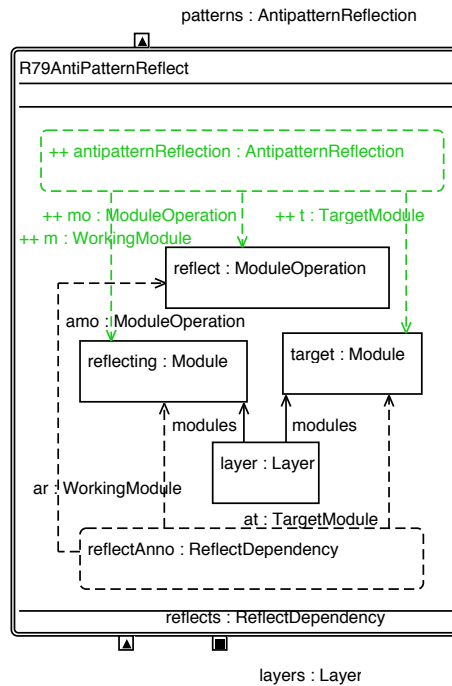


Figure C.84: This rule detects a design flaw of using a reflect module operation on the same layer.

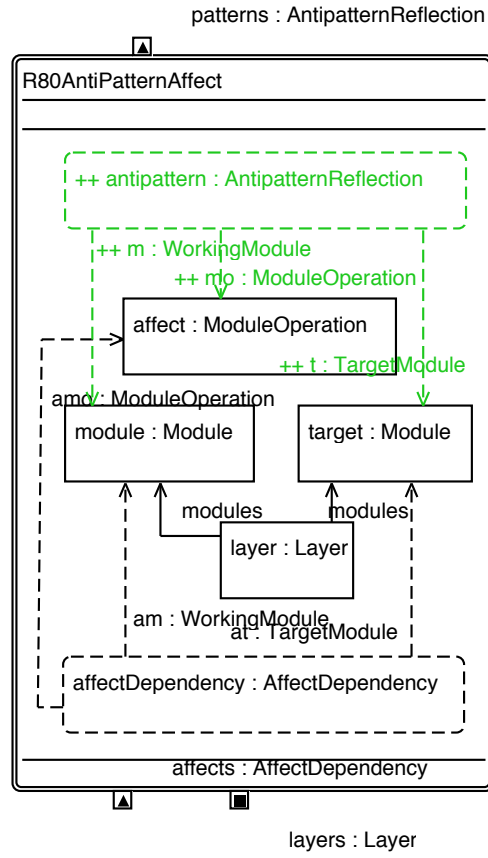


Figure C.85: This rule detects a design flaw of using an affect module operation on the same layer.

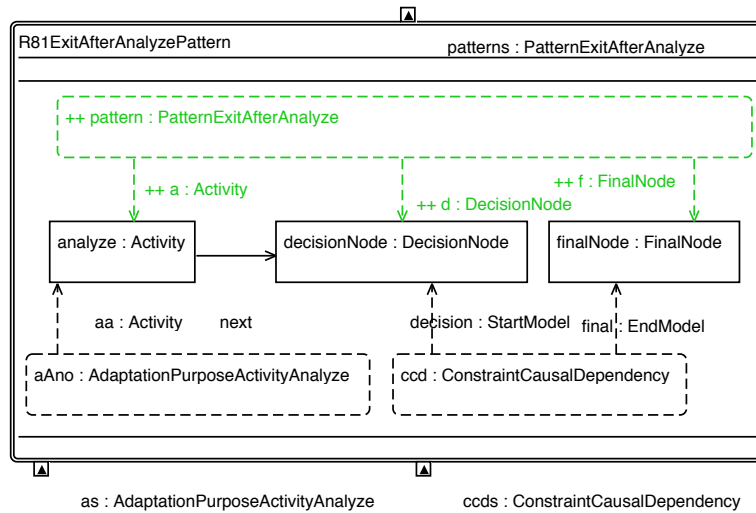


Figure C.86: This rule detects the exit after analyze pattern in a feedback loop module template.

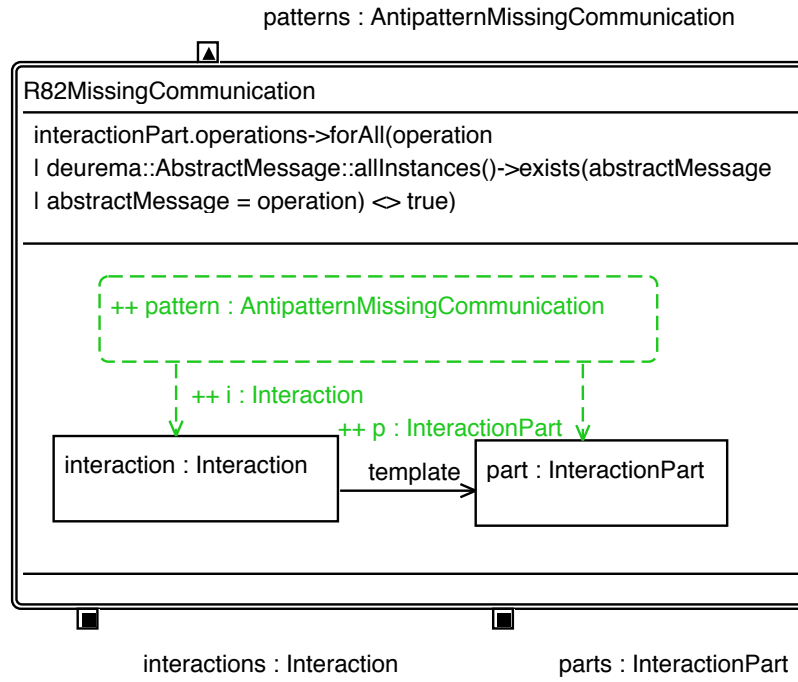


Figure C.87: This rule retrieves all interactions that have no message communication defined, which is a design flaw.

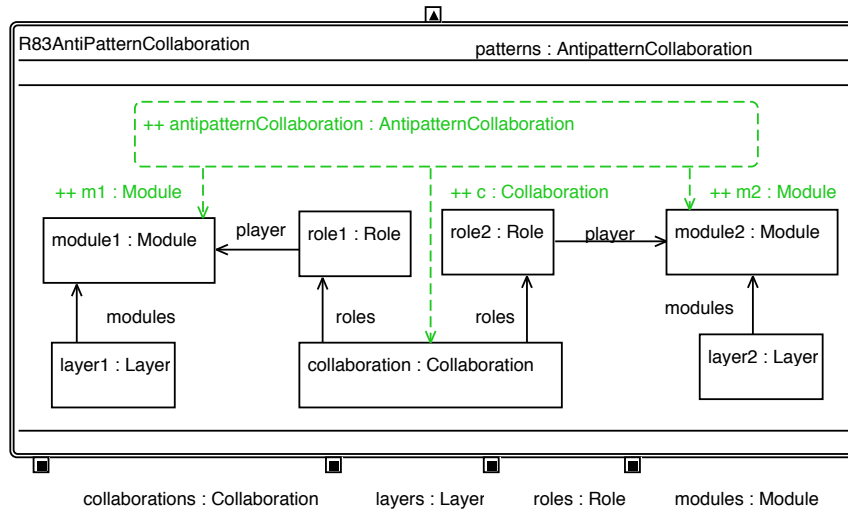
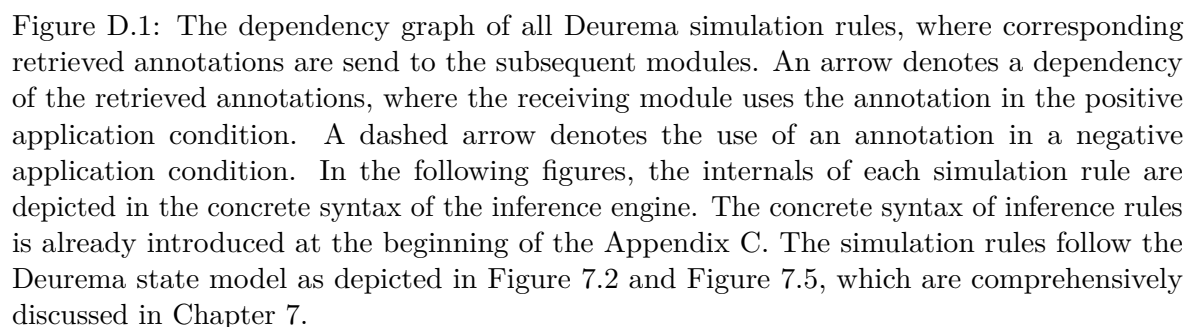


Figure C.88: This rule detects the design flaw, where a collaboration is used between different layers in the adaptive SoS architecture.

D.1. Simulation Rules



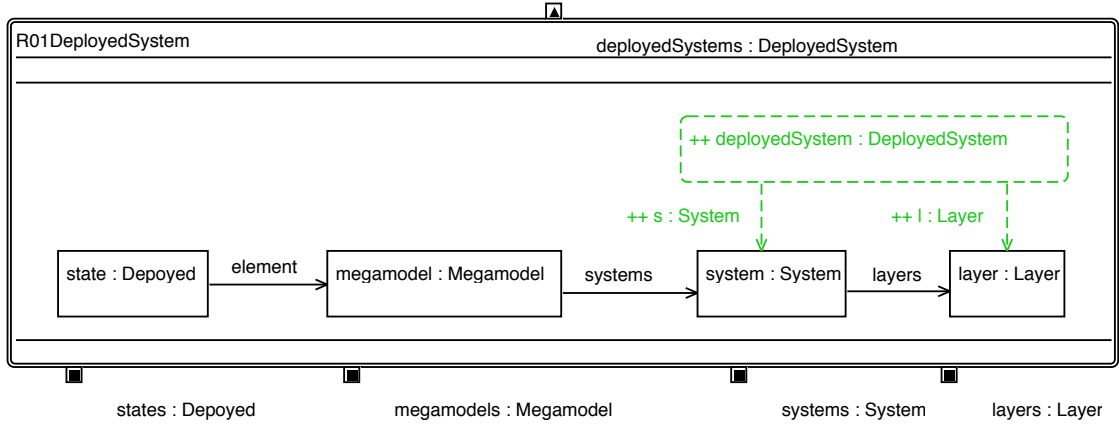


Figure D.2: This rule searches for all deployed systems in the megamodel and annotates each system and corresponding layer combination. The annotation corresponds to the deployed state.

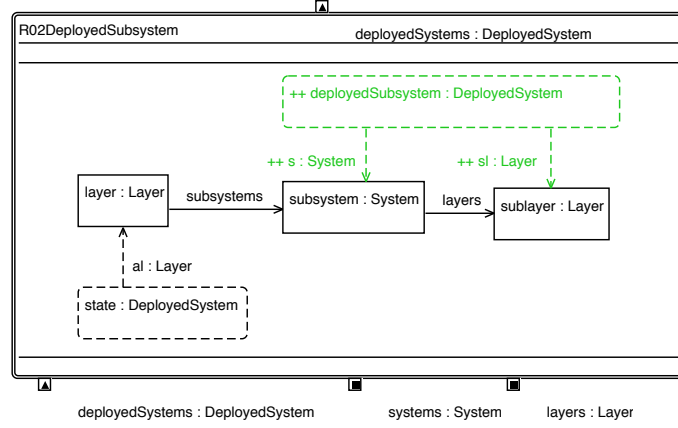


Figure D.3: This rule recursively retrieves deployed subsystems based on the found deployed systems of the inference rule R01DeployedSystem in Figure D.2. All subsystems and corresponding layer combinations are annotated. The annotation corresponds to the deployed state.

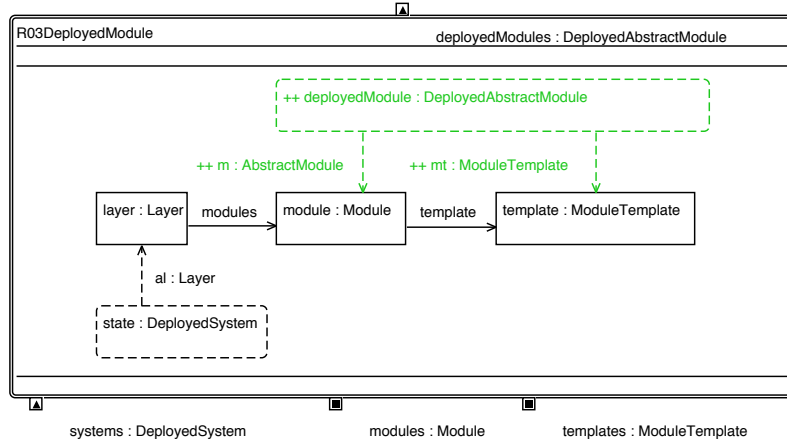


Figure D.4: Based on all retrieved deployed systems, this rule searches for all deployed modules and its corresponding template description. Each module is annotated with a deployed state.

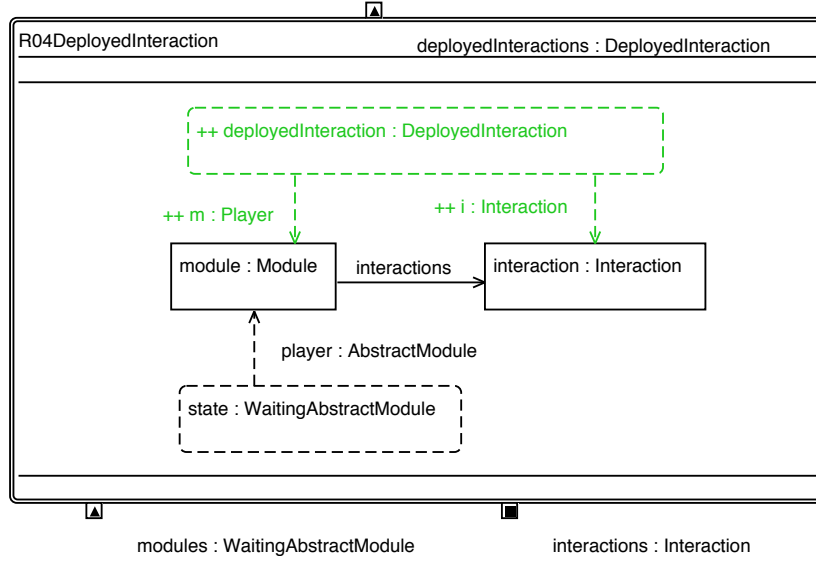


Figure D.5: This rules searches for a module that realizes an interaction as defined by the collaboration role mapping. Only those interactions that are played by a module are important for the simulation and thus, marked as *deployed* for further processing.

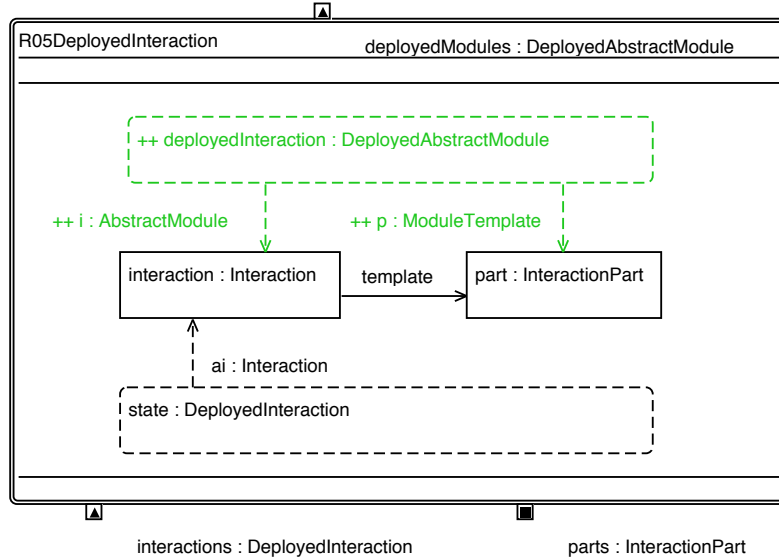


Figure D.6: The corresponding template specification is retrieved for all deployed interactions, which is defined in the collaboration choreography specification of the corresponding interaction role part.

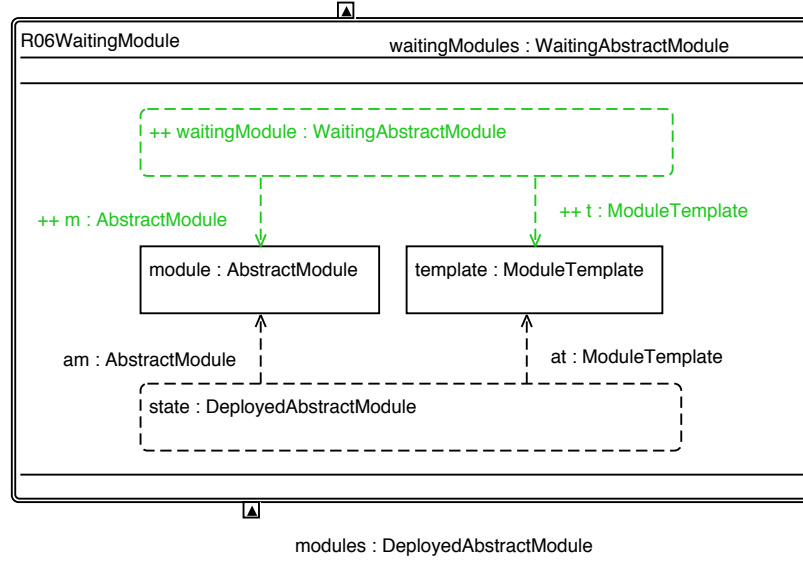


Figure D.7: Each deployed module and interaction (represented by the `AbstractModule` object in the pattern) change in the waiting state during the build process of the underlying megamodel. A corresponding state annotation marks the module respectively interaction instance and the template description.

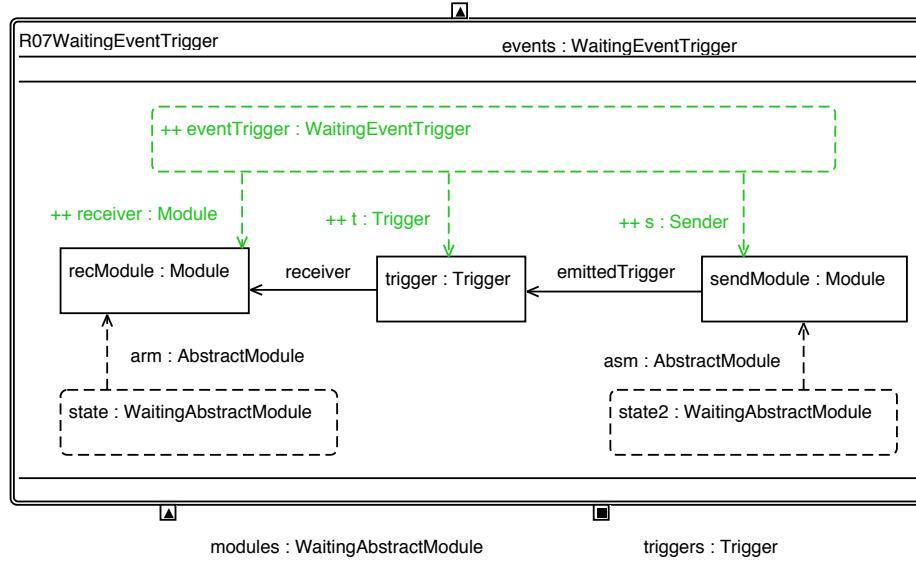


Figure D.8: A module can trigger another module, which defines a causal order between both modules. This rule detects all direct trigger dependencies between deployed modules, which enables a correct detection of an emitted event trigger during a simulation run. The rule annotates the sender and receiver module instance as well as the corresponding event trigger.

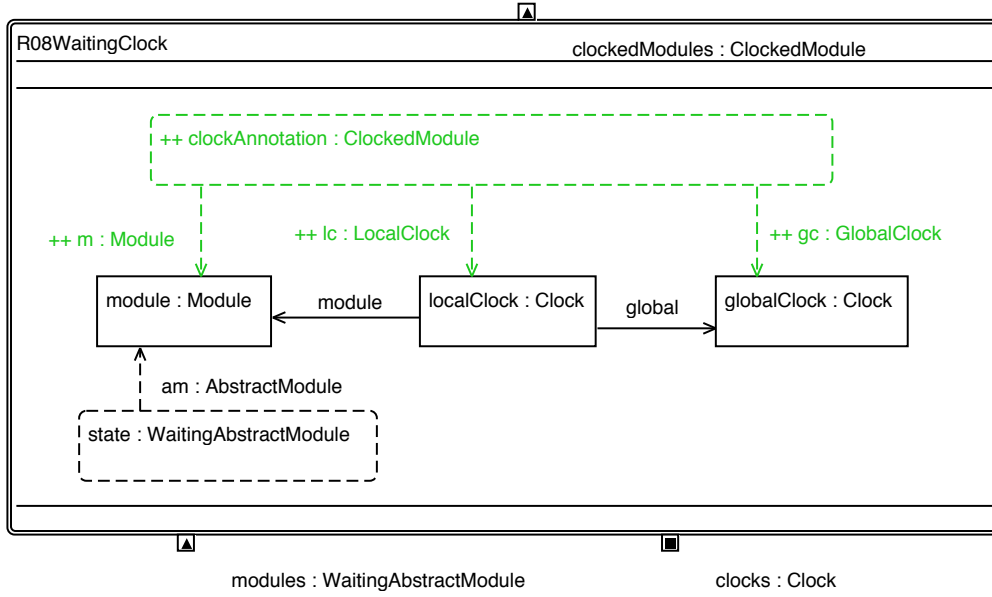


Figure D.9: During the simulation, each module maintains its own local clock. Furthermore, each local clock has a reference to the global simulation clock. This rule retrieves the individual clocks and annotates the corresponding module instance. This clock annotation is important for the retrieval of an enabled timed trigger condition belonging to the module instance.

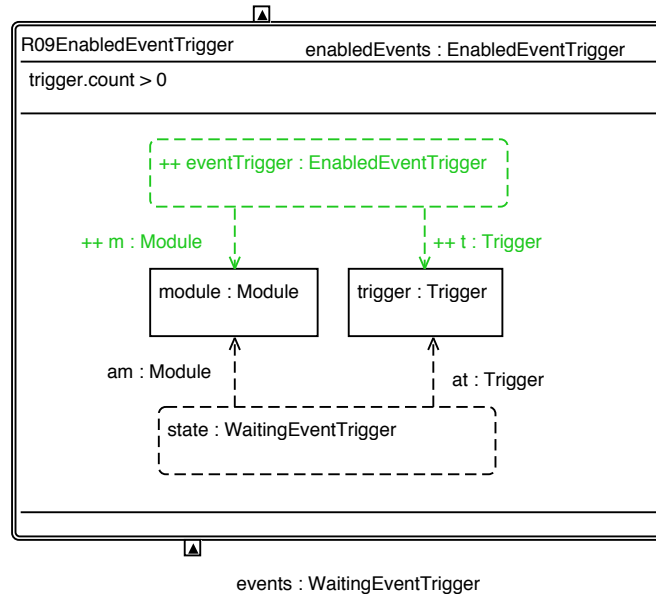


Figure D.10: The rule detects enabled event trigger by means of the trigger count. A trigger count greater zero indicates that another module emits at least one trigger event. The rule annotates the receiving module as well as the corresponding event trigger. The rule depends on the beforehand found causal trigger dependencies between modules.

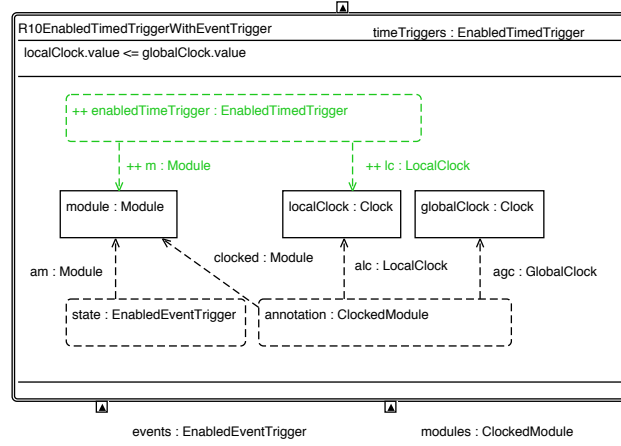


Figure D.11: This rule checks the trigger conditions for each module instance by determining event trigger and timed trigger.

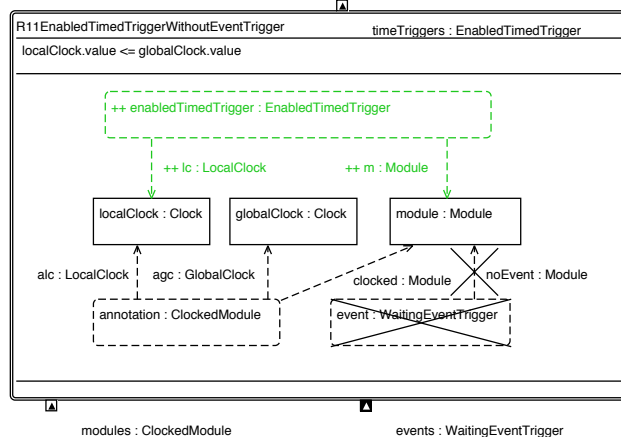


Figure D.12: This rule checks the trigger conditions for each module instance by determining timed trigger only, where a local clock implies a corresponding timed trigger.

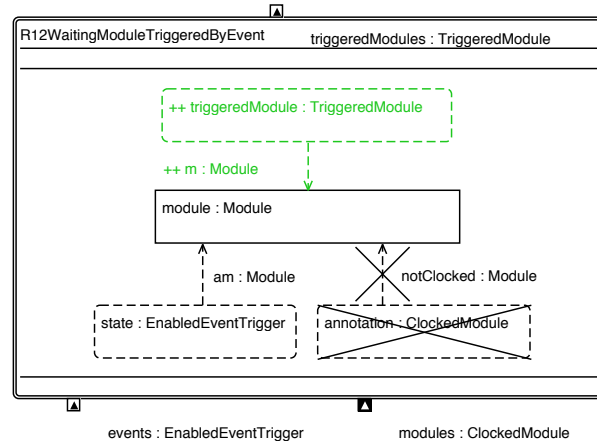


Figure D.13: This rule checks for a successfully triggered module by determining module instances that have only specified incoming event triggers. The negative application condition ensures that no timed trigger is specified.

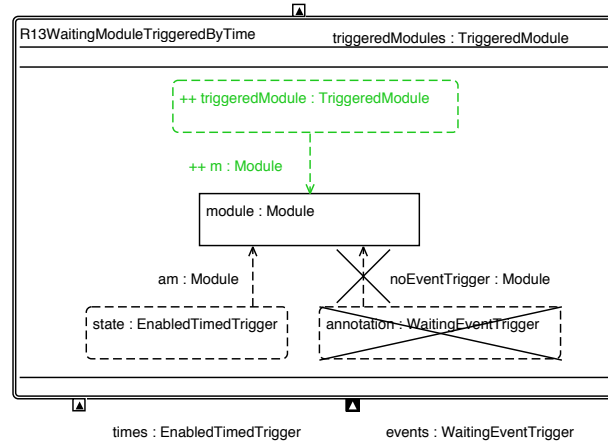


Figure D.14: This rule checks for a successfully triggered module by determining module instances that have only one specified timed trigger. The negative application condition ensures that no event triggers are specified.

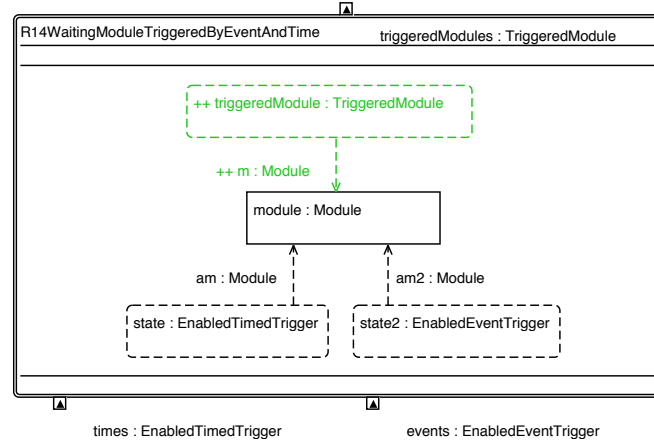


Figure D.15: This rule checks for a successfully triggered module by determining module instances that have specified timed and event triggers.

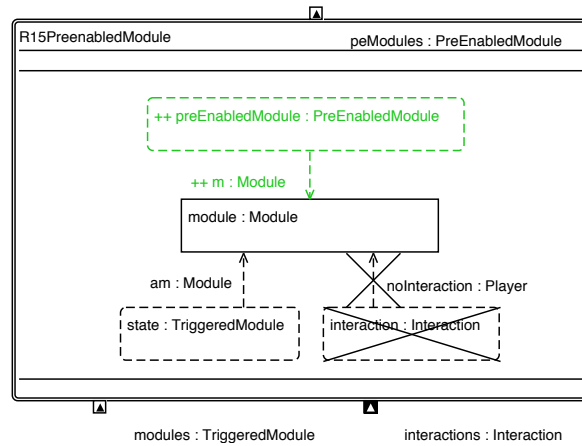


Figure D.16: After a module instance is successfully triggered, it becomes preenabled if no active collaboration interaction is performed. This rule annotates the module instance accordingly.

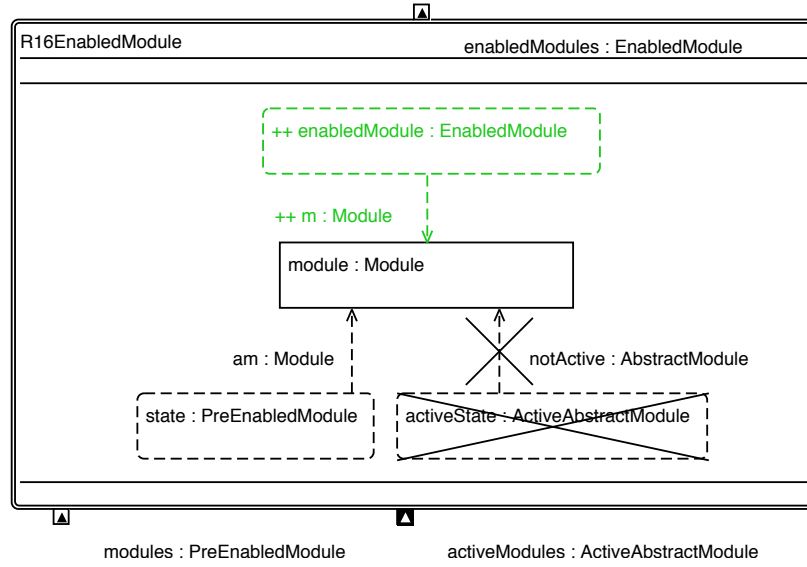


Figure D.17: This rule detects a preenabled module instance, which is not active ensured by the negative application condition. The module becomes enabled and can be picked by the Deurema scheduler.

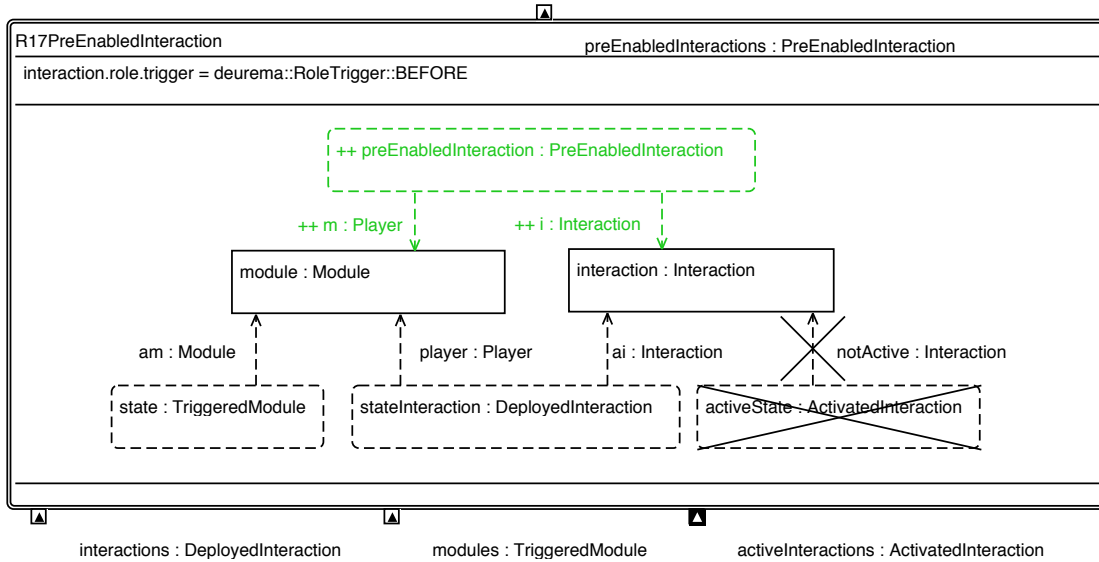


Figure D.18: The rule infers preenabled interactions, where the corresponding player module has a role trigger denoting the execution of the collaborative behavior before the execution of the local adaptation behavior. The player module must already be successfully triggered.

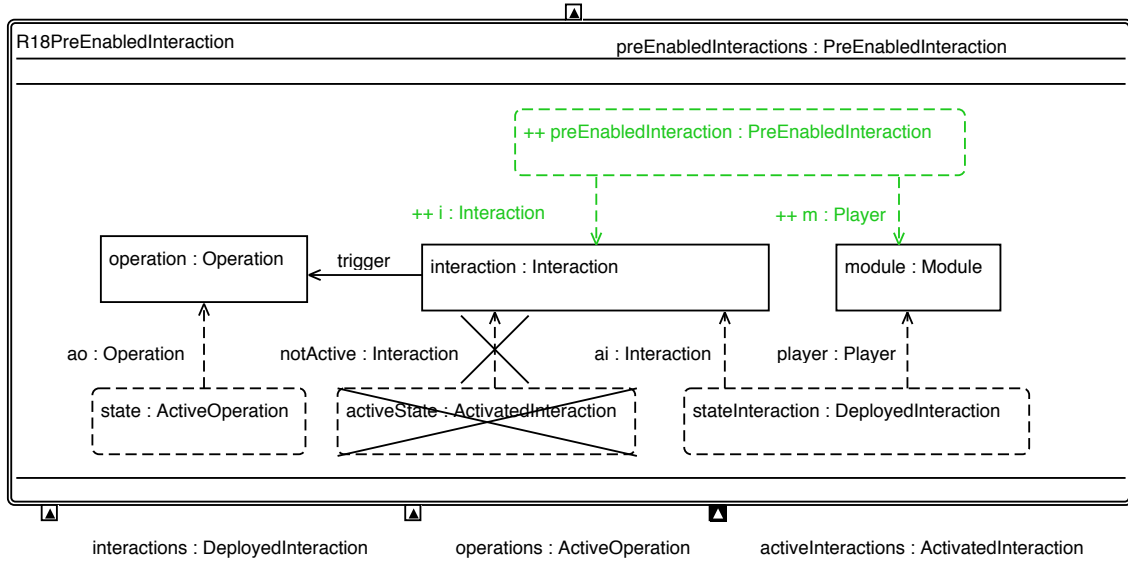


Figure D.19: The rule infers preenabled interactions, where the corresponding player module integrates the collaboration behavior into the local adaptation behavior. This rule focus on feedback loop templates and searches for an active operation that triggers the corresponding interaction instance.

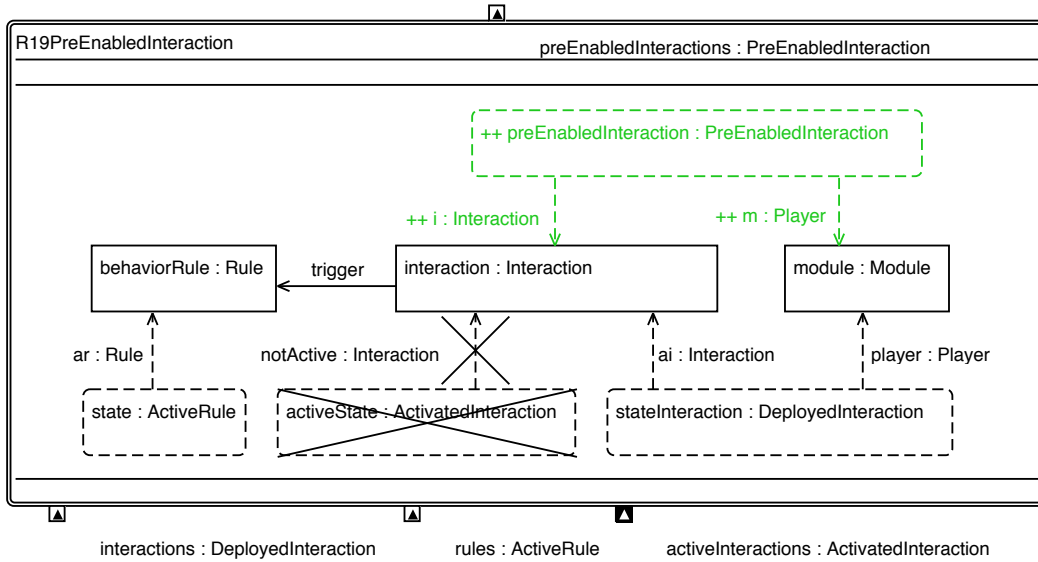


Figure D.20: The rule infers preenabled interactions, where the corresponding player module integrates the collaboration behavior into the local adaptation behavior. This rule focus on behavior rule templates and searches for an active rule that triggers the corresponding interaction instance.

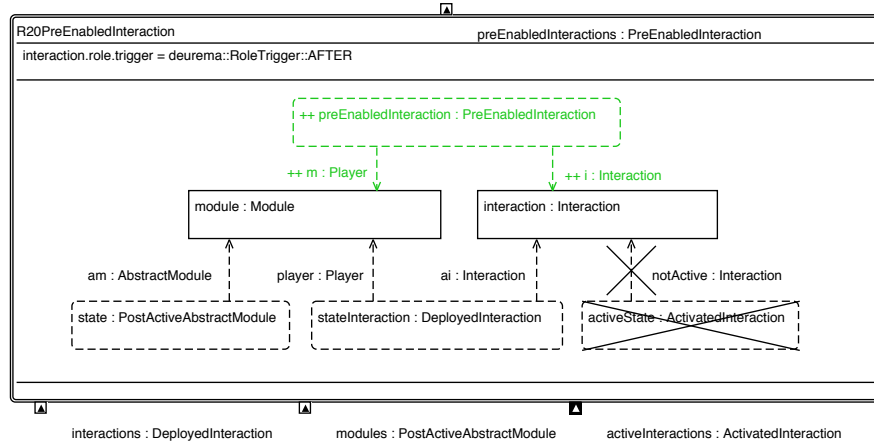


Figure D.21: The rule infers `preenabled` interactions, where the corresponding player module has a role trigger denoting the execution of the collaborative behavior after the execution of the local adaptation behavior. The player module must be in the `postactive` state.

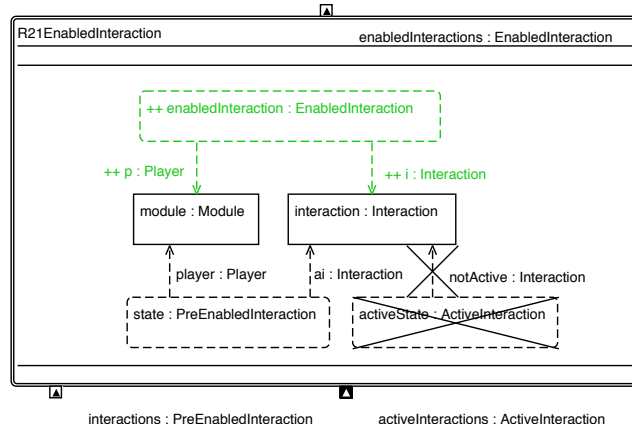


Figure D.22: A `preenabled` interactions becomes immediately `enabled` afterwards and thus, is ready for execution. The negative application condition ensures that each interaction does not become `active` twice.

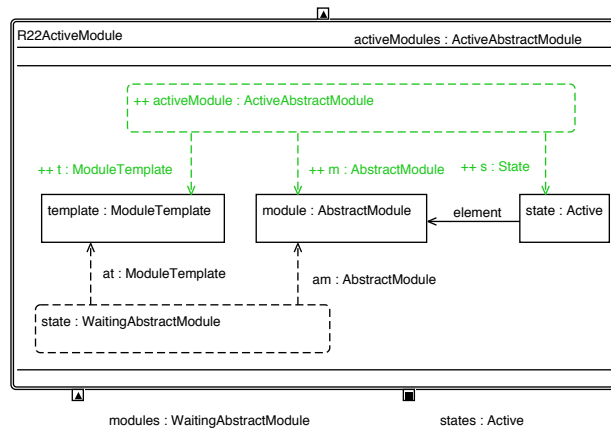


Figure D.23: This rule detects an `active` module. The Deurema interpreter adds the `active` state during the execution of the module.

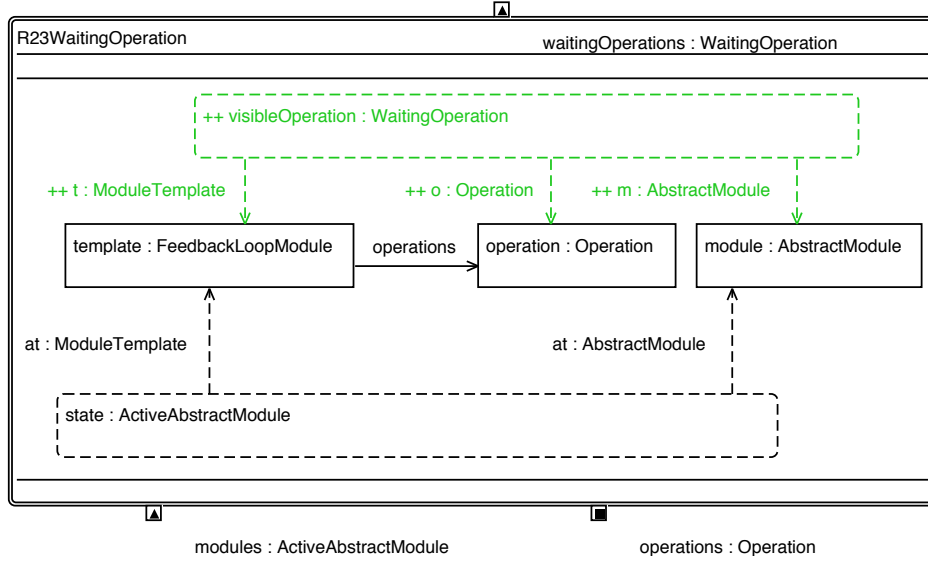


Figure D.24: This rule detects an active feedback loop module and corresponding template description and infers the contained operations, which are denoted as waiting.

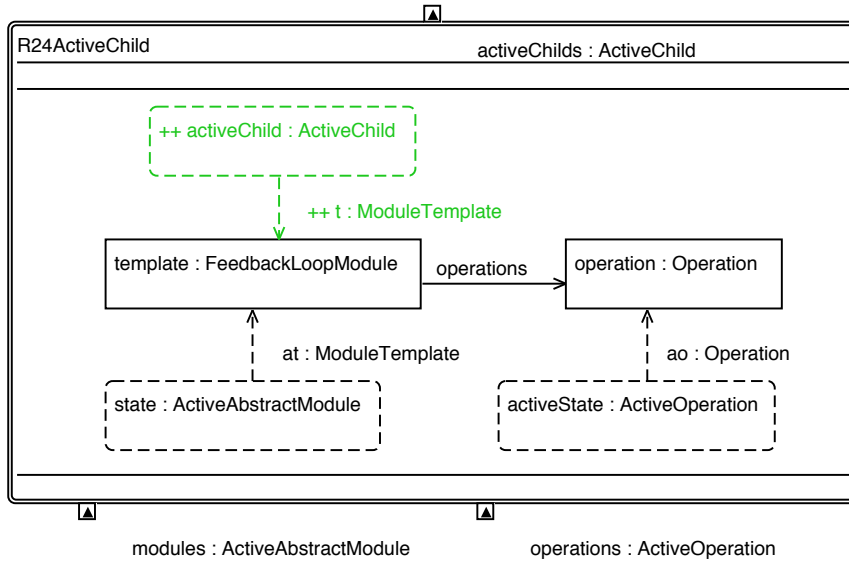


Figure D.25: This rule supervises a feedback loop module template and its contained operations. As long as at least one contained operation is active, the parent template gets an ActiveChild annotation.

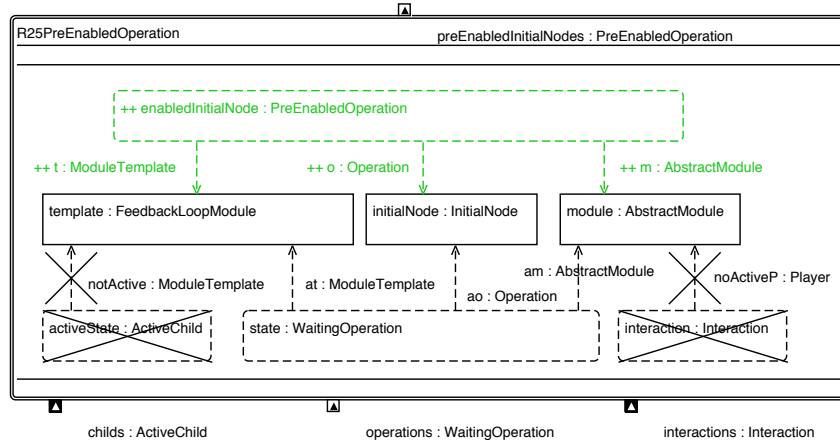


Figure D.26: The rule infers preenabled initial nodes of an active feedback loop module template. The initial node is the starting point for each feedback loop. The execution of the feedback loop can only start, if all collaboration activities are completed, which is ensured by the negative application condition.

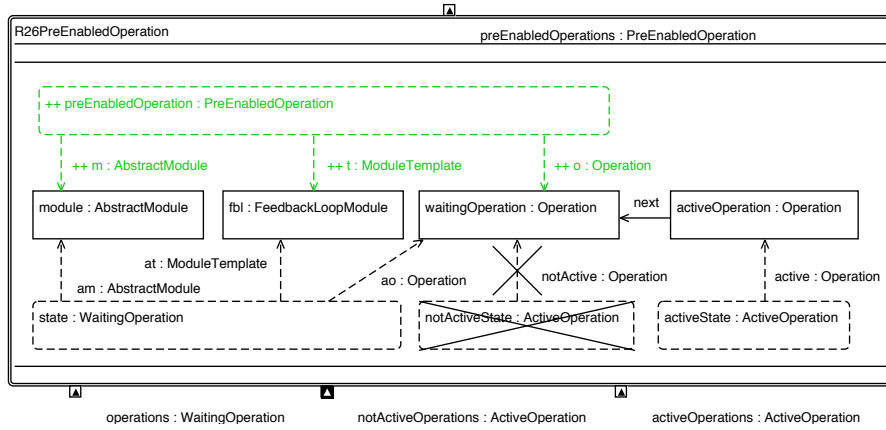


Figure D.27: The rule infers preenabled operations of an active feedback loop module template by following the control flow (next reference) of currently active operations.

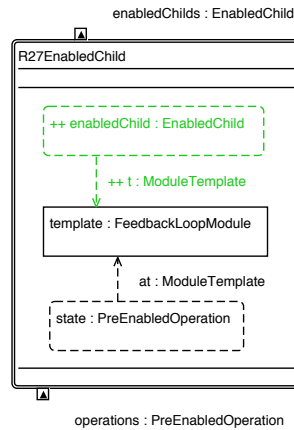


Figure D.28: This rule annotates a feedback loop template with an EnabledChild annotation as long as a contained operation is denoted as preenabled.

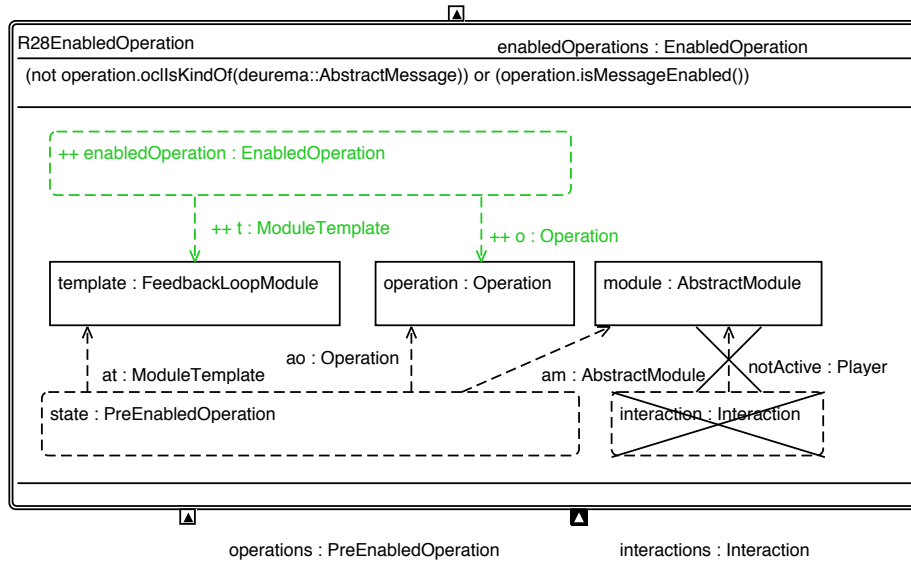


Figure D.29: The inference rule infers enabled operations within a feedback loop template, where the operation is in the preenabled state and does not participate in any interactions. Interaction messages, which are subclasses of the operation class, have an additional OCL constraint that evaluates, whether the message is enabled or not. Messages that perform as senders are always enabled. Receiver messages must wait until a sender emits a message.

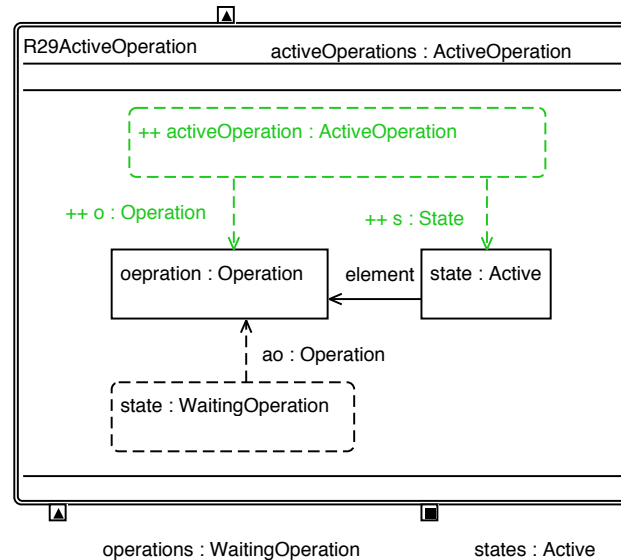


Figure D.30: This rule infers active operations, which are marked by the Deurema interpreter during the execution.

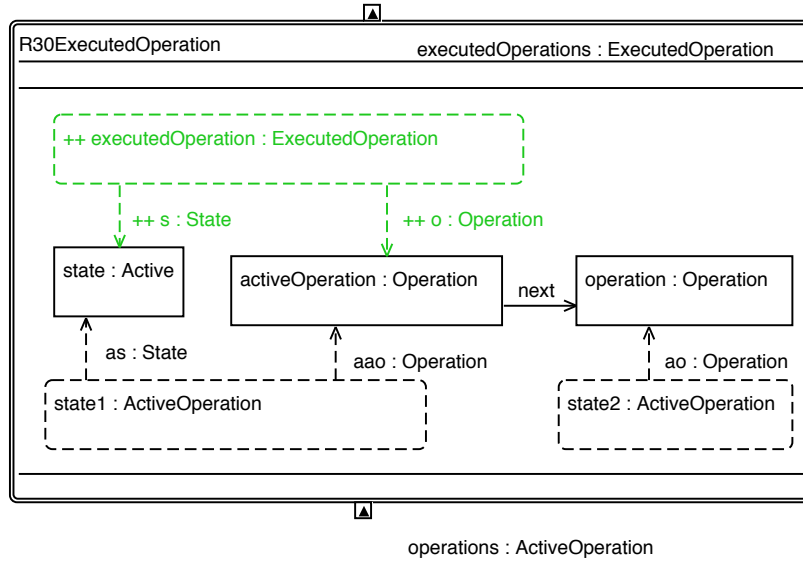


Figure D.31: The rule finds executed operations within a feedback loop module template by means of two active operations that are directly connected over the control flow (next reference). If the subsequently operation becomes active and thus, is executed by the Deurema interpreter, the previous operation is annotated as executed.

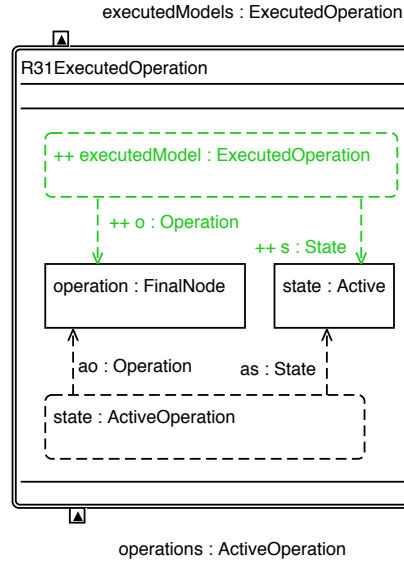


Figure D.32: The rule finds executed final nodes within a feedback loop module template. If the final node is marked as active by the Deurema interpreter, it has no further side effect and can be cleaned up.

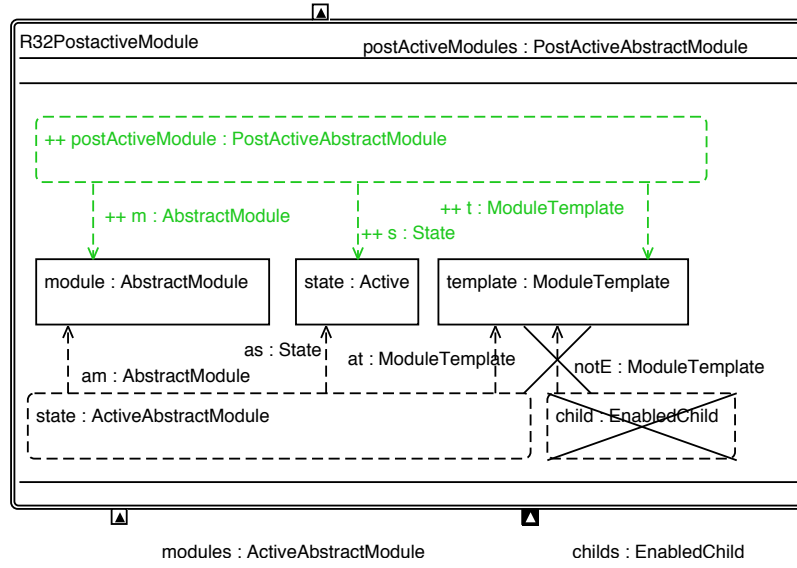


Figure D.33: This rule detects postactive module instances together with its corresponding template specification. A module instance is postactive, if no local adaptation steps are possible denoted by the EnabledChild annotation. This rule also works for interaction instances and the corresponding interaction part specification.

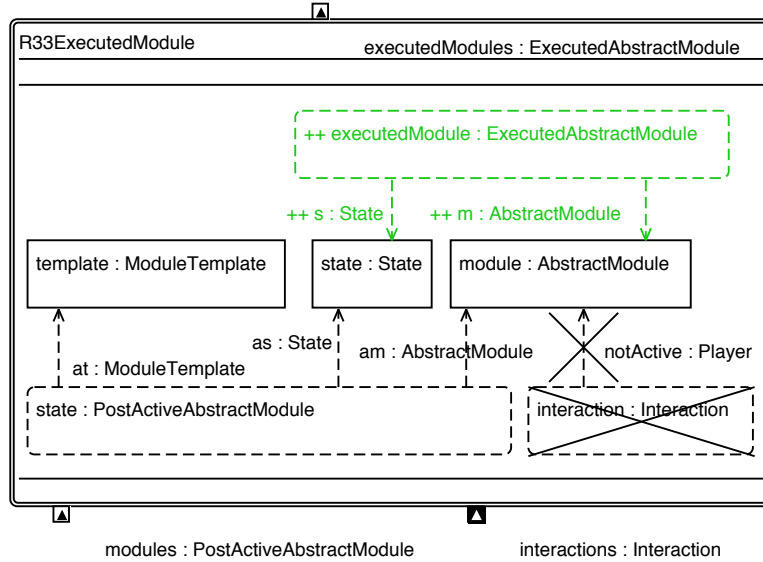


Figure D.34: If a module instance does not collaborate after it became postactive, it is denoted as executed. The negative application condition ensures that no interactions are actively executed.

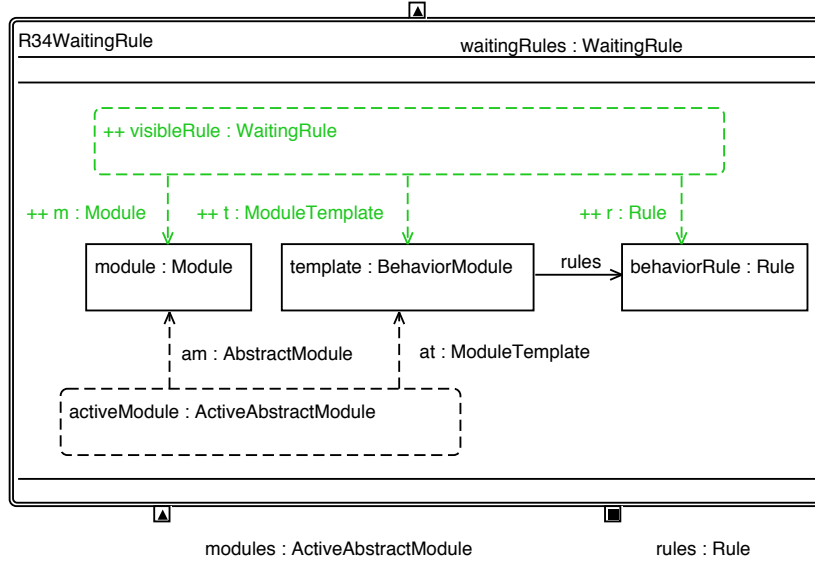


Figure D.35: A behavior rule becomes waiting and therefore, visible for the simulation environment, if the corresponding module instance changes into the active state.

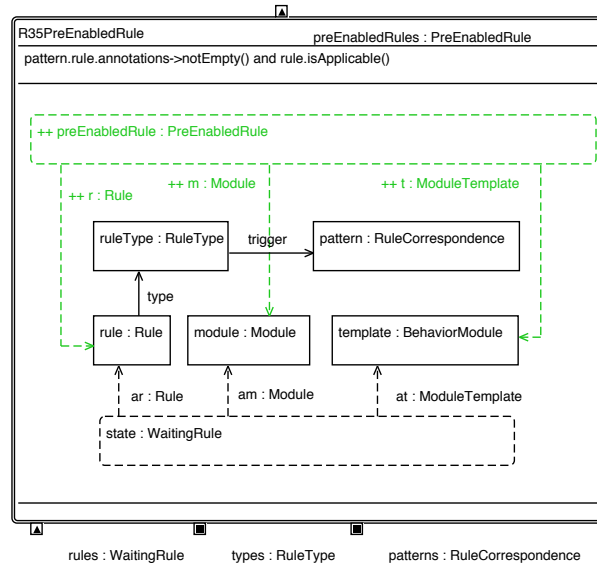


Figure D.36: This inference rule searches for a behavior rule, where the LHS has a match in the corresponding domain. Furthermore, all rule properties such as the period or probability must be fulfilled, which is ensured by the OCL constraint. This rule determines white box behavior rules.

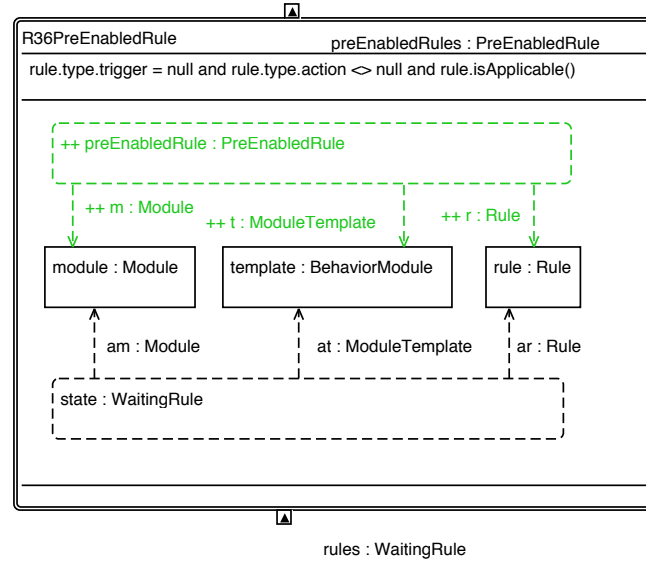


Figure D.37: This inference rule searches for a behavior rule, where the LHS has a match in the corresponding domain. Furthermore, all rule properties such as the period or probability must be fulfilled, which is ensured by the OCL constraint. This rule determines black box behavior rules.

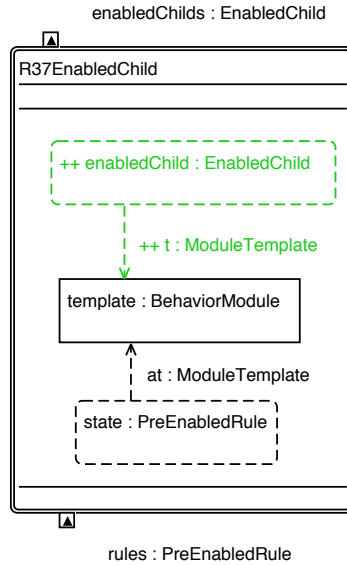


Figure D.38: This rule annotates a behavior module template with an EnabledChild annotation as long as a contained behavior rule is denoted as preenabled.

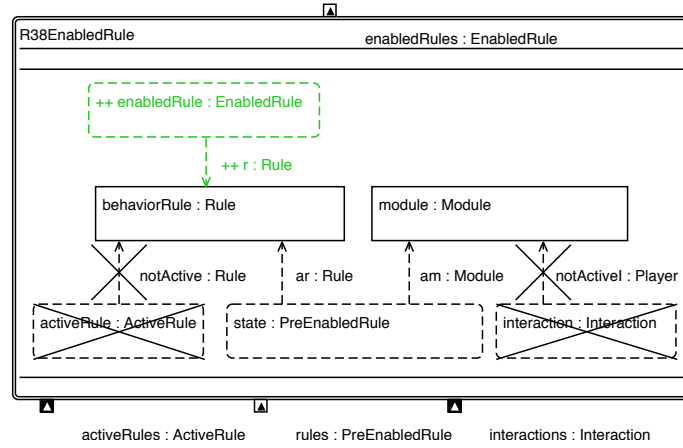


Figure D.39: A behavior rule is inferred as enabled by the inference engine, if it is not already active (not currently executed by the Deurema interpreter) and if it is not collaborating with another module. Both constraints are ensured by the negative application conditions.

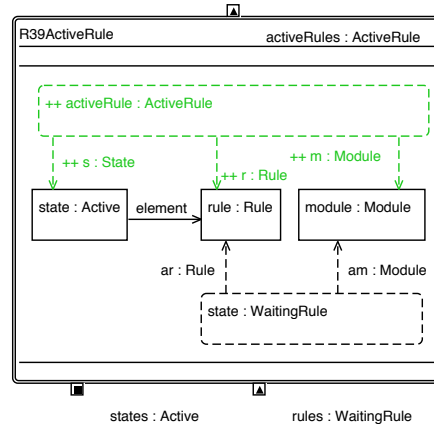


Figure D.40: A behavior rule is active, if the Deurema interpreter annotates a corresponding state marker, which is detected by the inference engine.

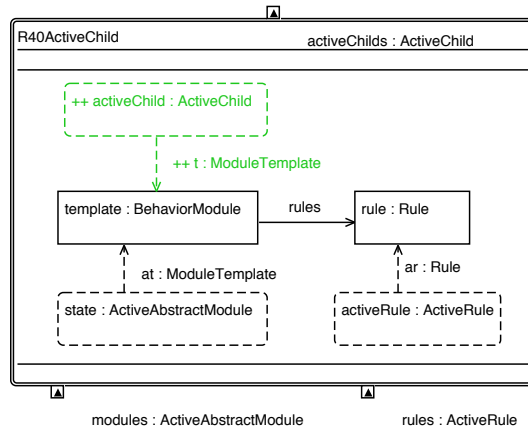


Figure D.41: This rule supervises a behavior rule module template and its contained rules. As long as at least one contained rule is active, the parent template gets an ActiveChild annotation by the inference engine.

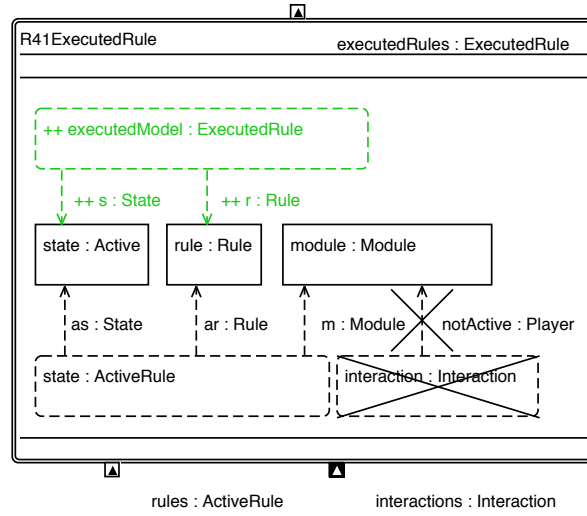


Figure D.42: A behavior rule is denoted as executed after the successfully execution by the Deurema interpreter and if no additional interactions are played.

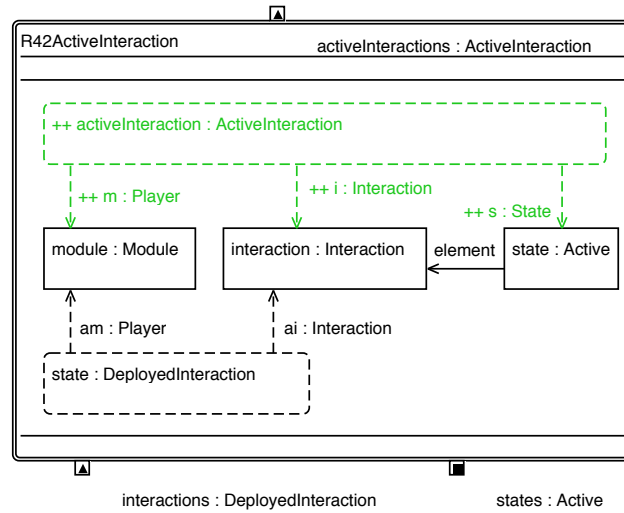


Figure D.43: The inference rule detects ongoing active interactions, which are currently executed by the Deurema interpreter. This rule annotates the interaction and corresponding player module instance.

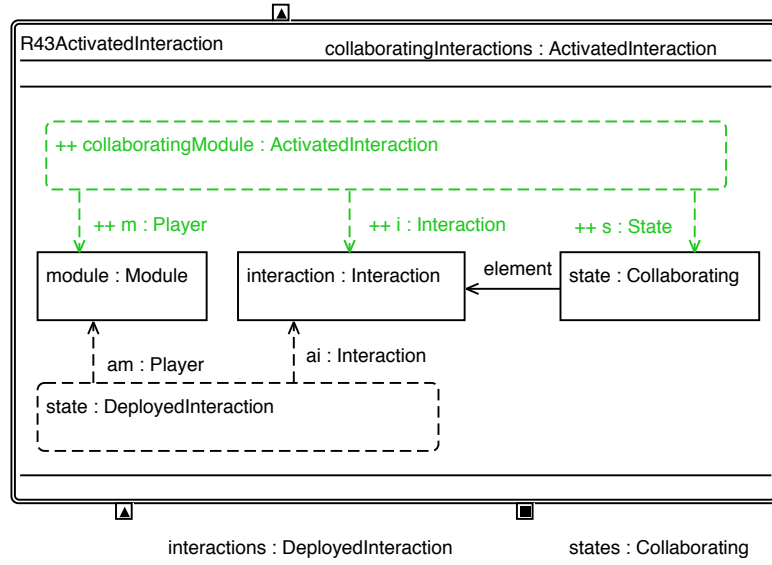


Figure D.44: This rule works similar to the rule in Figure D.43 and detects ongoing active interactions, which are currently executed by the Deurema interpreter. This rule annotates the interaction and corresponding player module instance. The difference is that the `ActivatedInteraction` annotation is used to determine postactive modules afterwards.

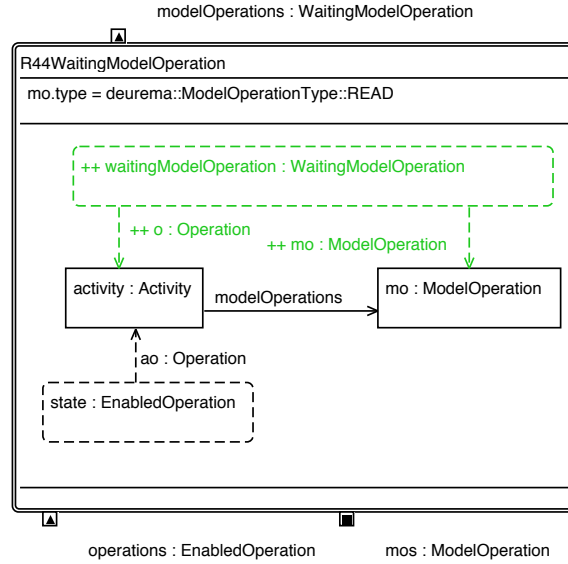


Figure D.45: The rule detects read model operations, where the contained model queries can be executed to retrieve the local amount of knowledge from the runtime model view, which is further hand over to the local adaptation effect.

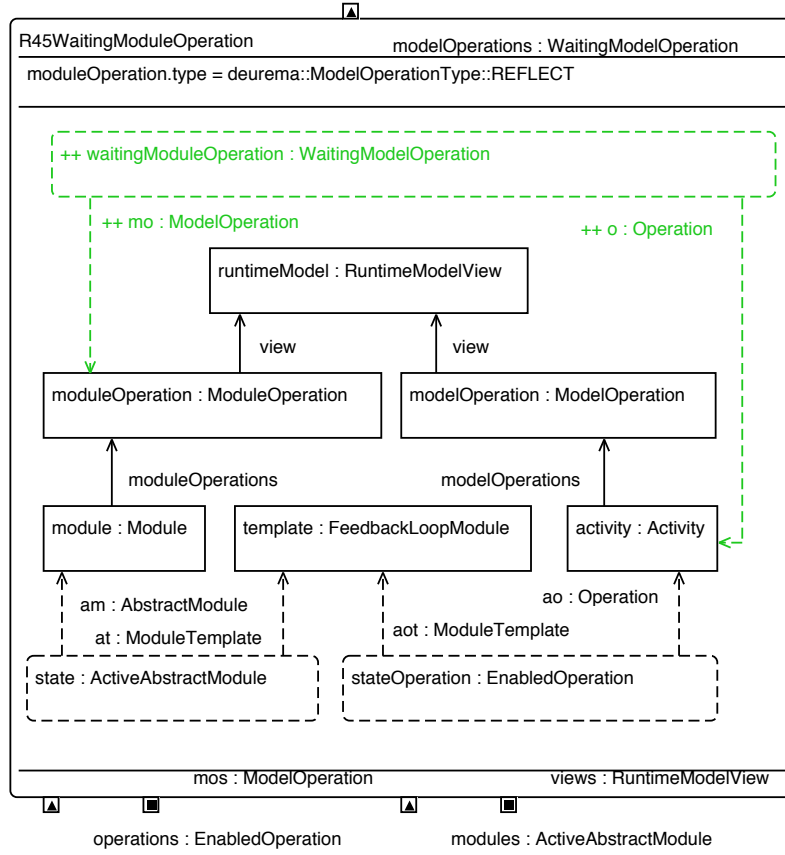


Figure D.46: The rule detects reflect model operations, which are ready for execution afterwards to update the local runtime model views within the annotated template description.

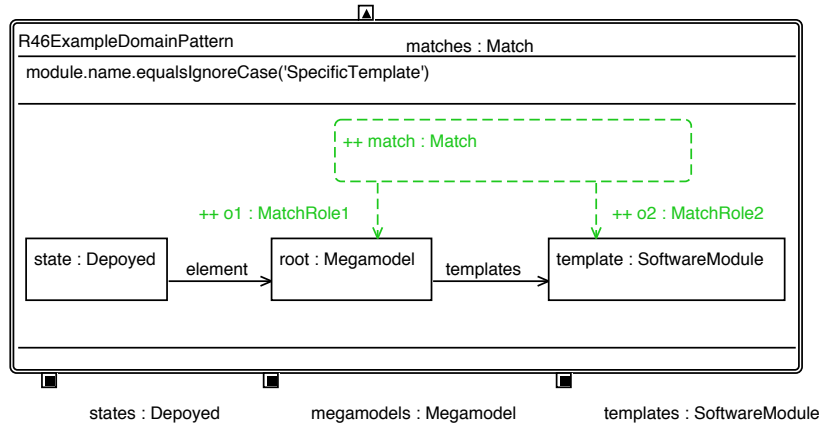


Figure D.47: This is an exemplary graph transformation behavior rule of a domain specific adaptation effect. This example rule searches for a template description in the Deurema megamodel, where the name of the template is *SpecificTemplate*. A side effect of this rule could be for example the deletion of the template specification from the overall Deurema model, which might cause further adaptation effects of the adaptive SoS architecture. In general, Deurema supports arbitrary, domain specific graph transformation behavior rules.

D.2. Simulation Metrics

Although it is not in the focus of this thesis to create an efficient simulation environment for Deurema models, this section pinpoints to measured performance metrics for two scenarios. In general, a simulation performance evaluation is very hard because different aspects have to be considered. Deurema supports the specification of adaptive behavior using a black-gray-white box mechanism. Except for the white box behavior specification, the Deurema execution environment does not know internals about the underlying, domain specific implementation. Furthermore, the simulation of the modeled SoS architecture includes the invocation of the domain specific behavior, which of course needs time for its execution. Separating execution time of domain specific behavior and time that is needed by the Deurema execution environment is hard to realize. There are the following key aspects that influence the execution time of a Deurema model simulation.

At first, the inference engine maintains the Deurema megamodel, which includes all Deurema models, the runtime models representing the available knowledge, and the dependencies between adaptation effects and their access to the runtime model views. As described in Chapter 7, the inference engine tracks changes in the megamodel and retrieves enabled Deurema elements for their execution. Therefore, the inference engine applies all simulation rules depicted in the Section D.1 above for each simulation step (cf. Figure 7.10 in Section 7.3). Applying the simulation rules comprises a time consuming pattern matching on the Deurema megamodel to reason about the current SoS situation. Retrieved matches and corresponding state annotations are directly marked in the Deurema model.

Second, the time of picking the next enabled Deurema model element for its execution depends on the scheduler implementation of the Deurema simulator. A random-based implementation or a FIFO algorithm are much faster than an earliest deadline first or complex domain specific scheduler implementation.

Third, the execution of a single Deurema element by the interpreter is very fast, but the invocation of domain specific adaptation effects can take a long time. For example, a domain specific planning activity may derive long term optimization strategies by applying complex and time consuming learning algorithms. In contrast, if the adaptation effect is defined as white box, the corresponding graph transformation rule must be applied on the beforehand retrieved knowledge base (cf. Section 5.3.2). Again, executing the graph transformation rule comprises a pattern matching part, which can be time consuming depending on the rule pattern and the amount of data in the base graph that has to be investigated. However, in both cases (black box and white box), the time for executing the adaptation effect remains problem specific.

Fourth, model operations are defined by means of model queries, which are again graph pattern. Executing these model queries includes a pattern matching part on the underlying knowledge base. Furthermore, the pattern size and the amount of model queries influence the execution time.

In summary, there are many dimensions that influence the execution time during a simulation of the adaptive SoS architecture. In the following, two scenarios are discussed, which are used to measure execution metrics of a Deurema model simulation. The shown metrics should be seen as starting point. Due to the goals of this thesis, implementing a scalable simulation framework is not in the focus of this work and belongs to future work. All performance metrics from the following Deurema model simulations are measured on an Intel Xeon E5-2630 processor with about 384 GB main memory and 2,3 GHz. The operating system is a Debian

8 (Jessie) and the Deurema execution environment is a single core Java program developed with the Eclipse Modeling Framework with the JDK 1.7.

Smart Home

Figure D.48 sketches the simulation scenario of a variable number of smart home instances. Each smart home is modeled as black box software module. The implementation of the module refers to a Java method, which is invoked by the Deurema interpreter. To decrease the execution time of the Java method, which is domain specific behavior and thus, less important for investigating the execution characteristics of the Deurema simulation environment, the implementation is very simple and tracks its own execution by means of an execution log. Obviously, the drawback of this simple implementation is that no meaningful domain specific behavior is executed. However, each smart home software module triggers one feedback loop module at a higher layer as shown in Figure D.48. Each feedback loop comprises a full MAPE cycle, whereas the adaptation activities are again modeled as black boxes that trace their invocation by means of an execution log. In this scenario, each smart home module instance has a corresponding feedback loop module instance. All feedback loops refer to the same template description, which is the full MAPE cycle. For the performance evaluation there are two different scenarios. At first, the number of smart home and feedback loop pairs is constant and the number of simulation steps is increased. Second, the simulation steps are fix and the number of pairs is increased. Therefore, on the one hand, this scenario shows basic metrics for multiple independent module instances by means of smart home and feedback loop pairs. On the other hand, there is the trigger dependency between a smart home and its corresponding feedback loop module. Furthermore, the intra-loop coordination of the feedback loop must be considered during a simulation.

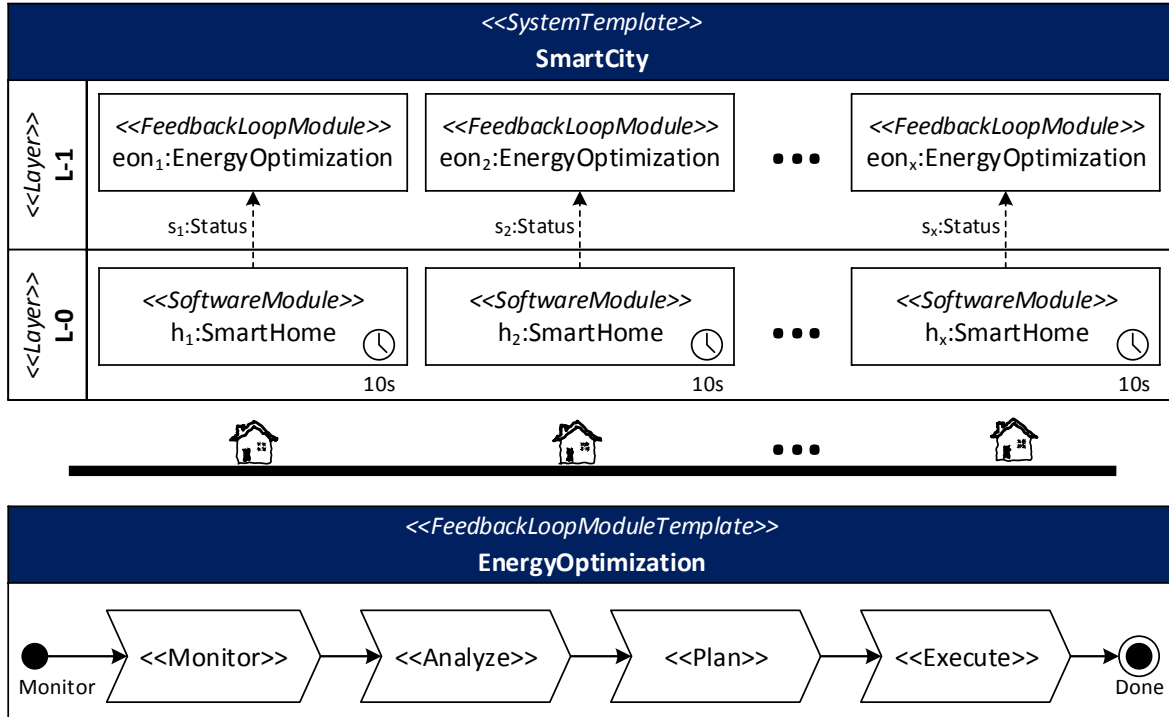


Figure D.48: Smart home simulation scenario

Smart Car

A more complex simulation scenario is sketched in Figure D.49. This scenario comprises multiple collaboration instances and thus extends the smart home scenario by using the Deurema collaboration concept. In this scenario, there are always two cars with two corresponding feedback loops, where one car plays a leader role and the other car realizes a follower role within a platoon collaboration. The interaction is integrated into the feedback loop by means of a heart beat protocol as comprehensively discussed in Section 5.5. As in the smart home scenario, all adaptation effects are modeled as black boxes that trace their execution. This scenario is investigated in two variants. At first, the number of platoon collaboration is fix and the number of simulation steps is increased. Second, the simulation steps remain fix and the number of collaboration instances is increased. Thereby, each collaboration comprises exactly one leader and follower car as shown in Figure D.49.

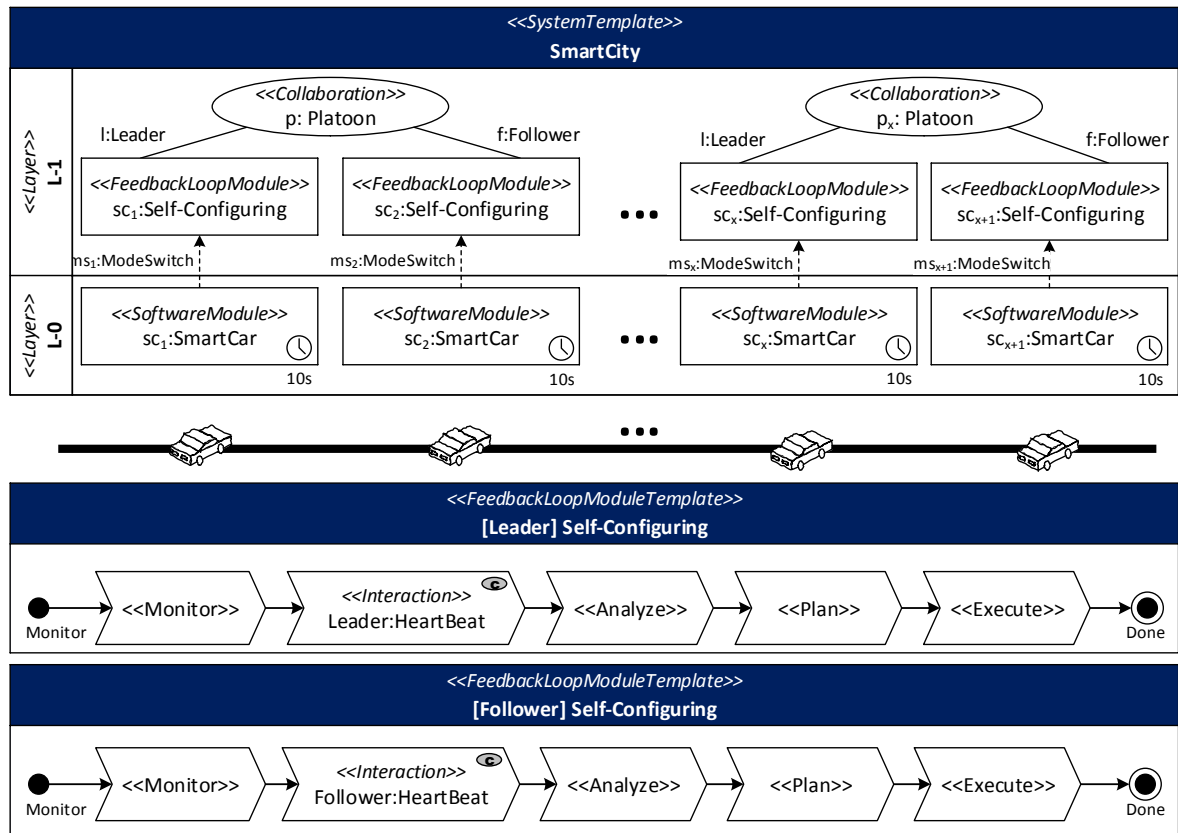


Figure D.49: Smart car simulation scenario

Increasing Simulation Steps

The performance metrics of the smart home and smart car scenario for a constant number of module and collaboration elements, but an increasing number of simulation steps is shown in Table D.1. A visualization is depicted in Figure D.50. The number of pair respectively collaboration instances is ten. The simulator uses a random-based scheduling algorithm. The needed simulation time increases with the number of simulation steps, whereas the smart home scenario is faster than the smart car scenario. The difference in the execution times between both is because of the different complexity of the setting. For the smart car scenario, the Deurema execution environment must additionally consider collaborations between modules.

Table D.1: Performance metrics for an increasing number of simulation steps for the smart home and smart car scenario. The time is measured in milliseconds.

Simulation Steps	Execution Time (ms)
Smart Home	
1	40
10	1.350
50	7.947
100	18.850
500	176.110
1.000	486.620
Smart Car	
1	66
10	2.756
50	15.121
100	67.894
500	534.283
1.000	1.305.783

Number of instances: 10; Initial build of the megamodel for smart home scenario: 254 ms; Initial build smart car: 657 ms; The mean execution time for one simulation steps over all timing measurements is about 415 ms for the smart home scenario and 1160 ms for the smart car scenario.

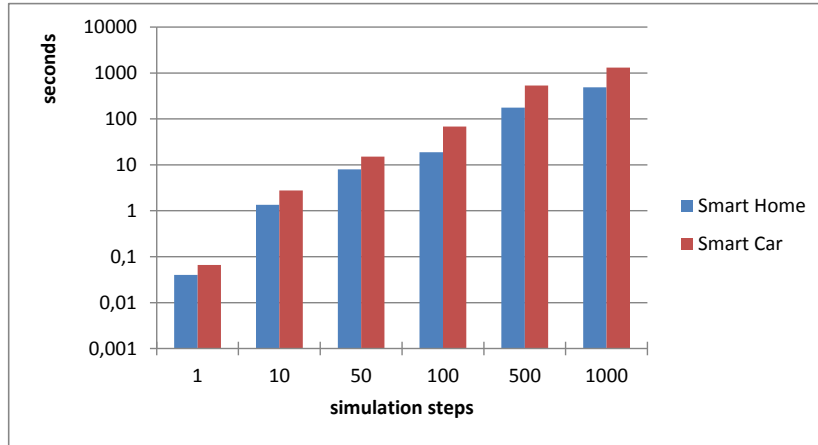


Figure D.50: Simulation metric for increasing simulation steps. Measured time is in seconds. Each simulation run was twenty times repeated and the mean value is plotted in the diagram. The corresponding values are depicted in Table D.1.

Increasing Simulation Instances

The performance metrics of the smart home and smart car scenario for a constant number of simulation steps, but increasing number of instances is shown in Table D.2. A visualization is depicted in Figure D.51. The number of simulation steps is one hundred. The simulator uses a random-based scheduling algorithm. The needed simulation time increases with the number of instances, whereas the smart home scenario is faster than the smart car scenario. The increased simulation time is caused by the increasing complexity of the Deurema model and thus, the increasing time of the graph pattern matching via the application of simulation rules and corresponding state annotations.

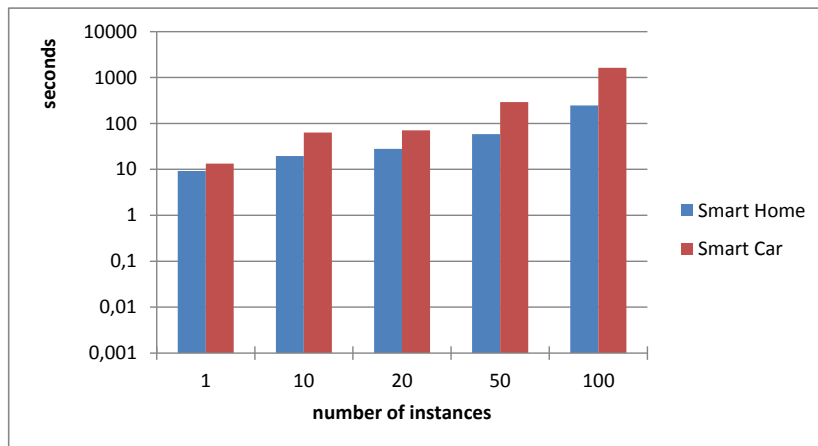


Figure D.51: Simulation metric for scaling simulation instances. Measured time is in seconds. Each simulation run was twenty times repeated and the mean value is plotted in the diagram. The corresponding values are depicted in Table D.2.

Table D.2: Performance metrics for an increasing number of simulation instances for the smart home and smart car scenario. The time is measured in milliseconds.

Pairwise Instances	Modules	Collaborations	Execution Time (ms)	Initial Build (ms)
Smart Home				
1	2	0	9.264	127
10	20	0	19.460	254
20	40	0	28.080	416
50	100	0	58.482	972
100	200	0	245.853	2.014
Smart Car				
1	4	1	13291	175
10	40	10	63.114	657
20	80	20	70.548	945
50	200	50	293.400	2.663
100	400	100	1.627.111	6.692

Number of simulation steps: 100; The metric *instance* refers to the pair of smart home module and corresponding feedback loop respectively to a complete leader-follower setting of a platoon collaboration; The initial build refers to the first application of the simulation rules and corresponding annotation of state information within the Deurema model.

In summary, this section shows some basic execution metrics of two scenarios for the Deurema execution framework. Complex investigations that pinpoint to the execution metrics of different parts in the Deurema execution framework such as the interpreter, simulator, inference engine, model queries, or adaptation effects are not in the focus of this thesis and are a good starting point for future work.

List of Figures

1.1. Overview of goals	3
2.1. External adaptation and MAPE-K feedback loop	14
2.2. Hierarchy of self-* properties	16
2.3. System type evolution	20
2.4. Original and its model	22
2.5. Metamodel with corresponding model	23
2.6. Megamodel containing models and relationships	26
2.7. Graph transformation rule	28
2.8. Eurema Feedback Loop Diagram	30
2.9. Eurema Layer Diagram	31
2.10. Collaboration terms	33
2.11. Smart city running example	36
4.1. Overview	58
4.2. Smart city running example: Deurema adaptive SoS modeling	59
4.3. Modeling the adaptation logic in Deurema	61
4.4. Deurema knowledge as runtime models	63
4.5. Deurema system interactions via collaborations	64
4.6. Deurema adaptive SoS architecture	65
4.7. Deurema analysis	67
4.8. Simulating Deurema models	68
4.9. Realization of Deurema models	69
5.1. Smart city running example: Deurema system modeling	72
5.2. Deurema metamodel of core concepts	73
5.3. Deurema system template example	75
5.4. Smart city running example: Deurema runtime models	76
5.5. Runtime reflection model types	77
5.6. Runtime system models	78
5.7. Runtime context models	79
5.8. Runtime adaptation model types	79
5.9. Change and evaluation model	80
5.10. Variability and modification model	81
5.11. Runtime causal connection model types	83
5.12. MAPE feedback loop with runtime models	84
5.13. Sketch of a self-configuring smart car with runtime models	86
5.14. Deurema runtime model	88
5.15. Deurema runtime model summary	89
5.16. Smart city running example: Deurema modules	90
5.17. Deurema module templates	91

5.18. Variables and runtime model views in module templates	93
5.19. Smart city running example: Deurema feedback loop template	94
5.20. Deurema FLD	95
5.21. Feedback loop activities as behavioral models	97
5.22. Deurema FLD runtime models	100
5.23. Deurema Self-Configuring feedback loop template	103
5.24. Smart city running example: Deurema software module template	104
5.25. Deurema Software Module Diagram	104
5.26. Deurema SMD example	105
5.27. Smart city running example: Deurema application module template	107
5.28. Deurema Application Component Diagram	107
5.29. Deurema ACD runnables and ports	109
5.30. Deurema ACD runtime models	111
5.31. Deurema ACD port and task combinations	112
5.32. ACD example	114
5.33. Smart city running example: Deurema behavior module template	115
5.34. Deurema Behavior Rule Diagram	117
5.35. Behavior rule property combinations	119
5.36. BRD rules and runtime models	120
5.37. Derived runtime model operation types	121
5.38. BRD rule example	122
5.39. Smart city running example: Deurema module trigger dependencies	123
5.40. Deurema module trigger	124
5.41. Deurema LD example	126
5.42. Smart city running example: Deurema collaborations	128
5.43. Deurema collaboration modeling dimensions	128
5.44. Deurema collaboration structure	129
5.45. Collaboration structure example	130
5.46. Collaboration knowledge specification	131
5.47. Deurema collaboration choreography	132
5.48. Choreography interaction example	133
5.49. Deurema interaction templates	134
5.50. Interaction template example using a message	135
5.51. Interaction template example using a service	136
5.52. Interaction template example using a model message	137
5.53. Delayed synchronization example	138
5.54. Message property example	139
5.55. Interaction role interface	140
5.56. Collaboration role integration	141
5.57. Deurema interaction trigger	142
5.58. Deurema collaborative elements	143
5.59. Collaboration deployment	144
5.60. Collaboration role delegation	145
5.61. Smart city running example: Deurema reflection and adaptation	146
5.62. Deurema reflection metamodel	147
5.63. Module reflection example	148
5.64. System reflection example	148

5.65. Collaboration reflection example	149
5.66. Deurema variability model	150
5.67. Reconfiguration example	151
5.68. Reconfiguration space	152
5.69. Resolved variable configuration	153
5.70. Deurema adaptation	154
5.71. Module meta-adaptation example	155
5.72. System meta-adaptation example	156
5.73. Collaboration meta-adaptation example	157
5.74. Smart car running example modeled in Deurema	159
5.75. Smart city running example modeled in Deurema	162
6.1. Smart city running example: Deurema analysis	167
6.2. Deurema analysis overview	168
6.3. Causal dependencies in a FLD	170
6.4. Causal dependency paths in FLD	171
6.5. Causal dependencies in BRD	173
6.6. Module trigger dependency	174
6.7. Collaboration trigger dependencies	175
6.8. Interaction message dependency	176
6.9. Analysis of the knowledge purpose	177
6.10. Analysis of reflective knowledge dependencies	178
6.11. Adaptation purpose in FLD	179
6.12. Adaptation purpose in BRD	180
6.13. Exit on analyze pattern	182
6.14. Combining interaction message types and message flow	182
6.15. Analysis rules for combining knowledge and causality	185
6.16. Analysis of hierarchical and layered control	187
6.17. Layered trigger design flaw	188
6.18. Analysis of causality knowledge design flaw	189
6.19. Analysis of architectural design flaws	190
7.1. Smart city running example: Deurema simulation	194
7.2. Deurema state model for modules and interactions	195
7.3. Simulation rule for detecting waiting modules	197
7.4. Simulation rule detecting a preenabled interaction	198
7.5. Deurema state model for contained module elements	199
7.6. Simulation rule for detecting a preenabled operation	200
7.7. Deurema controlled elements	202
7.8. Deurema behavior models	204
7.9. Deurema interaction	205
7.10. Deurema simulation workflow	206
7.11. Deurema simulation run example	209
7.12. Smart city running example: Deurema runtime analysis	211
8.1. Smart city running example: Deurema realization	216
8.2. Realized Deurema modules	217
8.3. Layered AUTOSAR architecture	218

8.4. AUTOSAR components, ports, and interfaces	218
8.5. Deurema modules as AUTOSAR compositions	221
8.6. Deurema software module as AUTOSAR component	223
8.7. AutonomousDriving and AdaptiveLightControl Deurema module templates	224
8.8. AutonomousDriving as AUTOSAR component architecture	224
8.9. AdaptiveLightControl as AUTOSAR component architecture	225
8.10. Embedded control model	226
8.11. Development stages in the embedded domain	228
8.12. Verification steps during the adaptive SoS modeling with Deurema	229
8.13. SystemDesk software tool	232
9.1. Smart city running example: Deurema application	233
9.2. Traffic Monitoring System modeled with Deurema	235
9.3. Smart home modeled with Deurema	239
9.4. Mapping DEECoo concepts to Deurema	243
11.1. Thesis goals and modeling language requirements	271
A.1. Deurema system and template metamodel	291
A.2. Deurema module template metamodel	292
A.3. Deurema feedback loop template metamodel	293
A.4. Deurema behavior module template metamodel	294
A.5. Deurema application module template metamodel	295
A.6. Deurema collaboration metamodel	296
A.7. Deurema message concept	297
A.8. Deurema variable types	298
A.9. Deurema view delegation concept	299
B.1. Message properties	301
B.2. Interaction scenarios with multiple senders and receivers	302
C.1. Inference engine rule example	306
C.2. Overview of annotation types for causal dependencies	307
C.3. Overview of annotation types for knowledge dependencies	308
C.4. Overview of annotation types for the adaptation purpose	308
C.5. Overview of annotation types for combined dependencies and patterns	309
C.6. R01CausalDependency	310
C.7. R02ConstraintCausalDependency	310
C.8. R03CausalDependency	310
C.9. R04ClosureDependency	311
C.10. R05ClosureDependency	311
C.11. R06Path	311
C.12. R07DestructionPath	312
C.13. R08TriggerDependency	312
C.14. R09ClosureTriggerDependency	312
C.15. R10LayeredTriggerDependency	313
C.16. R11DeferredCollaborationTrigger	313
C.17. R12InAdvanceCollaborationTrigger	313
C.18. R13InBetweenCollaborationTrigger	314

C.19.R14DeferredInterleaving	314
C.20.R15InAdvanceInterleaving	314
C.21.R16InterleavingCollaborationTrigger	315
C.22.R17CommunicationFlow	315
C.23.R18UniDirectionInteractionMessageDependency	316
C.24.R19BiDirectionInteractionMessageDependency	316
C.25.R20BiDirectionalCollaboration	317
C.26.R21UniDirectionalCollaboration	317
C.27.R22SelfRepresentative	317
C.28.R23ContextRepresentative	318
C.29.R24RequirementRepresentative	318
C.30.R25ChangeRepresentative	318
C.31.R26SensorRepresentative	319
C.32.R27EffectorRepresentative	319
C.33.R28Relective	319
C.34.R29Changeable	320
C.35.R30CausalConnected	320
C.36.R31KnowledgeAccessRead	320
C.37.R32KnowledgeAccessRead	321
C.38.R33KnowledgeAccessRead	321
C.39.R34KnowledgeAccessWrite	321
C.40.R35KnowledgeAccessAnnotate	322
C.41.R36KnowledgeAccessCreate	322
C.42.R37KnowledgeAccessDestroy	322
C.43.R38KnowledgeAccessModify	323
C.44.R39ReflectDependency	323
C.45.R40LayeredReflectDependency	323
C.46.R41AffectDependency	324
C.47.R42LayeredAffectDependency	324
C.48.R43AdaptationPurposeMonitor	325
C.49.R44AdaptationPurposeAnalyze	325
C.50.R45AdaptationPurposePlan	325
C.51.R46AdaptationPurposeExecute	326
C.52.R47AdaptationPurposeMAPE	326
C.53.R48AdaptationPurposeCollector	326
C.54.R49AdaptationPurposeCollector	327
C.55.R50AdaptationPurposeAnalyzer	327
C.56.R51AdaptationPurposeAnalyzer	327
C.57.R52AdaptationPurposeAnalyzer	328
C.58.R53AdaptationPurposeAnalyzer	328
C.59.R54AdaptationPurposePlanner	328
C.60.R55AdaptationPurposePlanner	329
C.61.R56AdaptationPurposeSensor	329
C.62.R57AdaptationPurposeSWC	329
C.63.R58AdaptationPurposeActuator	330
C.64.R59AdaptationPurposeSWA	330
C.65.R60AdaptationPurposeCompute	330

C.66.R61AdaptationPurposeSenseEffect	331
C.67.R62AdaptationPurposeSensingTask	331
C.68.R63AdaptationPurposeEffectingTask	331
C.69.R64AdaptationPurposeComputingTask	332
C.70.R65AdaptationPurposeComponentCentricTask	332
C.71.R66AccessPatternMonitorModify	332
C.72.R67AccessPatternExecuteModifyViolation	333
C.73.R68AccessPatternAnalyzeRead	333
C.74.R69AccessPatternPlanReadViolation	333
C.75.R70KnowledgeAware	334
C.76.R71KnowledgeModification	334
C.77.R72KnowledgeDerivation	334
C.78.R73KnowledgeTransition	335
C.79.R74KnowledgePropagation	335
C.80.R75KnowledgeSink	335
C.81.R76KnowledgeSource	336
C.82.R77MAPEFeedbackLoop	336
C.83.R78AMPEAntiPattern	337
C.84.R79AntiPatternReflect	337
C.85.R80AntiPatternAffect	338
C.86.R81ExitAfterAnalyzePattern	338
C.87.R82MissingCommunication	339
C.88.R83AntiPatternCollaboration	339
D.1. Deurema simulation rules dependency graph	341
D.2. R01DeployedSystem	342
D.3. R02DeployedSubsystem	342
D.4. R03DeployedModule	342
D.5. R04DeployedInteraction	343
D.6. R05DeployedInteraction	343
D.7. R06WaitingModule	344
D.8. R07WaitingEventTrigger	344
D.9. R08WaitingClock	345
D.10.R09EnabledEventTrigger	345
D.11.R10EnabledTimedTriggerWithEventTrigger	346
D.12.R11EnabledTimedTriggerWithoutEventTrigger	346
D.13.R12WaitingModuleTriggeredByEvent	346
D.14.R13WaitingModuleTriggeredByTime	347
D.15.R14WaitingModuleTriggeredByEventAndTime	347
D.16.R15PreenabledModule	347
D.17.R16EnabledModule	348
D.18.R17PreEnabledInteraction	348
D.19.R18PreEnabledInteraction	349
D.20.R19PreEnabledInteraction	349
D.21.R20PreEnabledInteraction	350
D.22.R21EnabledInteraction	350
D.23.R22ActiveModule	350

D.24.R23WaitingOperation	351
D.25.R24ActiveChild	351
D.26.R25PreEnabledOperation	352
D.27.R26PreEnabledOperation	352
D.28.R27EnabledChild	352
D.29.R28EnabledOperation	353
D.30.R29ActiveOperation	353
D.31.R30ExecutedOperation	354
D.32.R31ExecutedOperation	354
D.33.R32PostactiveModule	355
D.34.R33ExecutedModule	355
D.35.R34WaitingRule	356
D.36.R35PreEnabledRule	356
D.37.R36PreEnabledRule	357
D.38.R37EnabledChild	357
D.39.R38EnabledRule	358
D.40.R39ActiveRule	358
D.41.R40ActiveChild	358
D.42.R41ExecutedRule	359
D.43.R42ActiveInteraction	359
D.44.R43ActivatedInteraction	360
D.45.R44WaitingModelOperation	360
D.46.R45WaitingModuleOperation	361
D.47.R46ExampleDomainPattern	361
D.48.Smart home simulation scenario	363
D.49.Smart car simulation scenario	364
D.50.Metrics for simulation steps	366
D.51.Metrics for scaling simulation instances	366

List of Tables

2.1. Categories of communication	34
5.1. Change model examples for parameter and dynamic adaptation	82
5.2. Trigger and action combination for behavior models	98
5.3. Domain knowledge and domain functionality dimensions	101
6.1. Access patterns for combining knowledge and adaptation purposes	183
10.1. Requirements and general purpose modeling languages	246
10.2. Requirements and domain specific modeling approaches	251
10.3. Requirements and formal modeling approaches	257
10.4. Requirements and frameworks	260
10.5. Requirements and own research group experiences	265
B.1. Message properties execution scenarios	303
D.1. Performance metrics for increasing simulation steps	365
D.2. Performance metrics for increasing simulation instances	367

List of Abbreviations

ABS	Antilock Braking System.....	36
ACD	Application Component Diagram	108
ADL	Architecture Description Language.....	247
ALC	Adaptive Light Control	36
ASR	Traction Control System.....	36
AUTOSAR	Automotive Open System Architecture.....	217
BPMN	Business Process Model and Notation.....	23
BRD	Behavior Rule Diagram	116
CPS	Cyber-Physical System.....	16
DEECo	Distributed Emergent Ensembles of Components	241
Deurema	Distributed Eurema with Collaborations	57
DSL	Domain Specific (Modeling) Language	253
EBCS	Ensemble-Based Component System	241
ECU	Electronic Control Unit	27
ESP	Electronic Stability Program	36
Eurema	Executable Runtime Megamodels	29
FLD	Feedback Loop Diagram	29
FoS	Federation of Systems.....	18
fUML	Foundational UML	249
HiL	hardware-in-the-loop.....	227
IoT	Internet of Things	18
KAOS	Knowledge Acquisition in Automated Specification	24
LD	Layer Diagram.....	29
LHS	left-hand-side	27
LTL	Linear Temporal Logic.....	80
MAPE	Monitor, Analyze, Plan, and Execute.....	14
MAPE-K	MAPE activities share a common knowledge base	14
MART	Models@runtime.....	23
MDA	Model-Driven Architecture.....	21
MDE	Model-Driven Engineering	20
MiL	model-in-the-loop	227

MT	model test	227
mUML	Mechatronic UML	35
NCPS	Networked Cyber-Physical System	17
NES	Networked Embedded System	16
OCL	Object Constraint Language	80
OMG	Object Management Group	21
PIM	Platform Independent Model	21
PSM	Platform Specific Model	21
RHS	right-hand-side	27
SAS	Self-Adaptive System	13
SiL	software-in-the-loop	227
SMD	Software Module Diagram	105
SoaML	Service-oriented architecture Modeling Language	35
SoS	System of Systems	18
SPLE	Software Product Line Engineering	15
SysML	Systems Modeling Language	245
ULSS	Ultra-Large-Scale Systems	18
UML	Unified Modeling Language	21
QoS	Quality of Service	80

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit mit dem Titel „Modeling Collaborations in Adaptive Systems of Systems“ selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit wurde zuvor in keiner anderen Hochschule und in keinem anderen Studiengang als Prüfungsleistung eingereicht.

Potsdam, 11. Oktober 2016

Unterschrift:

Sebastian Wätzoldt

Statutory Declaration

I declare that I have authored this thesis entitled "Modeling Collaborations in Adaptive Systems of Systems" independently, that I have not used other than the referenced sources and resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. This thesis was neither submitted to another university nor to another course of study.

Potsdam, October 11, 2016

Signature:

Sebastian Wätzoldt
