Modeling Collaborations in Self-Adaptive Systems of Systems: Terms, Characteristics, Requirements, and Scenarios

Sebastian Wätzoldt, Holger Giese

Technische Berichte Nr. 96

des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam | 96

Sebastian Wätzoldt | Holger Giese

Modeling Collaborations in Self-Adaptive Systems of Systems

Terms, Characteristics, Requirements, and Scenarios

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2015
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://publishup.uni-potsdam.de/opus4-ubp/frontdoor/index/index/docId/7303
URN urn:nbn:de:kobv:517-opus4-73036
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-73036

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-324-4

An increasing demand on functionality and flexibility leads to an integration of beforehand isolated system solutions building a so-called *System of Systems* (SoS). Furthermore, the overall SoS should be adaptive to react on changing requirements and environmental conditions. Due SoS are composed of different independent systems that may join or leave the overall SoS at arbitrary point in times, the SoS structure varies during the systems lifetime and the overall SoS behavior emerges from the capabilities of the contained subsystems. In such complex system ensembles new demands of understanding the interaction among subsystems, the coupling of shared system knowledge and the influence of local adaptation strategies to the overall resulting system behavior arise. In this report, we formulate research questions with the focus of modeling interactions between system parts inside a SoS. Furthermore, we define our notion of important system types and terms by retrieving the current state of the art from literature. Having a common understanding of SoS, we discuss a set of typical SoS characteristics and derive general requirements for a collaboration modeling language. Additionally, we retrieve a broad spectrum of real scenarios and frameworks from literature and discuss how these scenarios cope with different characteristics of SoS. Finally, we discuss the state of the art for existing modeling languages that cope with collaborations for different system types such as SoS.

Contents

List of Figures

List of Tables

1 Introduction

Embedded software-intensive system can be found in many application domains such as mobile devices, vehicles, avionics, buildings, or production systems [36]. Recently, due to an increasing demand on functionality and flexibility, such beforehand isolated systems become interconnected to gain powerful System of Systems (SoS) solutions with an overall robust and flexible emerged behavior. As a consequence, such composite systems consist of independent subsystems, where each subsystem follows an individual strategy to fulfill local requirements. Moreover, the types of subsystems spawn a range from networked embedded systems or cyber-physical systems that are characterized by a tight coupling to the physical environment[1] to highly dynamic self-adaptive systems, which leads to diverse overall system solutions with complex interaction patterns. Additionally, in most cases, each subsystem has some individual self-* capabilities and is aware of its environment to react on changing demands properly [31]. However, the overall interconnected system is open in the sense that new subsystems (e.g., mobile devices) may join or leave over time by providing or removing additional functionalities. Furthermore, individual subsystems cooperate with other subsystems to reach overall system goals [20]. On the one hand, because of the dynamics within the system and an upfront unknown number of subsystem parts, there is a broad range of possible coordination schemes between system parts ranging from central coordination schemes to fully self-organizing solutions. On the other hand, there are high demands on reliability, availability and robustness of the overall system.

[1]The term networked embedded system as well as cyber-physical system is discussed in detail in Section 2.

Those kind of assembled systems are the motivation for our research, which focuses on the modeling perspective of subsystem interaction with the following research question:

Q–1 How can a modeling language comprise interactions between independent system parts in a self-adaptive System of Systems?

Q–2 How can this modeling language for collaborations be used to visualize and simulate dependencies and impact relations, local and global goals, and knowledge propagation between system parts.

Q–3 How can this modeling language be used for formal verification to identify possible system threats or ensuring that the overall system behavior fulfills given requirements?

Q–4 Are there different patterns that support individual use case scenarios?

Q–5 How can a tool support help realizing concepts of the collaboration modeling language with respect to the implementation and execution of the modeling approach?

The research questions ***Q–1***, ***Q–2*** and ***Q–3*** are mainly motivated by Stankovic et al. [95], who emphasized that building complex physical computing systems *"requires a deep understanding of how to model and analyze large-scale systems' behaviors, which necessitate large-scale coordination and cooperation"*.

Regarding to the first and third research questions (***Q–1***, ***Q–3***), an appropriate formalism for a modeling language should be used for enabling further investigation such as verification (e.g., model checking techniques) and simulation. Furthermore, the modeling language should include a well-defined set of clear concepts with a nonambiguous semantic that suits the description of collaborations in self-adaptive SoS.

On basis of ***Q–1***, we can further investigate different realistic scenarios for understanding the influence of subsystem interaction by looking at ***Q–2***. Additionally, we should be enabled to verify the overall system according to given goals to ensure that the assembled system behavior is as expected (***Q–3***). Afterwards, we may identify some best practices, where systems perform well or find typical pattern that may introduce threats in the system by looking at ***Q–4***.

The ***Q–5*** research question is orthogonal to the questions ***Q–1*** to ***Q–4*** and tackles the tool support. This includes the modeling of the systems collaboration, a way to specify and execute verification and simulation scenarios as well as visualizing threats, dependencies, interaction impacts or the current system structure.

Therefore, the tooling should include implementation guidelines for the modeling language concepts, a framework to execute the modeling language or should describe how different modeling aspects can be used in existing tools.

Within this report, we have the following aims. First, before we can start investigating our research questions, we want to clarify our understanding of different terms as for example self-adaptation or runtime models. Additionally, we discuss several kinds of systems as cyber-physical system or system of systems. Therefore, we review a broad spectrum from literature and subsume the existing perspectives and definitions to establish a common understanding. Second, from this unified view on terms and system types, we collect important characteristics for SoS from literature and discuss how those characteristics influence the modeling of subsystem interactions. Third, we derive a requirements catalog from the characteristics a modeling language for collaborations should be aware of. This requirement catalog can be used as basis for further research according to our research question. Fourth, we collect several use cases and scenarios from literature about real system implementations or simulations to visualize the design spectrum of system interaction on the one hand. On the other hand, we can take some of these use cases as an evaluation basis for our modeling approach concerning system collaboration in our future research work. Therefore, we identify how the beforehand collected characteristics for SoS are covered in or influence the system design of the scenario.

The rest of the report is structured as follows. After a discussion about terms and different kinds of systems in Section 2, we describe a set of important characteristics for SoS with the scope of collaborations in Section 3. Afterwards, we derive a set of requirements from the characteristics for modeling collaborations in SoS in Section 4. Additionally, we collect and introduce several real use-case scenarios from literature in Section 5 for giving an overview about the different kinds of self-* systems and their realization in practice. We review related work in Section 6 and finally, we conclude with a discussion about next steps in research in Section 7.

2 Prerequisites

In this section, we discuss two directions that are needed for this report. First, we discuss and define different system types in Section 2.1 that are needed in Section 3, where we derive a set of typical characteristics for SoS. Second, we clarify our understanding of different model-driven techniques and terms in Section 2.2 by subsuming the current state of the art literature and discussing different research viewpoints from that.

2.1 System Types

This section subsumes different system types. First, we start with self-adaptive systems that provide flexible, dynamic behavior by providing so-called self-* properties. Second, we discuss systems that have to integrate physical elements together with software aspects. Therefore, we start with cyber-physical systems and increase the system size, which leads to networks of cyber-physical systems and further to system of systems.

2.1.1 Self-Adaptive System

Definitions for (self-)adaptive systems vary a lot 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 [31, 73]. However, a broad discussion about different viewpoints concerning self-adaptive software is given by Salehie et al. [89]. They identify different domains, as for example autonomic computing [65], 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 (cf. [89]), but rather is continued during system execution. Therefore, the system is able to react on changes in the environment, failures or new requirements at runtime. Second, self-adaptive software introduce so-called *self-** properties into the system as for example *self-configuration*, *self-optimization*, *self-healing*, or *self-protection* (cf. [65]). The key requirement for the self-* capabilities is that the system is aware of its own state

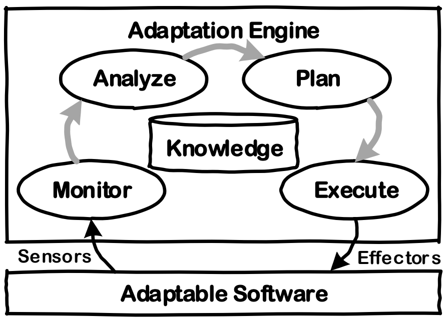


Figure 2.1: MAPE-K feedback loop in an external adaptation engine according to [65]

(self-awareness [74, 101]), requirements/goals (requirements-awareness [91]), or context (context-awareness [101]). As a consequence, for the rest of this report, this *-aware property is enough for our definition for a self-adaptive system.

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 according to the systems' purposes.*

Beside the definition, we are looking only at 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. Furthermore, from the autonomic computing domain, Kephart et al. [65] propose a reference architecture with an *autonomic manager* (also called adaptation engine) that is decoupled from the managed subsystem (also called adaptable software system). In this approach, the managed subsystem 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 in Figure 2.1, Kephart et al. propose four dedicated adaptation activities, namely Monitor, Analyze, Plan, and Execute (MAPE) that are sequentially executed. The MAPE activities share a common knowledge base (MAPE-K) that for example consists of s set of different runtime models. The MAPE activities form a feedback loop that is usually executed in a periodical way. First, the monitor activity retrieves information from the running system that are extracted via (software) sensors and updated in the knowledge base. Second, the analyze activity checks on basis of the updated knowledge whether all system requirements and constraints are met or if an adaptation is needed. 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 translated and executed afterwards. Therefore, the execute activity uses so-called effectors to force the adaptation changes back to the running system.

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 [31]. Therefore, we find adaptive characteristics for systems related to our research scope as discussed in Section 3 and in many real use-cases as shown in Section 5. A comprehensive discussion and a taxonomy of self-* properties is given in [89].

2.1.2 Cyber-Physical System

Cyber-Physical Systems (CPS) evolved from embedded systems that *"combine physical processes with computing"* [71]. Whereas embedded systems are mostly closed, self-contained and do not *"expose the computing capability to the outside"* [71], an increasing number of devices and demands of combined functionalities led to a more and more interconnection of embedded systems to so-called *Networked Embedded System (NES)* [36]. As a consequence, such isolated control systems and afterwards interconnected embedded systems become open to its environment and build different variants of cyber-physical systems that integrate the cyber (software) and physical part [32]. Because 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 [60]. 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 [111].

Although there are several definitions for cyber-physical systems in literature, e.g., from Broy et al. [20], acatech the National Academy of Science [36], or Choi et al. [32], all definition emphasize the tight coupling of physical processes and software computation that is characteristic for a cyber-physical system. Therefore, we use the following short, but precise definition from Lee et al. [72].

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 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 in Section 2.1.1. A comprehensive discussion about CPS and related system types is given by Kim et al. in [67].

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, we emphasize 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 literature, such as Kim et al. [68] or Choi et al. [32], emphasize the distributed nature of a NCPS and the resulting need for coordination schemes between independent subsystems. We combine both viewpoints from [68] and [32] to the following definition, which is used for the rest of this report.

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. [68] stated that the autonomous subsystem must be considered as unreliable and loosely synchronized. We follow this argumentation due to the openness characteristic of the NCPS, where subsystems may arbitrary leave or join the NCPS over time. Additionally, it has to be noted that NCPS must deal with the same problems of the tight coupling between physical components and software as described for CPS in Section 2.1.2. Moreover, often each autonomous subsystem of 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, while considering the shared ownership. However, due to the composition of subsystems that autonomously adapt to their individual contexts into an NCPS, also unwanted co-adaptation effects may emerge from interference of the individual feedback loops placed in the autonomous subsystems (cf. [76]).

2.1.4 System of Systems

If we follow the trend of integrating more and more independent subsystems, we will reach the point, where system borders become unclear. Isolated system solutions become integrated into federations of distributed systems. We refer to such kind of systems by using the term *System of Systems (SoS)* [99]. Note that similar observations concerning the emergence of such systems and the importance of its collaborating subsystems have been made for many related research areas as for example Ultra-Large-Scale Systems (ULSS) [83], and for particular technological domains as software-intensive systems [110], next generation of embedded real-time systems [18, 82, 92, 97], vehicular systems [3], and service-based systems [35].

We define the term SoS following the ideas in the research agenda from [99] and from Gezgin et al. [47] as follows.

Definition ***System of System:*** *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 system solution and the interaction between systems. On the one hand, each system cannot achieve the overall 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 its 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.

2.1.5 Discussion

Depending on the research domain, different viewpoints on the system types exists. We do not claim that we unify those different viewpoints in this report, but rather than give a broad discussion about different kinds of systems by citing the state of the art literature for each system type. Therefore, we will find characteristics in CPS that also appear in NCPS as well as in SoS. However, although there are unclear borders between the system types, we try to pinpoint specific characteristics for SoS in the Section 3. Concerning our research questions as discussed in the introduction, we investigate SoS that are composed of subsystems, which show characteristics from a broad spectrum of CPS. Because of the different terminology between NCPS and SoS, we use both system types synonymously for the rest of this report. For us 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. Already existing scenarios of such SoS are discussed in Section 5.

2.2 Terms and Techniques

While following the definitions of different system types in Section 2.1, we observe that the complexity of nowadays system rises. One approach for tackling the complexity of large software systems is the usage of MDE techniques. The goals of MDE techniques are twofold. First, MDE abstract from *"complexities of the underlying implementation platform"* [42] by handling models as first-class entities during software development. Second, as stated by France et al., MDE *"hide[s] the complexities of runtime phenomena"* [42] 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 our former research, we use different MDE techniques such as model transformation, model checking, or model verification. Additionally, with respect to our research questions as motivated in Section 1, we want to develop a modeling language for collaboration that enables simulation capabilities and system verification. In the following in this section, with this motivation in mind, we clarify important terms that are related to MDE.

2.2.1 Model-Driven Engineering

First, we discuss the model-driven engineering term itself following the definition from France et al. [42].

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 [93], 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 [42].

Related to 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 [77]. Furthermore, the OMG defines different 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 [19, 77]. Appropriate modeling languages, e.g., the Unified Modeling Language (UML) [57] 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 Section 2.2.3. Further discussion about MDA principles can be found in [19, 70, 77].

2.2.2 Model

According to the definition of MDE, models play a key role for system and software development. Therefore, we subsequently discuss the model term in detail. There are varying definitions of a model depending on its purpose of usage. In this report, we follow the definition from our former work in [49] 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.*

As a note, this definition goes hand in hand with the model definition in the context of the MDA given by the OMG in [77, p. 5] and enriches the model definition of [60], 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., class diagrams, component diagrams), the behavior (e.g., automata, activity diagrams), or the interaction between system parts (e.g., 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 large set of models using MDE techniques [59]. If the size of the models or the number increases, further model management techniques are needed to handle those during system development or at runtime as discussed in Section 2.2.5.

2.2.3 Metamodel

As emphasized in Section 2.2.1, metamodels play an important role enabling MDE techniques such as model transformation, simulation and verification. Defining of what a metamodel is, we follow the argumentation in [14] and the definition of the OMG in [77].

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

The word *"defines"* in the definition means that a *"metamodel is a formal specification"* [14] of a model. Therefore, a metamodels provide concepts (define 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 in [14].

2.2.4 Runtime Model

In contrast to development-time models, runtime models provide *"views of some aspect of an executing system and are thus abstractions of runtime phenomena"* [42]. 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 representation at runtime, which may support dynamic adaptation of the system [42]. Bencomo [13] and Blair et al. [16] extend the viewpoint on runtime models requiring a causal connection[1] to the running system. For the rest of this report, we take the following definition of a runtime model from our former work [49] that follows the line of argument from [13, 16, 42].

Definition ***Runtime Model:*** *A runtime model is a model that complies with the model definition (cf. Section 2.2.2) and, in addition, is characterized as follows: part of its purpose is to be employed at runtime in a system and its encoding enables its processing at runtime. The runtime model is causally-connected to the original (running system), meaning that a change in the runtime model triggers a corresponding change in the running system and vice versa.*

In our former work, we describe a modeling language for self-adaptive systems called *Executable Runtime Megamodels (EUREMA)* [103] with respect of modeling the adaptation loop according the MAPE approach [65]. In EUREMA, we use runtime models that can be accessed by the MAPE activities within the modeled feedback loop. Based on the experience with the EUREMA modeling language, we propose

[1]The concept of a causal connection is originally discussed by Maes [74].

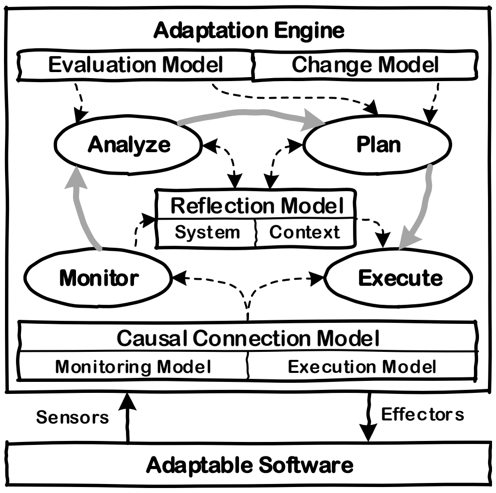


Figure 2.2: MAPE[2] feedback loop with different runtime model types [106]

a categorization of runtime model types in [106]. The main runtime model types of this categorization together with the MAPE activities are depicted in Figure 2.2. Within our categorization, we distinguish between Reflection Models, which describe parts of the running system itself or its context, Causal Connection Models, which are responsible to keep the runtime models synchronized with the running system, adaptation models (Evaluation Model and Change Model in Figure 2.2), which describe the possible configuration space of the system inside the given requirements, and collaboration models (not shown in Figure 2.2), which describe the interaction of the system with other (sub)systems. We refer the interested reader for a detailed discussion about different runtime model types to our former work [106]. In our runtime model taxonomy, only the Causal Connection Models are responsible to directly establish and maintain the causal connection to the running system (adaptable software). Other runtime models (such as Reflection Models) are a representation of the running system that is updated by the adaptation activities. Consequently, we do not require a direct causal connection for all runtime models to the running system as done in [13, 16]. The definition of runtime models above requires a causal connection, which is in our understanding fulfilled, if the runtime model is indirect causal connected. The complete runtime model taxonomy together with a discussion about characteristics for each runtime model type can be found in [106].

In general, runtime models are able to represent system information during system execution [16]. In the context of our research questions, runtime models are for example important for the data exchange within the collaboration of sub-

[2] The MAPE-K feedback loop is introduced in the context of self-adaptive systems in Section 2.1.1.

systems. Furthermore, such runtime models can be dynamically manipulated on a much higher level of abstraction to achieve interoperability, e.g., by automatically transform different formats, force runtime model manipulations to the running system or enabling runtime analysis and verification.

We want to note that *executable models* are a related research direction to runtime models. Both approaches emphasize the runtime representation of key aspect 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. [64] propose an approach named "knowledge-based executable model" for the quantitative evaluation of SoS architectures and claims that it overcomes the limitations of conventional architectural evaluation methods. The executable model 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 SoS architecture neither do not represent the running system nor has a causal connection. Also Kilicay et al. [66] argues for a shift towards executable models and the need for simulation tools. He suggests 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. Xiong et al. [64] and Kilicay et al. [66] do not use the executable models as system representative during runtime but rather for simulation of different system configurations.

2.2.5 Model Management

Because models are the main artifacts in MDE, new challenges concerning the model management arise [59]. In MDE, models are created out of 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, where 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. in [59]. Hebig et al. give an overview about definitions from the two founders of the term megamodel, which are Bézevin (cf. [4, 15]) and Favre (cf. [40]), and provide a unified definition of the term together with a core metamodel for megamodels. For the rest of this report, we take the definition from [59], because the authors already subsume different viewpoints and the state of the art literature.

Definition ***Megamodel:*** *A megamodel is a model (cf. model definition in Section 2.2.2) that contains models and relations between them.*

First of all, we want to use megamodels as model management technique for runtime models. Thus, megamodels can be used to maintain relations between runtime models such as different views or that a requirement model influence parts of the system reflection model. Due to the distributed nature of SoS (see characteristics in Section 3), different runtime model versions may located in different parts of the SoS that can be described in megamodels, too. Additionally, we can use megamodels to determine dependencies between runtime models of collaborating systems.

Concerning the collaboration of subsystems, we need techniques to describe 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. We already start a discussion about impact relation of self-adaptive systems in [106].

The definitions of the different terms from the MDE and system types are used as basis to derive characteristics and specific requirements concerning our research questions in the next sections.

3 Characteristics

As we outlined in Section 2.1, a SoS is an ensemble of systems that comprises a broad spectrum of subsystems types including embedded systems, NES, CPS, and NCPS. Therefore, the overall SoS inherits the characteristics from its contained subsystems that we derive in this section. As basis, we use our discussion of system types from Section 2.1 and common literature.

The main characteristics are already caused by CPS that are stated as a federation of *"open, ubiquitous systems [that] dynamically and automatically reconfigure themselves and cooperate with other CPS"* [20]. Therefore, we derive four main characteristics of a SoS that we name ***C–1*** *open*, ***C–2*** *dynamic*, ***C–3*** *collaborative*, and ***C–4*** *independent*. From these four characteristics, we further refine SoS properties as follows:

C–1 Open: The SoS is considered as open, because it is a federation of subsystems, where an arbitrary number of unknown, potentially heterogeneous system parts may leave or join the overall SoS during system operation at arbitrary point in times (cf. [5, 20, 99]).

- ***C–1.1 Distributed:*** Within the SoS, the integrated, autonomous subsystems are inherent distributed, potentially over larger geographic spaces [47].
- ***C–1.2 Scalable:*** The integration of independent system solutions and for the most cases the self-coordinating, decentralized collaboration scheme between those systems leads to a large-scale federation of systems.
- ***C–1.3 Flexible:*** The flexibility characteristic of the SoS enables the reaction of the system to changes in the environment and system structure. This goes hand in hand with an *elastic* overall system architecture. Furthermore, we call the SoS *agile*, if the SoS is flexible and able to react on changes rapidly [99].

C–2 Dynamic: 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.

- ***C–2.1 Adaptive:*** The SoS is adaptive such that each autonomous subsystem can adapt its structure as well as behavior to the particular needs and constraints in the current SoS configuration. Furthermore, an adaptive SoS exploits the full potential of autonomous, beforehand isolated subsystems by building interconnections and collaborations [99]. Additionally, the SoS handle the emergent behavior by dynamically adapt and absorb deviations in the system structure.

C–2.2 ***Resilient:*** 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 [99].

C–3 ***Collaborative:*** Due to the definition in Section 2.1.4, the independent subsystems in a SoS must be collaborative to achieve common goals.

C–3.1 ***Emergent:*** Due to the collaborative nature, we observe *emergent behavior*, where the overall behavior at the level of the SoS is the result from the local behavior of the contained autonomous subsystems and their interaction [47].

C–3.2 ***Competitive:*** Beside the emergent, collaborative behavior, a SoS is competitive, because each subsystem follows and optimizes its behavior according to local goals (cf. [47]). In the worst case, this leads to contradicting behavior on the SoS level. Consequently, the SoS is responsible of finding appropriate trade-offs between local and global goal optimizations.

C–4 ***Independent:*** As stated in [47], a SoS is independent in the sense that all subsystems are independently designed, developed and managed for their own purposes. Furthermore, subsystems operate independently at runtime except during the collaboration.

C–4.1 ***Evolutionary:*** The *development* of a SoS is characterized by uncoordinated, independent local evolution steps of the autonomous subsystems, 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 [47].

C–4.2 ***Autonomous:*** The SoS is *operational independent* [47], which means that the autonomous subsystems are operated independent from each other and that no global coordination scheme, concerning the operation of the autonomous subsystems, can be assumed.

C–4.3 ***Decentralized:*** The *managerial independence* [47] of a SoS implies that also the management of the autonomous subsystems is not centralized and thus the management decisions for different autonomous subsystems may be conflicting.

C–4.4 Concurrent: 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 subsystems due to established collaboration connections.

C–4.5 Incomplete: Concerning the knowledge, each independent subsystem has always only incomplete views of the overall SoS and the environment. These views may be extended during the collaboration with other subsystems or by sensing the local environment. Furthermore, due to concurrency effects and decentralized coordination, the SoS has to deal with inconsistencies by aggregating and distribution of the knowledge to subsystems. In general, a complete and comprehensive view on the overall SoS state with an aggregated, unified view of the knowledge cannot be given.

Concerning our research questions, we consider the characteristics above and how they influence the design as well as concepts of our collaboration modeling language. The openness characteristic ***C–1*** of a SoS implies that the overall system is composed of a varying number of subsystems. As a consequence, clear system borders cannot be defined because subsystems interactively join and leave the overall SoS at arbitrary point in times. Furthermore, the overall behavior of the SoS emerges from the available subsystems, where the size and number of subsystem is 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 ***C–1.1*** distributed characteristic. This distribution of the system can be twofold. In the first aspect, the SoS can be logical 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 (cf. [99]). Beside the distribution aspect, the SoS offers high potentials concerning ***C–1.2*** scalability and ***C–1.3*** flexibility characteristics. The resulting elasticity and agility of the overall SoS (cf. [99]) is needed to cope with the permanent changing inner system architecture as well as with the interaction with an unknown affecting environment. 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, we must be aware that new collaboration capabilities may arise in 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"* [72]. 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 subsystem emerges in the overall SoS. Furthermore, the ***C–2.1*** adaptation capabilities of the SoS in combination with the ***C–1.3*** flexibility capabilities lead to the ***C–2.2*** resilient (robustness) characteristic of the SoS in the sense that the likelihood of catastrophic and single point of failures is drastically reduced (cf. [99]). From the modeling perspective, the modeling of feedback loops is well known for the 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 subsystems and its interaction with other subsystems has to be covered as well.

The importance of modeling the interaction aspects between subsystems is strengthened by the ***C–3*** collaborative characteristic of a SoS, which can be derived from the ***C–1*** openness and ***C–2*** dynamics capabilities. Usually, each autonomous subsystem of a SoS 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. Therefore, on the one hand, the overall system behavior ***C–3.1*** emerges from the contained subsystems in the sense that different system parts work together to reach global goals that cannot be achieved by a single subsystem. On the other hand, each subsystem behaves according to local optimization strategies that might influence or even worse contradict other subsystems. This phenomenon is described by the ***C–3.2*** competitive characteristic, where the SoS is responsible of finding appropriate trade-offs between local and global requirements (cf. [47]). 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 (cf. [76]). A modeling language for collaboration should comprise such effects and should be able to visualize as well as analyze them.

Ruling such SoS is challenging due to the complexity (***C–1***), dynamics (***C–2***), and emergence (***C–3***). The fourth characteristic ***C–4*** copes with the independence of a SoS. At first, the autonomous subsystems ***C–4.1*** evolve over time. This holds for the evolutionary development cycles as well as for the evolution of the subsystem during system execution (e.g., due maintenance activities cf. [31, 73]). Such local evolutions steps in independent subsystems leads to uncoordinated evolution in the overall SoS. Another aspect is the operational independence [47] of the subsystems within the SoS that we call the ***C–4.2*** autonomous SoS characteristic, because, in general, we do not assume a global coordination scheme controlling the operation of all subsystems. Therefore, each subsystem must be seen as operational independent from the other subsystems following local strategies. The operational independence goes hand in hand with managerial independence [47] of the SoS. In general, no central instance exists that is able to coordinate the interaction between all subsystems and determine all local management decisions. Thus, the SoS is characterized by a ***C–4.3*** decentralized coordination scheme. Additionally, the ***C–4.4*** concurrent characteristic inherently arise due to the independent behavioral effects of the subsystems.

The last, ***C–4.5*** incomplete characteristic focus on the partial knowledge aspect in the SoS. Each subsystem has always a local, incomplete view on 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 SoS should be aware of different local views in the subsystems, outdated knowledge 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.

In summary, considering the discussed characteristics, it is highly attractive that SoS 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 SoS architectures, and to adapt to open and dynamic contexts, while considering the shared ownership of knowledge inside the subsystems. As a next step, we derive important requirements for our collaboration modeling language from the discussed characteristics in the next Section 4.

4 Modeling Language Requirements

According our research questions in the introduction and discussed characteristics of SoS in Section 3, the engineering of self-adaptive SoS must explicitly cover the coordination among the individual subsystems. Because the subsystems itself usually have adaptive capabilities, which further can be realized in form of feedback loops (cf. [22, 31, 72]), we assume that the overall SoS is composed of subsystems that contains such feedback loops. Furthermore, we focus on SAS that follow the MAPE-K blueprint from Kephart et al. [65] as discussed in Section 2.1.1. This imposes several requirements for designing (modeling) and realizing the coordination, which have been mainly derived from [109].

The major requirement for modeling is to explicitly support interacting feedback loops during the design and runtime of the SoS, particularly, by specifying the coordination as discussed in the following.

R–1* *Degree of decentralization: Since there is no dichotomy of centralized and decentralized control, there is a range of different patterns determining the degree of decentralization [109]. For example, SoS subsystems 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. The modeling language should therefore support the whole spectrum of decentralization in self-adaptive SoS, which influences the type of coordination.

R–2* *Degree of distribution: As emphasized in ***C–1.1***, SoS are potentially distributed over large geographic spaces. Therefore, a modeling language for collaborations should be aware that interacting subsystems can be distributed over large distances or may run on different execution nodes. This would affect, among others, the communication mechanism between subsystems, may cause timing delays during knowledge exchange, or cause security issues by crossing subsystem boundaries over collaborations. Consequently, as the degree of decentralization, the distance between subsystems affect the type of coordination and therefore, a modeling language for collaboration should distinguish between variants of *inter-loop* and *intra-loop* coordination [104].

R–3 ***Types of coordination:*** The modeling language should allow the specification of different types of coordination such as delegating tasks among MAPE activities or coordinating individual MAPE activities. For the former case a typical example could be a client-server scenario, where the client (e.g., a mobile phone) has limited computational power or want to save energy. Therefore, the client may delegate complex analysis and planning tasks to the more powerful server node. The coordination of MAPE activities is useful in distributed scenarios. As an example, mobile robots may coordinate their monitoring by exchanging observations about their local environment to achieve a higher coverage during exploration of its surroundings. The coordination type is determined by its participants and the protocol between the participants.

R–4 ***Coordination roles:*** The modeling language should cover the specification of the participants when individual MAPE activities coordinate to accomplish self-adaptation. This includes the specification of the local behavior of MAPE activities relevant for the coordination. It seems reasonable to specify the coordinating participants by means of *roles* to foster separation of concerns with respect to a participant's behavior relevant and irrelevant to the coordination. For instance, a mobile phone, which behaves as a client (role), should follow instructions from a participant that acts as a server (role). The interplay between these two roles should be specified in a protocol to achieve the aim of the coordination. Likewise, some processing or filtering of exchanged knowledge according the current role is conceivable before sending or after receiving the knowledge.

R–5 ***Coordination protocol:*** To specify the coordination behavior among distributed MAPE activities, the modeling language should support the specification of the employed protocol, that is, the sequence of interactions among them such as push-pull, request-reply, or negotiation. This corresponds to the sequence of exchanged messages among MAPE activities. Additionally, managerial independence (cf. ***C–4***, ***C–4.3***) of subsystems requires well-defined collaboration contracts and interfaces that define in which form the required knowledge is exchanged. For instance, according to a client-server scenario, the client has to authenticate itself by the server first, before the server node starts offering a service to the client. Another variant is the use of a gossip protocol by the server to inform all possible clients about the available pool of provided functionality.

R–6 ***Knowledge representation:*** Feedback loops are characterized by MAPE activities sharing *knowledge* and thus, the knowledge influences the coordination. The modeling language should support runtime representations of knowledge with a clear semantic. Models at runtime [16] (as discussed in Section 2.2.4) are able to represent system information during system execution. Furthermore, data formats must be transformed to link different subsystems 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 transform different formats, force 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.

R–7 ***Knowledge exchange:*** Beside the semantic of knowledge, the modeling language should support the specification of which, how much, and in which way the knowledge is exchanged. Therefore, when specifying a coordination protocol, 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 if this processing is relevant for the coordination (cf. local behavior of ***R–4*** *Coordination roles*). 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 (***C–1.1***), 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 are collected over a time period, of a specific time frame or knowledge that is not older than a given timestamp. For example, if independent robots exchange their information about the observed local environment with each other, they must specify how the observations are represented (cf. ***R–6***, e.g., in a runtime model), which observations are exchanged (e.g., observed obstacles, robot positions, waypoints), and how old/complete the data is. The explicit modeling of knowledge and its usage (local and via collaborations) enables further impact analysis in the sense that changes in the knowledge base by one participant may influence (over the collaboration) the behavior of an interacting subsystem.

R–8 Communication Paradigm: Besides specifying the roles and protocol of a coordination, the underlying communication paradigm has to be defined, especially as the paradigm 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 offer a library for common communication paradigm and can hide implementation details for realizing the communication infrastructure. For example, due to an unreliable network, a mobile client should communicate asynchronously by directly exchanging messages to the server.

R–9 Pattern catalog: Being able to specify the coordination among distributed MAPE activities with all its facets as just discussed is a promising way to model in detail various control patterns [109]. Having a uniform modeling approach enables the specification, comparison, and reuse of solutions to build a pattern catalog at different levels of abstraction. In this context, the client-server example mentioned above is an instance of the *master/slave pattern*. Respectively, the exploring robots example (cf. ***R–3***, ***R–7***) shows some characteristics of the *information sharing pattern*. Both patterns are discussed in [109]. 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. With respect to our research questions, standardization concerns are beyond the scope of this report.

Based on the requirements ***R–1*** to ***R–9*** for the specification of a coordination, the requirements concerning the realization of such a specification comprise the monitoring, analysis and execution of the specified collaboration concepts as described in the following.

R–10 Monitoring of Collaboration: The realizing framework of the modeling language should support monitoring capabilities of the specified collaboration concepts for retrieving runtime information about the joint interaction of subsystems. This requires well defined interfaces that can be provided by well-known middlewares, standards or frameworks, which bridge the gap between modeled collaborations and realistic runtime introspection. Furthermore, model-driven techniques such as runtime models on a higher layer can give proper abstractions to represent key information of interest from the specified collaboration as well as may help processing, exchanging, updating, and merging the retrieved knowledge at runtime.

R–11 Analysis of Collaboration: Developing coordinating MAPE activities requires analysis, for instance, to ensure that an activity performing a certain role fulfills the behavior as defined by the coordination. This can be considered as consistency between the behavior of an activity and the behavior defined for a role in the coordination. Other analysis capabilities can be concerned with the effects of knowledge access and distribution to ensure that a certain role has only access to or is influenced by the knowledge specified in the collaboration. Thus, when implementing or even executing coordinating activities, means to ensure that the roles and protocols are properly realized as required. Based on the monitoring capabilities as discussed in ***R–10***, the modeling language and its realization should be analyzable to ensure that modeled collaboration constraints are fulfilled during execution.

R–12 Implementation and Execution of Collaboration: After specifying a coordination, guidelines for supporting the implementation of the coordination should be provided to ease the development. 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 mapping of the coordination specification to other specific frameworks or models specifically targeting the development of self-adaptive software, which would show how the coordination could be realized by already existing mechanisms and tools. This additionally facilitates the use of existing runtime environments, such as middleware employed in frameworks, to execute distributed self-adaptive software.

Having discussed the requirements for modeling and realizing coordination within a SoS, we discuss several real-world scenarios and implementations for different system types in the following sections.

5 Scenarios

In this section, we describe a broad spectrum of real scenarios in the context of SAS, CPS, NCPS, and SoS from literature. An overview of all scenarios is given in Table 5.1, where we provide the primary system type, a short description, the key self-* capability and the main coordination scheme of the scenarios. In the following, we sequentially discuss each scenario as listed in Table 5.1 in detail and emphasize the key aspects that are especially of interest for the collaboration modeling language. Afterwards, we close this section with a mapping of our derived characteristics from Section 3 to the presented scenarios and discuss for the corresponding scenario how those characteristics may influence the modeling of collaboration aspects. Furthermore, we emphasize how the scenarios reflect the elaborated requirements of Section 4.

5.1 Self-Adaptive Systems

In the context of SAS, Vromant et al. [104] presents a traffic monitoring system with self-healing capabilities. The system consists of a set of intelligent cameras that are equally distributed along the highway. Thereby, each camera has a specific viewing range of the road. The aim of the camera system is the decentralized detection of traffic jams. If a traffic jam is detected, cameras group each other into so-called *organizations*, where the viewing range of each camera in the organization detects the same traffic jam. This scenario supports two adaptive strategies. First, the grouping, maintaining and restructuring of organizations according to the current traffic situation. Second, a self-healing mechanism of the overall system that allows the failing of cameras during operation, which may enforce restructuring of existing organizations or may change the behavior of neighboring cameras.

In the context of the Rainbow framework, Garland et al. [45] describe two SAS case studies, which are a web-based client-server system and a videoconferencing system. The web-based client-server system consists of server groups. Furthermore, each server group maintains several servers and is a connection point for an arbitrary number of clients. The goal of the system is keeping the response time for client request under a predefined maximum threshold. The response time depends on two key aspects, namely, server load and available bandwidth. In summary, the

Table 5.1: Overview of SAS, CPS, NCPS, and SoS scenarios

Source	Type	Description	Self-* Capability	Coordination and Properties
[104]	SAS	traffic monitoring system	self-healing	distributed, decentralized, collaborative
[45]	SAS	*RAINBOW* framework, web-based client-server system	self-optimizing	hierarchical
[45]	SAS	*RAINBOW* framework, videoconferencing system	self-optimizing	hierarchical
[69]	SAS	audio streaming system	self-organizing self-optimizing	decentralized, real-time, agent-based, market-based
[54]	SAS, CPS	mobile learning application	self-awareness, self-healing	distributed, hierarchical, collaborative
[102]	SAS	EJB-based web shop	self-healing	runtime models
[39]	CPS	cooperative vehicle safety system	situation-awareness, context-awareness	decentralized, collaborative, real-time
[113]	CPS, NES	multi-layered control approach	self-healing	hierarchical, real-time, safety-critical
[29]	CPS	smart home	context-aware, self-configuration, self-healing	feature runtime models
[95]	CPS, NCPS	emergency response system	situation-aware, context-aware	distributed, real-time, safety-critical, mission-critical
[49]	CPS, NCPS	factory automation simulation	context-aware, self-optimizing	distributed, runtime models, safety-critical, uncertainty
[68]	SoS	autonomous robots and isolated wireless sensors	context-aware, self-optimization, self-healing	distributed, decentralized, collaborative
[17]	SoS	autonomous vehicles crossing an unsignalised junction	context-aware	decentralized, collaborative, real-time, safety-critical
[85]	SoS	search and rescue with a fleet of heterogeneous, autonomous robots	context-aware, self-optimizing	distributed, decentralized, collaborative
[98]	SoS	*PLASMA*, robot convoy	context-aware, situation-aware	distributed, hierarchical, collaborative
[27]	SoS	Railcab, train platooning	context-aware, self-optimizing	distributed, collaborative, real-time, safety-critical
[30]	SoS	*SATRE* project, vehicle platooning	depend on scenario	distributed, hierarchical, collaborative, real-time, safety-critical
[112]	SoS	smart warehouse with robots	self-optimizing	hierarchical
[38]	SoS	intelligent transportation system catalog	depend on scenario, e.g., self-optimizing	depend on category, e.g., collaborative, safety-critical
[95]	SoS	*SISAL*, assisted living	situation-awareness, context-aware, self-healing	distributed, safety-critical, mission-critical

system adaptively optimizes incoming client requests according to the given key performance indicators by adding servers or migrating clients to other available server groups. The second case study in [45] is about a videoconferencing system that adaptively optimize for the two given concerns performance and cost. In this scenario, different clients may join or leave the video conference over time, which opens the adaptation space of removing or replacing gateways or proxy server depending on the current situation. A very interesting aspect of this scenario is the possible conflict of the two adaptation concerns. Increasing the performance of the system within one adaptation loop (e.g., by adding more gateways for a faster video transmission) may contradict the other adaptation loop that tries to reduce cost.

A completely decentralized use case of a self-organizing multi-agent system is described by Klein et al. [69]. In this example, an audio streaming system must apply a sequence of different filter operations on audio packets before it can be provided to the client. The overall sequence must be performed within hard real-time constraints. As a consequence, if the sequence of filter is not applied in the given amount of time, the audio packet is dropped. Independent agents can perform one specific filter operation for one audio packet at a time. Furthermore, agents can reconfigure its behavior to provide another filter operation, whereas the reconfiguration takes time too and agents cannot operate during reconfiguration. Agents work fully autonomous in a self-interested manner without any coordination to other agents. Additionally, agents or the hardware that executes a specific amount of agents may fail over time. 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 [54]. Students with mobile phones build up groups into so-called *Mobile Virtual Devices (MVD)*, where each group work on given tasks. Within a group, one mobile device is elected as master device that communicate with a server to gather new tasks and report results back. All other devices in the MVD monitor themselves and provide data of interest to the master device. The tasks depend of, among other, the GPS signal of the mobile devices in a MVD. Each device runs a local feedback loops that enables self-awareness. If the GPS signal quality becomes worse, the master device starts a self-healing adaptation loop, which is distributed over all devices in the MVD, for providing enough functionality so that the students can still work on their tasks. For example, other available devices can join the MVD or further GPS capabilities are enabled. In contrast to the decentralized scenario in [69], the described approach of Iglesia et al. [54] uses a centralized master-slave coordination pattern between MVDs and the server and for all devices insight the MVD.

An example for the usage of runtime models that represent key information of the running system in the context of SAS are discussed in our former work in [102]. In this paper, we describe a self-healing scenario of an *Enterprise Java Beans*-based web shop. The web shop monitors running components and is able via structural adaptation to replace or restart faulty components at runtime. The key aspect in the paper [102] is the demonstration how model-driven techniques such as model transformation and pattern matching can help creating appropriate runtime model representation of the system, applying (synchronize) changes in the runtime model back to the system and analyzing runtime models with predefined graph patterns for detecting adaptation issues in the system.

5.2 Networked Embedded and Cyber-Physical Systems

In the context of a CPS, Fallah et al. [39] describes two cooperative vehicle safety (CVS) scenarios, namely *active safety* and *situational awareness safety*. In the active safety scenario, vehicles continuously monitor and receive data from the environment for predicting possible threats (e.g., collisions). Systems that realize such a collision warning functionality for enabling active safety in a vehicle must provide information about possible threats within a latency of lower than few hundred milliseconds (cf. [39, 94, 105]). As a consequence, such systems operate under strict real-time constraints. Missing a deadline, e.g., providing the collision warning to late, may lead to a situation, where the accident cannot be avoided by the driver. In contrast to the active safety scenario, a situation awareness application has less timing constraints informing the driver about possible hazards in the future. A warning about a traffic jam in 5 kilometer distance is one situation awareness example.

Another scenario from the automotive domain is given by Zeller et al. [113], which describes a multi-layered self-adaptation approach for enabling self-healing. The authors take a realistic example and simulate different system variants of a car with up to 100 electronic control units,[1] a communication infrastructure and different software functions. Zeller et al. distribute the functions (in form of software components) over the available ECUs and group them according to different variants as for example required safety integrity levels or available network topology on different software layers. Each layer hierarchical controls the layer below. In

[1]An electronic control unit (ECU) is a small interconnected, embedded electronic device in a car that realizes the needed software functionality. The software functions range from controlling the engine over safety functionalities as for example ABS or ESP to non-critical infotainment application as radio, multimedia or navigation.

the case of a hardware (ECU) failure, local adaptation loops try to heal the system by shifting software components from the broken ECU to available resources in the system. If the local adaptation fails, the upper layer is informed, which again tries to heal the system. Therefore, in this scenario, there are multiple control loops within different layers that are coordinated in a hierarchical manner.

Another domain of a CPS are smart homes, where Cetina et al. [29] sketches different scenarios such as self-configuration or self-healing to improve the system according to changing user needs. For example, an adaptable system for smart homes must support the incorporation of new devices as well as the handling of device failures. In other scenarios, the system has to cope with changing user profiles or must fulfill requirements of different users at the same time. Cetina et al. [29] describe a model-based reconfiguration engine to cope with the described situations running a feedback loop on top of a runtime model representation of the smart home.

Stankovic et al. describe typical settings for emergency response systems in [95] with focus on a massive amount of sensor nodes. They emphasized that in case of earthquakes, forest fires or other disasters, large sensor networks may enable the fast retrieving of necessary information from the geographic area to derive further appropriate disaster control strategies. A key aspect of such a scenario is the deployment of a massive amount of independent sensor nodes that build up a sensor network. Due to sensor failures, changing environmental conditions, limited network capabilities, and timing conditions, the overall system has to be highly adaptive to the current situation. Despite of the harsh conditions, the system has to provide a minimal, robust amount of functionality until the disaster could be eliminated. Such disaster scenarios describe a typical kind of systems that are composed of different applications and hardware parts (e.g., sensor nodes), interact or influence directly with an unpredictable environment, and are in the most cases safety- of mission-critical.

In [49], we introduce a simulated factory automation scenario that consists of three autonomous robots. These robots must coordinate each other by transporting virtual products through an unknown and uncertain environment. Each robot is equipped with several sensors (e.g., infrared distance sensors, laser scanner, indoor GPS system) to monitor its current situation (e.g., battery level) and the environment. Each robot may adapt its current behavior regulated by its battery level, pending tasks, goals, environmental situation and possible collaborations with other participants. Key aspect of the paper [49] is the discussion about identifying and handling uncertainty within this distributed CPS (e.g., in the runtime model representations) at development and runtime.

5.3 Networked Cyber-Physical and System of Systems

Kim et al. [68] introduce a case study,[2] where autonomous, mobile robots collect data from wireless sensors. These sensors are isolated and statically deployed in space. Each robot computes the most effective path through the placed sensors following a multiple Traveling Salesmen (mTSP) algorithm. The load distribution for each robot (sensor places, path length) is computed in a fully decentralized manner and copes with robot failures. Along the path, robots act as data collectors. Furthermore, robots opportunistically exchange collected data about other robots and already covered sensors. The proposed framework in [68] is further applied in different applications as described in [32].

Bouroche et al. [17] describe a scenario, where autonomous vehicles have to cross an unsignalised junction. The vehicles can coordinate each other as soon as they reach a critical zone around the junction and adapt its behavior accordingly by fulfilling overall safety constraints. A vehicle is aware of the junction and can sense whether it is free or not. As global constraint, there is at any time at most one vehicle on the junction, whereas the number of vehicles and the arriving direction at the junction is not known in advance. As soft goal, vehicles should not stop or slow down by crossing the junction if it is free. We consider the scenario of Bouroche et al. [17] as NCPS, because the vehicles temporary communicate via a wireless infrastructure and therefore, form the overall varying system structure.

Ortiz et al. [85] uses a fleet of autonomous, heterogeneous robots in a "search and rescue" scenario. The robots collaborate with each other to explore an unknown environment, create a visual representation, find an object of interest (e.g., an injured person), and maximize sensor coverage of the area (e.g., establishing a proper ad-hoc communication infrastructure). In that scenario, robots join different groups depending on its sensor and actor capabilities. Furthermore, because of the spatial distribution of robots in combination with a limited communication infrastructure, the robots use a distributed task allocation algorithm and directly collaborate with other robots in range. To achieve the overall goal in this use case, the distributed system must cope with a large, unknown environment, partial communication and environmental data as well as robot or sensor failures at every point in time.

In [98], the authors describe a convoy scenario. Three or more autonomous robots follow a provided path in form of spatial coordinates (waypoints). One dedicated leader robot is aware of the path information and guides an arbitrary number of follower robots through the environment. Each follower robot moves and keeps

[2]More information about the case study from [68] can be found on this website: `http://ncps.csl.sri.com/demo.htm`.

safety distance to the robot in front autonomously but coordinates and shares sensor information with the leader robot.

The same platooning use case but in the train transportation domain is described in our former work [27]. A network of autonomous, self-optimizing shuttles in the *Railcab* project[3] move on an existing rail track and individually build platoons with other shuttles on the track. Within the platoon, the shuttles travel with only a few centimeters between them and therefore, must coordinate and reconfigure its behavior under hard real-time conditions to ensure safety constraints (cf. [27, 43]). As a consequence, the shuttles form a distributed NCPS[4] due to the varying collaborations within the platoons and the exchanged information over temporary established communication links.

Furthermore, the automotive domain, especially the European *Safe Road Trains for the Environment* (SARTRE) project[5] as introduced in [30], extensively investigate several research questions around autonomous driving cars in vehicle platoons. Within the platoon, a dedicated leader takes over control about arbitrary follower cars. In addition, follower cars can join and leave the platoon at arbitrary point in times. The key motivation behind autonomous driving vehicles within a platoon is for example saving resources such as fuel during the movement in the platoon and safety issues such as reducing the overall risk of car accidents.

Wurman et al. [112] describes a distributed multi-agent system,[6] where autonomous robots carry mobile storage units in a warehouse. The robots move from its storage box to a packing station, where a human takes goods from the mobile storage unit and pack it into boxes for shipping. Afterwards, the robot can move back to the storage box. Robots calculate the path to the packing station and back to the storage by its own but retrieve moving tasks from a global instance. This scenario is very interesting for testing several optimization strategies looking at different key aspects as for example reducing path length of a robot, increasing throughput of packing goods by decreasing the overall costs (needed robots and storage capabilities).

[3]More information about the Railcab project can be found on the official website: `http://www.railcab.de/`.

[4]Originally, each shuttle in [27] is described as mechatronic system that is characterized by mechanical parts (e.g., such as brakes or engines), electronic components (including the software) and the interaction between those. For our understanding as discussed in Section 2.1.2, this goes hand in hand with our definition of a CPS, which combines physical and cyber parts of a system.

[5]More information about the SARTRE project can be found on the official website: `http://www.sartre-project.eu`.

[6]We put this use case under the NCPS section, because the autonomous robots together with the warehouse system evolve into an overall connected network of independent subsystems.

The European Telecommunications Standards Institute (ETSI) defines a use case catalog in [38] concerning intelligent transportation systems (ITS). An ITS consists of independent cars, service centers, roadside stations and a communication infrastructure. The ETSI defines different categories as for example *co-operative road safety* and *traffic efficiency* for such an ITS. Furthermore, they provide for each category a set of concrete use cases, such as vehicle status warnings, traffic hazard warnings, or collision risk warnings, an ITS may support.

Stankovic et al. outlined a scenario for a SoS called *sensor information system for assisted living* (SISAL) in [95], where a broad spectrum of intelligent monitors, together with special devices, and applications should assist elderly people as well as improve their living standard. The authors mention a broad spectrum of applications that can be combined into an overall SISAL system (which is in fact a SoS). Examples range from intelligent heart beat monitors, which detect irregular patterns, robotic helpers that may for example improve mobility, to smart pantries, which automate the buying process of required food or drug items.

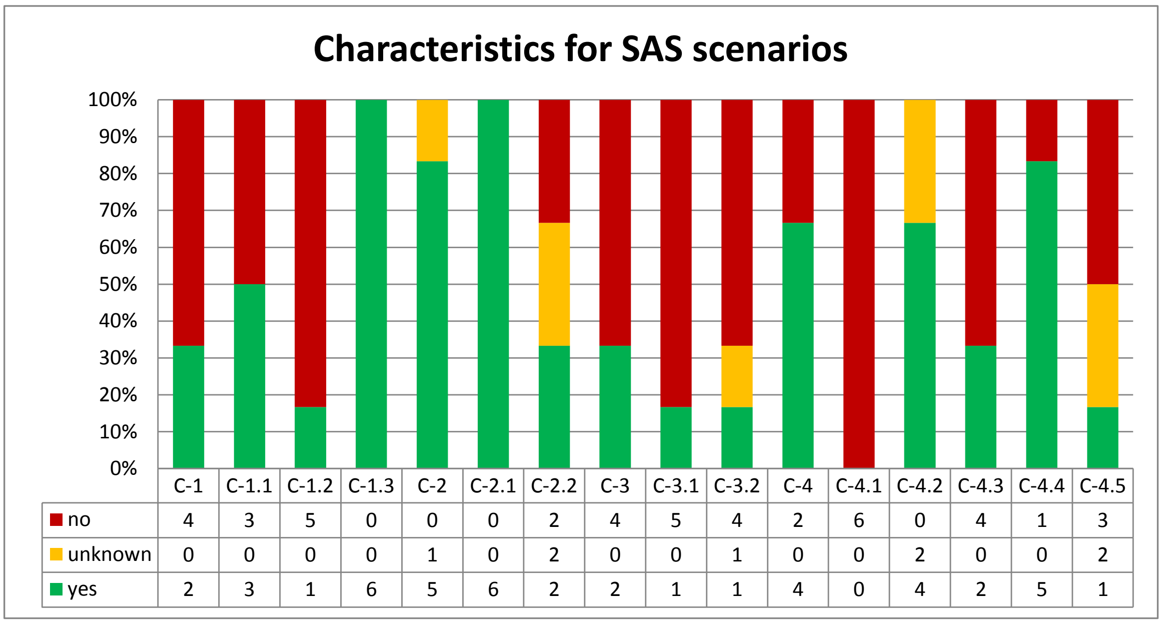
5.4 Discussion of Scenario Characteristics

After we introduced several scenarios for SAS, CPS, and SoS system types, we subsume for each scenario the characteristics following our structure as discussed in Section 3. Therefore, we systematically reviewed the literature of the scenario description and decide for each characteristic ***C–1*** to ***C–4***, if it is applicable to the scenario. Table 5.2 shows an overview of mapped characteristics, where a *check mark* (✔) denotes that the characteristic is found in the scenario, a *minus* (–) denotes that we do not have enough information to decide whether the scenario has this characteristic or not, and a *cross* (✘) means the scenario does not have this characteristic. The scenarios are grouped by the system type according to the introduction in the former subsections. Furthermore, we individually count for the SAS, CPS, and SoS system type how often each characteristic (does not) occur.

From the overview in Table 5.2, we see a first not surprising trend that the number of found characteristics increases depending on the typical system size. Therefore, we found much more characteristics in a SoS than in a SAS setting. We want to note that the overview in Table 5.2 cannot be seen as representative statistic but rather gives a first impression about a broad spectrum of different scenarios from various application domains. However, we argue that this impression is enough to get an understanding of typical coordination problems in different domains and how they are solved in the related scenarios. Thus, in the following, we discuss the found characteristics for each system type in more detail.

Table 5.2: Overview of scenario characteristics (✓ : scenario has this characteristic; – : not enough information; ✗ : scenario does not show characteristic)

Source	Type	Description	Characteristics															
			C-1				C-2			C-3			C-4					
				C-1.1	C-1.2	C-1.3		C-2.1	C-2.2		C-3.1	C-3.2		C-4.1	C-4.2	C-4.3	C-4.4	C-4.5
[104]	*SAS*	traffic monitoring system	✗	✓	✗	✓	✓	✓	–	✓	✗	✗	✓	✗	✓	✓	✓	✓
[45]		Rainbow framework, web-based client-server system	✓	✗	✓	✓	–	✓	✗	✗	✗	✗	✗	✗	–	✗	✓	✗
[45]		Rainbow framework, videoconferencing system	✓	✓	✗	✓	✓	✓	✗	✗	✗	✓	✓	✗	✓	✗	✓	✗
[69]		audio streaming system	✗	✗	✗	✓	✓	✓	✓	✗	✓	–	✓	✗	✓	✓	✓	–
[54]		mobile learning application	✗	✓	✗	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✗	✓	–
[102]		EJB-based web shop	✗	✗	✗	✓	✓	✓	–	✗	✗	✗	✗	✗	–	✗	✗	✗
SAS		✓	2	3	1	6	5	6	2	2	1	1	4	0	4	2	5	1
		–	0	0	0	0	1	0	2	0	0	1	0	0	2	0	0	2
		✗	4	3	5	0	0	0	2	4	5	4	2	6	0	4	1	3
[39]	*CPS*	cooperative vehicle safety system	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓
[113]		multi-layered control approach	✗	✓	✗	✗	✗	✓	✓	–	✗	✗	✓	✗	✓	✗	✓	✗
[29]		smart home	✓	✓	✗	✓	✓	✓	✗	–	✓	✗	✓	✗	✓	✗	✓	–
[95]		emergency response system	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓
[49]		factory automation simulation	✓	✓	✗	✓	✓	✓	✗	✓	✓	✗	✓	✗	✓	✓	✓	✓
CPS		✓	4	5	1	4	4	5	3	3	3	0	5	0	5	3	5	3
		–	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	1
		✗	1	0	4	1	1	0	2	0	2	5	0	5	0	2	0	1
[68]	*SoS*	autonomous robots and isolated wireless sensors	✓	✓	✗	✓	✓	✓	✗	✓	✗	✗	✓	✗	✓	✓	✓	✓
[17]		autonomous vehicles crossing an unsignalised junction	✓	✓	✓	✓	✓	✓	✓	✓	✗	–	✓	✓	✓	✓	✓	✓
[85]		search and rescue with a fleet of heterogeneous, autonomous robots	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓
[98]		PLASMA, robot convoy	✓	✓	✗	–	✓	✓	✗	✓	✗	✗	✓	–	✓	✗	✓	–
[27]		Railcab, train platooning	✓	✓	✗	✓	✓	✓	✓	✓	–	✗	✓	–	✓	✓	✓	✓
[30]		SATRE project, vehicle platooning	✓	✓	✗	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✗	✓	✓
[112]		smart warehouse with robots	–	✓	✓	✓	✓	✓	–	–	–	–	✓	–	✓	✗	✓	✓
[38]		intelligent transportation system catalog	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
[95]		SISAL, assisted living	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SoS		✓	8	9	5	8	9	9	6	8	3	2	9	4	9	6	9	8
		–	1	0	0	1	0	0	1	1	2	2	0	3	0	0	0	1
		✗	0	0	4	0	0	0	2	0	4	5	0	2	0	3	0	0



	C-1	C-1.1	C-1.2	C-1.3	C-2	C-2.1	C-2.2	C-3	C-3.1	C-3.2	C-4	C-4.1	C-4.2	C-4.3	C-4.4	C-4.5
🟥 no	4	3	5	0	0	0	2	4	5	4	2	6	0	4	1	3
🟨 unknown	0	0	0	0	1	0	2	0	0	1	0	0	2	0	0	2
🟩 yes	2	3	1	6	5	6	2	2	1	1	4	0	4	2	5	1

Figure 5.1: Characteristics in SAS scenarios

5.4.1 SAS Characteristics

Figure 5.1 depicts the occurrence of found characteristics in the presented SAS scenarios. First, we can observe that the openness (***C–1***) characteristic is not the main focus of this system type. Thus, distribution aspects (***C–1.1***) and scalability (***C–1.2***) are no typical phenomena, too.

According to our discussion of this system type in Section 2.1.1, such systems are characterized by dynamically react on environmental or requirement changes introducing self-* capabilities. Therefore, we found in almost all our described SAS scenarios the flexibility (***C–1.3***), dynamic (***C–2***), and adaptive (***C–2.1***) characteristic (cf. Figure 5.1), which goes hand in hand of our understanding of SAS.

On the other hand, we do not found often characteristics according to the collaboration (***C–3***) aspect. We argue that typically in small SAS only a few self-* capabilities are implemented or even limited to one adaptation loop controlling one specific aspect (such as self-healing or self-optimization) of the system. Therefore, the need of collaborating subsystems (***C–3***) not exists. Furthermore, in such isolated system solutions, we cannot observe emergent behavior (***C–3.1***) and competitive (***C–3.2***) subsystems, which is reflected in Figure 5.1. In a SAS, competitive behavior may arise if multiple feedback loops follow contradicting strategies. For example, a feedback loop optimizing the energy consumption of a system may contradict another feedback loop that tries to increase the throughput. However, because SAS are mostly not composed at runtime (cf. missing ***C–1*** characteristic as discussed above), such contradicting strategies can often resolved at design time of the system.

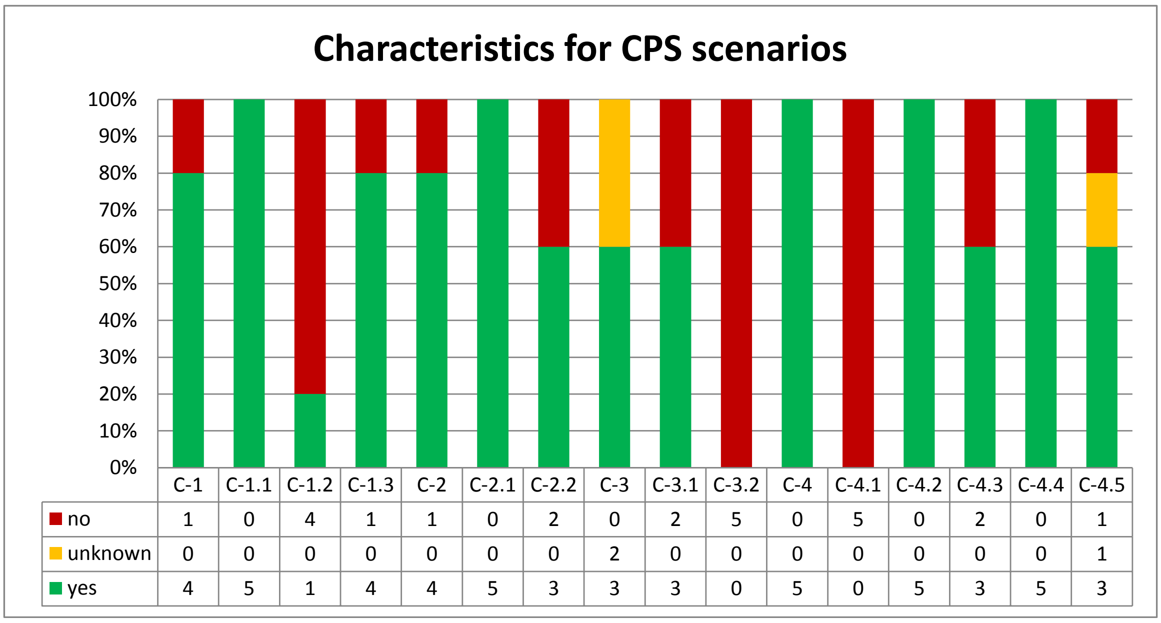
Finally, if we look at the independence characteristics, we can identify the important characteristics autonomy (***C*–4.2**) and concurrency (***C*–4.4**). The former is caused by the adaptive (***C*–2.1**) characteristic. The latter can be explained by the external adaptation approach that is used in all our scenarios, where we can observe at least the two running system parts that are the adaptation engine and the adaptable software (cf. discussion about SAS in Section 2.1.1). Again, because of the missing openness characteristic, we can neither observe the independent evolution (***C*–4.1**), the need for decentralization (***C*–4.3**), nor a significant incomplete knowledge base (***C*–4.5**) for the SAS scenarios.

In summary, SAS focus on dynamically adapt its behavior to cope with changing requirements and user needs at runtime. Therefore, distribution aspects, emergent behavior and evolutionary development are not primary characteristics for this system type.

5.4.2 CPS Characteristics

In contrast to the SAS scenarios, in the context of CPS, the openness (***C*–1**) characteristic plays a more important role as depicted in Figure 5.2. Because CPS evolved from embedded systems that are mostly a composition of linked embedded components that physically interact with an open world (cf. discussion in Section 2.1.2), also distribution aspects (***C*–1.1**) become more important. As shown in Figure 5.2, the scalability (***C*–1.2**) characteristic was not frequently found in our scenarios. We argue that scalability strongly depends of the concrete use case. For example, a CPS consisting of hundreds of sensor nodes will show different scalability characteristics as a single moving robot. Therefore, we expect an increasing demand on scalability depending on the number of subsystems as well as the overall system size.

The CPS scenarios show the same trend concerning the dynamic (***C*–1.3**, ***C*–2**, ***C*–2.1**) characteristics. As we already emphasized in Section 2.1.2, this could be explained by the physical interaction with the environment as well as reliability demands (e.g., cope with hardware/sensor failures at runtime). Therefore, CPS are often realized with self-* capabilities, which further enable adaptive behavior. Another reason for the adaptive characteristics could be that CPS are mostly composed of several embedded systems. In control engineering, embedded systems are usually designed in form of feedback loops to guarantee stable and robust behavior against external disturbances (cf. Section 2.1.2). Therefore, the control engineering design of embedded systems introduces adaptive characteristics, too.

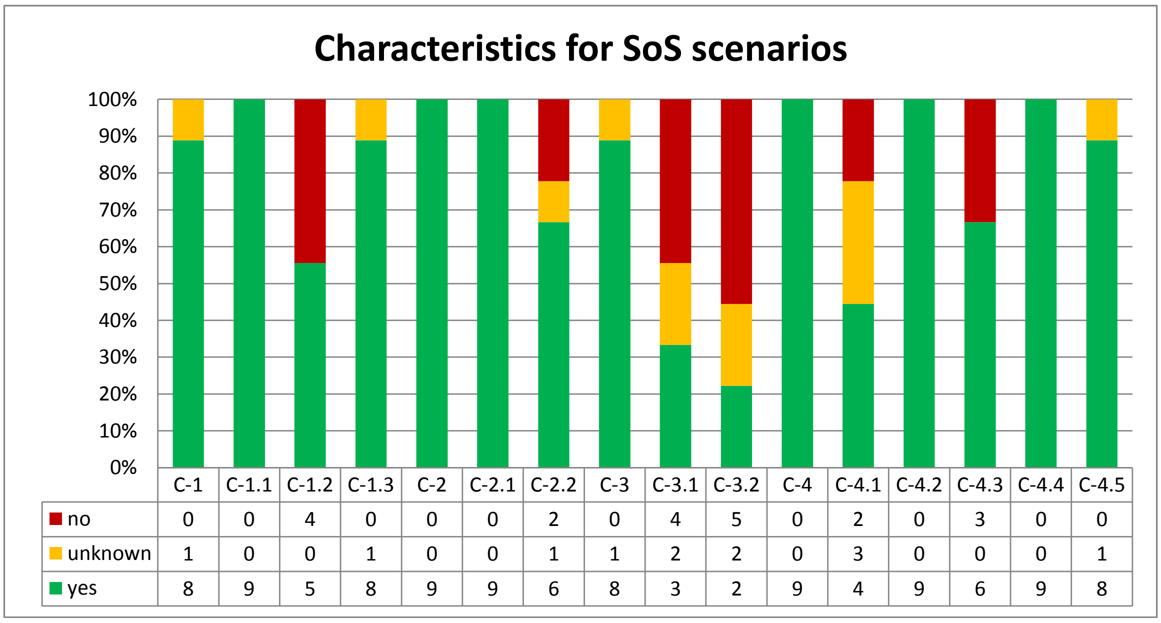


	C-1	C-1.1	C-1.2	C-1.3	C-2	C-2.1	C-2.2	C-3	C-3.1	C-3.2	C-4	C-4.1	C-4.2	C-4.3	C-4.4	C-4.5
🟥 no	1	0	4	1	1	0	2	0	2	5	0	5	0	2	0	1
🟨 unknown	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	1
🟩 yes	4	5	1	4	4	5	3	3	3	0	5	0	5	3	5	3

Figure 5.2: Characteristics in CPS scenarios

Furthermore, in contrast to the SAS scenarios, the composition of a CPS from several embedded components explains the higher counts of the collaboration (*C–3*), emergent (*C–3.1*), independence (*C–4*), autonomous (*C–4.2*), and concurrency (*C–4.4*) characteristics. In general, we could argue that the coordination effort increases with the number of composed embedded subcomponents. In addition, each embedded component usually controls or realizes one specific system functionality independent from the rest of the system, which opens further challenges concerning decentral control and emergent behavior.

Finally, we found no CPS scenario with competitive (*C–3.2*) behavior or the evolutionary development (*C–4.1*) characteristic. In our presented scenarios, most CPS have high demands on reliability or safety often under hard real-time conditions, which enforces clear coordination schemes and a tight coupling of subcomponents. Therefore, on the one hand, competitive behavior often cannot be tolerated and must be eliminated during system design. On the other hand, the tight coupling of subcomponents to the overall system explains the negative counts for the independent evolution of subsystems in Figure 5.2. In other CPS scenarios with varying system boarders due to loosely coupled subsystems or long term running components, the evolution aspect may look different.



	C-1	C-1.1	C-1.2	C-1.3	C-2	C-2.1	C-2.2	C-3	C-3.1	C-3.2	C-4	C-4.1	C-4.2	C-4.3	C-4.4	C-4.5
🟥 no	0	0	4	0	0	0	2	0	4	5	0	2	0	3	0	0
🟨 unknown	1	0	0	1	0	0	1	1	2	2	0	3	0	0	0	1
🟩 yes	8	9	5	8	9	9	6	8	3	2	9	4	9	6	9	8

Figure 5.3: Characteristics in SoS scenarios

5.4.3 SoS Characteristics

Figure 5.3 shows the characteristics for the SoS scenarios. As expected, SoS inherits the characteristics of its containing subsystems. Therefore, the depicted results for the dynamic and adaptive characteristics (***C–2***, ***C–2.1***) are straight forward as discussed for SAS and CPS. Furthermore, SoS are by definition open systems (***C–1***) that have to cope with a large number of distributed (***C–1.1***) subsystems, which further causes scalability (***C–1.2***) and flexibility (***C–1.3***) issues. In these open scenarios, coordination needs (***C–3***) are very important as reflected in Figure 5.3 .

In contrast to the SAS and CPS scenarios, the evolutionary development characteristic (***C–4.1***) of subsystems can be found more often. The increasing system size together with highly independent subsystem solutions raises further challenges concerning incomplete views and partial knowledge as represented by the ***C–4.5*** characteristic.

5.4.4 Requirements Discussion

The overall design spectrum of SoS and the continuous integration of beforehand isolated system solutions leads to a broad spectrum of systems. Depending on the system type and concrete use case scenario, we can observe different characteristics that raise several requirements concerning our collaboration modeling language. By looking at our scenarios, which are representative for several realistic use cases from different domains, and collecting the identified characteristics, we observed that every discussed characteristic from Section 3 appears.

Table 5.3 summarizes how the different SoS characteristics (cf. Section 3) cause specific requirements for our modeling language (cf. requirement discussion in Section 4). We can observe that the degree of decentralization and distribution is correlated with the size of the system. Therefore, we claim that a modeling language for collaboration should be aware of possible distributed interacting subsystems and must support the whole spectrum of centralized and decentralized control (cf. ***R–1*** and ***R–2***). Interconnecting isolated system solutions increases the overall capabilities that empowers the arising system reaching goals that cannot be fulfilled by the isolated system solutions alone. On the one hand, as we can observe it for the retrieved scenario characteristics (cf. Table 5.2), the system becomes flexible and benefits from the capabilities of the composed subsystems. On the other hand, the overall SoS has to coordinate tasks and must aggregate intermediate results. Thus, we argue that our modeling language has to cope with such coordination types (e.g., delegating tasks, cf. ***R–3***), which further needs orchestration effort to coordinate the subsystem interaction (cf. ***R–4*** to ***R–9***).

In summary, we found the complete spectrum of SoS characteristics in our scenarios from literature. As depicted in Table 5.3, the openness ***C–1*** of a system, the degree of distribution ***C–1.1***, independence ***C–4***, autonomy ***C–4.2***, and decentralization ***C–4.3*** are the main causes for our derived modeling language requirements. Furthermore, the (distributed) collaboration ***C–3*** and emergent behavior ***C–3.1*** aspect raises additional needs of modeling and understanding the interaction of system parts within the SoS. As we expect, well-established pattern, as represented by the ***R–9*** requirement in Table 5.3, would support tackling the problems caused by each system characteristics and may help by the system design solving domain specific problems.

We want to call special attention to the requirements ***R–10*** to ***R–12***, which are concerned with introducing meta concepts in our aimed collaboration modeling language. For example, ***R–10*** represents the need of introducing monitoring concepts to retrieve system interaction combinations at runtime. ***R–11*** demands analysis capabilities of modeled collaborations, e.g., on basis of the beforehand mentioned monitor capabilities, to investigate the impact of collaborating subsystems as well as to check whether global constraints are fulfilled. Finally, ***R–12*** targets the implementation and execution support of the modeling language. Therefore, we observe in Table 5.3 that the realization of the three requirements ***R–10*** to ***R–12*** beneficial support each discussed SoS characteristic.

Table 5.3: Derived requirements from SoS characteristics (✓ : characteristic directly causes requirement; – : characteristic contribute to the requirement; ✗ : characteristic do not directly cause requirement)

		Characteristic															
		C–1	*C–1.1*	*C–1.2*	*C–1.3*	*C–2*	*C–2.1*	*C–2.2*	*C–3*	*C–3.1*	*C–3.2*	*C–4*	*C–4.1*	*C–4.2*	*C–4.3*	*C–4.4*	*C–4.5*
Requirements	***R–1***	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✓	✗	✓	✓	✓	✗
	R–2	✓	✓	✗	✗	–	–	✗	✓	✓	–	✓	✗	✓	✓	✓	✗
	R–3	–	✓	✗	✓	✓	✓	✗	✓	✓	–	✗	✗	✗	✓	–	✓
	R–4	✓	✓	✓	✓	✗	✗	✓	✓	✓	–	✓	✗	✓	✓	✓	✓
	R–5	✓	✓	✓	✓	✓	✓	–	✓	✓	✓	✓	✓	✓	✓	✓	✓
	R–6	✓	–	–	✗	✗	✗	✗	✓	✓	–	✓	✓	–	✓	✓	✓
	R–7	✓	✓	✓	–	–	–	✗	✓	✓	✗	–	✓	✓	✓	✓	✓
	R–8	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗
	R–9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	R–10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	R–11	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	R–12	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

6 Related Work

In this section, we discuss the state of the art related to the modeling of collaborations in adaptive SoS. Thereby, we emphasize ideas from different modeling languages and discuss related techniques, which may influence the design of our modeling language. Furthermore, we give an overview of existing frameworks from the domain of self-adaptive systems that already cover parts of modeling feedback loops in potential distributed systems. Finally, we outline our related former work and discuss how we can benefit from that experience towards answering our stated research questions.

Inspired by [109], where the authors present a set of pattern for distributed feedback loops, the allocation of the individual adaptation activities and of the shared knowledge in the distributed system to different feedback loops as well as the coordination between these feedback loops including the knowledge exchange have to be well engineered to realize the envisioned self-adaptation. However, to the best of our knowledge, nowadays no approaches to systematically model and develop distributed self-adaptive software and the coordination of their feedback loops exist. 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. However, all of them do not explicitly specify how the distributed feedback loops are coordinated. Such specifications are essential for an engineering approach.

6.1 Modeling Languages and Techniques

In this section, we want to discuss two aspects. First, we look at modeling language approaches and show limitations related to our aimed modeling language concepts. Second, we discuss a spectrum of existing techniques that focus on specific problems to show the state of the art of existing modeling capabilities.

6.1.1 Modeling Languages and General Concepts

As we emphasized in our requirement Section 4, 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 idea of applying role modeling concepts is introduced and comprehensively discussed by Reenskaug et al. [87] and further enriched with new role modeling concepts for framework designs by Riehle et al. [88]. 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. One research challenge related to our modeling language is the mapping of the described concepts to the context of adaptive SoS by tacking dynamic role assignments and emergent behavior into account.

The Systems Modeling Language (SysML) [56] is a general-purpose modeling language specified by the OMG for modeling complex systems from a system engineering perspective. Therefore, this modeling language comprises the specification of requirements, system structure, behavior and constraints on system properties. In SysML, the focus is not specifying the software part of a system but rather defining the overall system architecture, its subcomponents, distribution and resource allocation. For example, one simple but powerful concept of SysML is the structural system specification via blocks. Blocks are modular units that can be, similar to the component concept in UML, 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 [56]. On the one hand, because SysML focus 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 system, 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 (ADLs). ADLs *"are formal languages that can be used to represent the architecture of a software-intensive system."* [33]. Such an architecture specification comprises subsystem components, structural system patterns, and interaction mechanism between components. For enabling further simulation and verification capabilities, ADLs are based on well-defined formal notations. For example, the *ArchWare* ADL introduced by Morrison et al. [81] 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, to the best of our knowledge, there is no ADL covering dynamic collaborations between feedback loops of large SoS. In general, ADLs 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 ADLs.

Table 6.1: General modeling languages cope with our derived requirements (✔ : requirement directly supported; – : requirement partially supported; ✘ : requirement not directly supported)

			Requirements											
			R–1	*R–2*	*R–3*	*R–4*	*R–5*	*R–6*	*R–7*	*R–8*	*R–9*	*R–10*	*R–11*	*R–12*
ModelingLanguage	[87, 88]	Role	✘	✘	✘	✔	✔	✘	✔	✘	✔	✘	✘	✘
	[56]	SysML	✔	–	✘	✘	✔	✘	–	✔	–	✘	✘	✘
	[33, 81]	ADL	✔	–	✘	✘	–	✘	–	–	✔	✘	✘	✘
	[57]	UML	✘	✘	✘	✔	✔	–	✔	✔	✔	✘	✘	✘
	[55]	SoaML	–	✘	✘	✔	✔	–	✔	✔	✔	✘	✘	✘

In contrast to the modeling from the system engineering perspective using SysML, the Unified Modeling Language (UML) [57] 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. [80], both modeling languages can be used describing different architectural perspectives of the overall SoS. The UML [57] 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) [55] is an UML profile that extents 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, botch 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.

Table 6.1 summarizes how the general modeling languages support our derived modeling language requirements from Section 4. As we emphasized, the role base

[1]For a comprehensive discussion about SysML and UML concepts, we refer to the corresponding specifications [56] and [57].

modeling concept focuses on separation of concerns and enables the specification of role based patterns. In addition, SysML and ADLs support the modeling of system architectures and therefore, provide concepts for modeling different control schemes (such as decentralized system and system hierarchies) and distribution aspects of the system. In contrast to SysML and ADLs, UML and SoaML focus on the software engineering perspective and thus provide additional concepts for the modeling of data (knowledge) in the system. In general, we can observe from Table 6.1 that the general modeling languages can be used for modeling a broad spectrum of systems but do not support domain specific issues as first class entities.

In MDE, Domain Specific Languages (DSLs) are used to provide specific modeling elements that are tailored to the corresponding problem domain. Therefore, DSLs 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. For example, Fleurey et al. [41] propose a DSL for the specification, simulation and execution of adaptive system. 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 consists of the beforehand mentioned variability, context constraint and rule model. The explicit modeling of feedback loops, collaborations, runtime model knowledge and a runtime verification of system constraints are not in the focus of the presented DSL.

The DSL from Fleurey et al. [41] can be seen as representative for many other DSLs, which enable the modeling of concrete problems. In most cases, DSLs have a formal background that enables simulation and verification of the modeled solutions. Even though well-established formal approaches such as π-calculus [78] or bigrahs [79] tackle structural dynamics with well-defined formalisms, to the best of our knowledge, no work exists that especially covers the problem of providing assurances for dynamic collaborations of arbitrary (or large) size as needed for SoS. Either the approaches require an initial system configuration and only support finite state systems (or systems for which an abstract finite state model of moderate size exist) [84, 100, 114] or they lack of the expressive power describing typical problems concerning the structural dynamics [8].

6.1.2 Specific Modeling Approaches

Beside modeling languages and formal approaches, there is a bunch of techniques that focuses on solving specific problems for distributed and self-adaptive systems. For example, Georgiadis et al. [46] propose a decentralized technique for the self-assembly problem. In this work, components can interact with each other via predefined static port and interface descriptions. A component manager is replicated in each component that coordinates proper interaction to finally provide a compound functionality of the overall system. Additionally, Sykes et al. [96] extends the ideas from Georgiadis et al. by replacing the manager component by a *gossip protocol* for coordination. Malek et al. [75] introduce a decentralized algorithm called *DecAp* for the redeployment problem of software components without global knowledge. All of these examples can be seen as representative solving existing problems in the context of software architectures and self-adaptive systems, but they present specific solutions for a concrete problem rather than systematically using a modeling approach, especially for the behavior of the coordination.

Looking more precisely in the self-adaptation context that emphasize the use of feedback loops, Oliveira et al. [1] 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. However, they only cover a single synchronization scheme for coordinating complete feedback loops while, in this report, our aim is a modeling language of arbitrary coordination schemes for individual feedback loop activities.

Furthermore, Weyns et al. [107, 108] have identified the need of coordination and they consider protocols, models, and channels for coordination but only as high-level concepts in their *FORMS* reference model specified in Z. As an extension of the *FORMS* model, Iftikhar et al. [63] provides an approach to formally model the MAPE activities using timed automata. This approach enables the formal verification of one feedback loop concerning given goals. Furthermore, the use of collaborations for the modeling of services [21, 90], the use of class diagrams for the structure and graph transformations for the behavior modeling [7], and a formal model of ensembles [62] have been proposed.

Similar to our mUML approach (discussed later in Section 6.3), Gezgin et al. [47] capture the dynamic architectures on the SoS level using graph transformation rules. In this approach, the overall SoS is decomposed in *system types* that can be seen as the subsystems of a SoS. Each system type has its own services and goals and can be composed over predefined interfaces (called roles). In this approach, explicit knowledge representation, runtime models and the modeling of feedback loops is not considered.

Recently, Calinescu et al. [28] presented the *DECIDE* approach that enables runtime verification of completely decentralized feedback loops on basis of probabilistic models such as probabilistic automata or continuous-time Markov chains. The DECIDE approach shows for an unmanned underwater vehicle (UUV) scenario, how each UUV adapts its local behavior according global QoS requirements.

On a higher level of abstraction, a set of architectural patterns for decentralized control in self-adaptive systems using the MAPE feedback loop approach have been presented in [109]. The authors discuss different variants of distributing individual MAPE activities and coordinate those between different feedback loops. The proposed patterns comprise typical scenarios such as master/slave, information sharing, regional planning, or hierarchical control. However, these patterns are discussed on basis of the structural distribution of black box MAPE activities neglecting the internal behavior. Furthermore, the coordination protocol between activities and the distribution of the knowledge are not considered.

All of the discussed approaches offer great ideas and concrete solutions tailoring a specific problem. Inspired by these approaches, our aim is the development of a modeling language with a formal background that supports the modeling of these problems. None of the discussed approaches focus on the development of a modeling language for adaptive SoS as discussed in Section 3 and Section 4. Therefore, they do not support the construction of dynamic collaborations as required for adaptive SoS, where systems dynamically join or leave the federation. Beside the collaboration aspect, the treating of the knowledge as runtime models and the influence of knowledge sharing activities to multiple feedback loops have to be considered.

6.2 Frameworks

Aßmann et al. [2] propose a conceptual reference architecture for self-adaptive systems with special focus on runtime models. The authors discuss several research questions that arise if multiple runtime models, where each model describes a very specific purpose of the system, must be managed in an appropriate framework. Furthermore, they emphasize the need of managing interconnected, multiple feedback loops following the MAPE-K approach as discussed in Section 2.1.1, which are typical scenarios in the CPS and SoS context.

Beside the conceptual considerations and roadmaps from Aßmann et al. [2] concerning runtime models or self-adaptive systems in general as outlined in [31, 73], there are a lot implementations of real frameworks in literature. For example, Edwards et al. [37] and Vromant et al. [104] provide architectural frameworks to support the implementation of distributed systems with hierarchical and decentralized control with focus on self-adaptation. Additionally, Edwards et al. [37] propose a set of well-defined component types that are responsible for typical adaptation activities such as monitoring of other components, analyzing the collected monitoring data, and manipulating other components to enforce adaptation. These component types are executed together with application specific components and realize the communication and interaction between different subsystems within the proposed framework. This approach proposes a layered architecture design for realizing the needed reflection capabilities between components. The interaction between layers/components is specified at the design time of the system components over well-defined interfaces. The ideas from [37] are refined and lead to the *PLASMA* approach as described by Tajalli et al [98]. The adaptive layered architecture of PLASMA has an additional planning layer on top that derives altering architecture configurations of the layer below according to given high level goals.

As already introduced in our scenarios in Section 5.1, Garland et al. [45] propose the *Rainbow* architecture framework enabling self-adaption with two main goals. First, Rainbow should enable the reuse of components and second, it handles as well as provides access to the knowledge base in the adaptation engine. Therefore, Rainbow uses a layered architecture that enables an architecture-based self-adaption of the underlying system.

Baresi et al. [6] introduce a self-adaptive middleware for smart spaces called *SeSaMe* that can handle highly dynamic, distributed large scale systems. The SeSaMe framework already has the notion of roles and enable the structurally decomposition of the system in hierarchical organized groups. A management layer in the middleware coordinates and supervises the groups in the system as well as supports group formation, self-configuration, and self-healing capabilities.

Frey et al. [44] introduce a goal driven control architecture approach together with three general patterns in the context of smart micro-grids. An interesting key aspect of this architecture is the resolution of conflicting goals concerning the local energy production and consumption of system participants.

The *DEECo* component model and its corresponding realized framework as described by Bures et al. [23] supports the specification of encapsulated components that can dynamically group each other in so-called ensembles. Thereby, each component can be a member to multiple ensembles. The runtime framework takes care of the knowledge distribution over predefined interfaces, the coordination between components and schedules the component execution.

From the multi-agent systems domain, Cossentino et al. [34] introduce an agent-oriented software engineering process called *ASPECS*. The design process especially focuses on open and dynamic systems, where the overall system is decomposed in organizations at design time. Within the organizations, agents behave according to a role and interaction specification, which is supervised by the execution framework. The ASPECS approach describes different engineering activities for the system requirements, analysis, design, implementation and deployment phase and thus supports a complete software development process in the context of agent-based systems.

Rajhans et al. [86] propose an architecture framework that is able to integrate multiple views and development models for CPS. The framework is able to integrate a broad spectrum of model types ranging from architectural descriptions to hardware design or control models. Furthermore, following the described design approach, the framework provides heterogeneous verification capabilities for the structure and behavior of the designed CPS. The integration of multiple models (e.g., partial runtime models) is very important for adaptive SoS, too. Therefore, we could benefit from the ideas of [86] for the design of our collaboration modeling language, where we have to integrate partial views and heterogeneous subsystems as well.

Almost all frameworks hide the collaboration aspects within the framework architecture. Therefore, the interaction of the subsystems is hidden in the specific implementation details of the frameworks, which increases the problem of predicting or understanding emergent behavior capabilities in an adaptive SoS [99]. However, for the design our aimed modeling language, we could benefit from the existing frameworks and bring ideas of explicit knowledge representation, local and global goal handling, multi-level modeling, runtime models, and collaboration as first class entities together. On the one hand, because of the different purpose of a framework and a modeling language, we cannot directly compare both with respect to our derived modeling language requirements as discussed in Section 4. But on the other hand, the modeling language should support the concepts from the frameworks, which has to be shown by remodeling the corresponding scenarios in future work (see further discussion in Section 7).

6.3 Former Own Work

In our own mechatronic UML (mUML) approach [25, 61] for the model-driven development of self-optimizing embedded real-time systems, we already support collaborations of self-optimizing autonomous systems in a rigorous manner by means of role protocols. Furthermore, for mUML and its collaboration concepts an overall assurance scheme has been presented in [52]. It combines a modular verification approach [50] for the component hierarchies of the autonomous systems, the compositional verification [53] of ad hoc real-time collaborations between the autonomous systems, and a fully automatic checker for inductive invariants of graph transformation system rules [9] describing the possible changes of the dynamic architecture at the SoS level. Additional work on assurances for mUML employs a multi-agent system view on a SoS to study how commitments between the collaborating systems can be modeled and analyzed [51]. Moreover, an extension for the invariant checker to cover real-time behavior has been developed in [11] and a code generation scheme in [26]. This scheme, which guarantees by construction that the timing properties of the model are satisfied by the code, ensures that the assurance results obtained for the models are also valid for the derived implementation. Therefore, the mUML approach provides assurances for systems that combine self-adaptive autonomous systems similar to a SoS. However, we still have no solution for collaborations with a dynamic number of roles, support for runtime models, and the independent evolution of the autonomous systems as well as collaborations. Moreover, in contrast to the challenges of adaptive SoS as discussed in Section 3 and Section 4, mUML provides no solution for collaborations with structural dynamics of the roles, is restricted to homogeneous systems (i.e., systems that evolve jointly and similarly and that have complete knowledge about each other), and does not support the runtime exchange of complex knowledge. Additionally, the self-adaptation is limited to pre-planned reconfigurations in hierarchical architectures. The mUML approach [25] supports the interaction of the autonomous systems by an extension of UML collaborations [53] and components for hybrid real-time behavior extending UML state machines [25].

Scenarios that require the covering of ad-hoc formation of collaborations between mechatronic or cyber-physical systems (e.g., vehicles that form convoys) or other forms of structural dynamism are captured by graph transformation systems [9, 11, 48]. In addition, first ideas for the exchange of models at runtime have been developed in the context of [24].

We show an approach for model-based architectural online reconfiguration [12] in the context of the *AUTOSAR* framework. In this approach, we present a modeling technique that captures all possible configuration of the system at design time. Furthermore, we realize the configurations within a standard toolchain from the automotive domain. This approach is limited with respect to flexible adaptations at runtime. Because of the used AUTOSAR framework, all system configurations must be planned at design time. On the one hand, this allows a resource effective implementation as requested for hard real-time automotive systems. On the other hand, this approach does not cover runtime model representations or dynamic collaborations.

In the context of service-oriented SoS, we present a formal modeling approach called *rigSoaML* that extends SoaML for the specification of evolving SoS architectures in [10]. In contrast to SoaML, rigSoaML allows the modeling of structural SoS dynamics as well as evolution of the system. The overall SoS can be decomposed in *Service Roles* that can interact with each other by means of building service contracts. Roles and service contracts are described by a set of graph transformation rules that are used to verify the overall system structure against given SoS constraints. Although this approach introduces scalable formal verification techniques

Table 6.2: Overview of own former modeling approaches supporting derived requirements (✔ : requirement directly supported; – : requirement partially supported; ✘ : requirement not directly supported)

			Requirements											
			R–1	*R–2*	*R–3*	*R–4*	*R–5*	*R–6*	*R–7*	*R–8*	*R–9*	*R–10*	*R–11*	*R–12*
Approach	[61]	mUML	✘	✘	✔	✔	✔	✘	–	✔	–	✘	–	–
	[58]	feedback loop modeling	✘	✘	–	✔	–	✘	–	✘	✘	✘	✘	✘
	[10]	rigSoaML	–	✘	–	✔	–	✘	✘	✘	✘	✘	✔	✘
	[103]	EUREMA	✘	–	✔	✘	✘	✔	✘	✘	–	✘	✘	–

on SoS system types, it does not take the notion of runtime models or feedback loops into account.

Our Executable Runtime Megamodels (EUREMA) approach [103] for the model-driven engineering of self-adaptive systems supports – in contrast to mUML – the flexible specification of self-adaptation by employing abstract runtime models (cf. [16]) of the context and the system itself. The self-adaptation behavior can be specified by rules operating on the runtime model abstractions. However, EUREMA is so far limited to centralized and non-distributed systems and does not address collaborations and the distribution aspect of feedback loops in adaptive SoS. Moreover, our former approaches addressing the modeling of feedback loops in general do not support the specification of the coordination among several feedback loops. For instance, our earlier work [58] models feedback loops as black boxes and only highlights the existence of interferences among feedback loops but without specifying them by means of coordination.

Table 6.2 subsumes our former work concerning our four main approaches that are mUML, the explicit feedback loop modeling, rigSoaML and EUREMA and shows how these approaches support our modeling language requirements. The idea of explicitly modeling feedback loops was initiated by our former work [58]. Therefore, this approach supports basic concepts for the description of feedback loops in form of black boxes and basic dependencies between them. The EUREMA modeling language [103] extends this work by introducing runtime models as first class modeling elements and a fine-grained modeling of feedback loop activities. Furthermore, EUREMA enables the specification of the control flow between adaptation activities as well as their access to the existing runtime models. rigSoaML focus on the verification of service oriented SoS. Therefore, it abstracts from concrete feedback loops and describe the overall SoS behavior with graph transformation rules. The verification is done on the specified rules to ensure global SoS properties. As a consequence, rigSoaML does not consider concrete feedback loops and their distribution as well as interaction. The mUML approach considers distributed systems, which are CPS and mechatronic systems. Therefore, it considers many aspects and modeling techniques for the tight interaction of physical and mechanical parts with the software part that controls the physical entities. Because of hard real-time constraints, safety issues and other resource restrictions in the embedded domain, mUML lacks in the specification of dynamic adaptation aspects and collaborations.

However, our aimed collaboration modeling language for adaptive SoS has to extend the ideas from EUREMA by considering distributed and decentralized systems but adopting the concepts of explicit knowledge representation and adaptation activity modeling. Additionally, the modeling language can benefit from the existing verification techniques as described in mUML and rigSoaML. Here, an integration of these formal concepts for enabling runtime verification of the system as well as its collaborating subsystems would significantly extend our former work. Our aimed modeling language can benefit from the mUML modeling techniques specifying CPS and should further extend these ideas by adding dynamic adaptation and collaboration aspects.

In the next section, we subsume this report and discuss concrete next steps for realizing the modeling language.

7 Conclusion and Future Work

In this report, we introduced five research questions concerning the modeling of collaboration in adaptive SoS in Section 1. The main idea is the explicitly modeling of interacting system parts for enabling further analysis and verification. Furthermore, we want to help understanding the impact of collaborating systems beyond own system borders for ensuring correct interaction and knowledge transfer.

As a prerequisite, we discussed several system types, terms and techniques in Section 2 by retrieving the state of the art literature and consolidate different opinions in well-defined definitions. On basis of this common understanding, we derived typical characteristics for SoS from literature in Section 3 to get an impression of the SoS nature and to identify important aspects for our modeling language. Furthermore, we derived a set of desirable modeling language requirements from the beforehand discussed characteristics in Section 4. Beside the discussion about terms, characteristics, and requirements, we introduce a set of real scenarios and application examples from literature and discuss these scenarios against the found SoS characteristics in Section 5. The related work discussion in Section 6 completes the picture of the state of the art approaches and discusses our own former work in this direction. On the one hand, the discussion of real implementations, approaches and frameworks for the different system types broadens our understanding of real system behavior and the solution space for different kind of problems. On the other hand, the scenarios can be used as evaluation base that might show strengths and weaknesses of our collaboration modeling language.

In future work, we plan to use the requirement catalog and SoS characteristics for our modeling language design. Concrete next steps are, the introduction of our modeling language elements and the discussion of the underlying modeling formalism. Afterwards, we are going to model representative scenarios from literature using the discussed scenarios in Section 5 to validate the language against the derived requirements in Section 4. Beside the modeling of scenarios, we plan to expand possible verification and simulation capabilities of our models on basis of the chosen formalism. On the one hand, we want to use the capabilities of existing verification tools and techniques. On the other hand, we plan to develop a simulation framework that helps understanding as well as visualizing the modeled interaction of subsystems and the specified adaptation logic.

List of Abbreviations

SAS	Self-Adaptive System	14
MAPE	Monitor, Analyze, Plan, and Execute	14
MAPE-K	MAPE activities share a common knowledge base	14
NES	Networked Embedded System	15
CPS	Cyber-Physical System	15
NCPS	Networked Cyber-Physical System	16
SoS	System of Systems	17
ULSS	Ultra-Large-Scale Systems	17
MDE	Model-Driven Engineering	18
MDA	Model-Driven Architecture	18
OMG	Object Management Group	18
PIM	platform independent model	19
PSM	platform specific model	19
UML	Unified Modeling Language	19
EUREMA	Executable Runtime Megamodels	20
ECU	electronic control unit	37
SysML	Systems Modeling Language	50
ADL	Architecture Description Language	50
SoaML	Service-oriented architecture Modeling Language	51
DSL	Domain Specific (Modeling) Language	52
mUML	mechatronic UML	56

References

[1] F. Alvares de Oliveira, R. Sharrock, and T. Ledoux. “Synchronization of Multiple Autonomic Control Loops: Application to Cloud Computing”. In: *Proceedings of the 14th International Conference on Coordination Models and Languages*. Volume 7274. COORDINATION. Springer, 2012, pages 29–43.

[2] U. Aßmann, S. Götz, J.-M. Jézéquel, B. Morin, and M. Trapp. “A Reference Architecture and Roadmap for Models@run.time Systems”. In: *Models@run.time*. LNCS. Springer, 2014, pages 1–18.

[3] R. 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, page 5.

[4] M. Barbero, M. D. Fabro, and J. Bézivin. “Traceability and Provenance Issues in Global Model Management”. In: *Proceedings of 3rd Workshop on Traceability*. ECMDA-TW. June 2007, pages 47–55.

[5] L. Baresi, E. Di Nitto, and C. Ghezzi. “Toward Open-World Software: Issue and Challenges”. In: *Computer* 39.10 (2006), pages 36–43.

[6] L. Baresi, S. Guinea, and A. Shahzada. “SeSaMe: Towards a Semantic Self Adaptive Middleware for Smart Spaces”. In: *Engineering Multi-Agent Systems*. LNCS. Springer, 2013, pages 1–18.

[7] L. Baresi, R. Heckel, S. Thöne, and D. Varró. “Style-based Modeling and Refinement of Service-oriented Architectures”. In: *Software and Systems Modeling* 5.2 (2006), pages 187–207.

[8] J. Bauer and R. Wilhelm. “Static Analysis of Dynamic Communication Systems by Partner Abstraction”. In: *Static Analysis*. Volume 4634. LNCS. Springer, 2007, pages 249–264.

[9] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. “Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation”. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE. ACM, 2006, pages 72–81.

[10] B. Becker and H. Giese. *Modeling and Verifying Dynamic Evolving Service-Oriented Architectures*. Technical report 75. Hasso Plattner Institute at the University of Potsdam, 2013.

[11] B. Becker and H. 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, pages 203–210.

[12] B. Becker, H. Giese, S. Neumann, M. Schenck, and A. Treffer. “Model-Based Extension of AUTOSAR for Architectural Online Reconfiguration”. In: *Models in Software Engineering*. Volume 6002. LNCS. Springer, Oct. 2010, pages 83–97.

[13] N. Bencomo. “On the Use of Software Models during Software Execution”. In: *Proceedings of the ICSE Workshop on Modeling in Software Engineering*. MISE. IEEE, 2009, pages 62–67.

[14] J. Bézevin. “On the Unification Power of Models”. In: *Software & Systems Modeling* 4.2 (2005), pages 171–188.

[15] J. Bézivin, S. Gerard, P.-A. Muller, and L. Rioux. “MDA components: Challenges and Opportunities”. In: *First International Workshop on Metamodelling for MDA*. Nov. 2003, pages 23–41.

[16] G. Blair, N. Bencomo, and R. B. France. “Models@run.time”. In: *Computer* 42.10 (2009), pages 22–27.

[17] M. Bouroche, B. Hughes, and V. Cahill. “Real-time Coordination of Autonomous Vehicles”. In: *Intelligent Transportation Systems Conference*. IEEE, Sept. 2006, pages 1232–1239.

[18] B. Bouyssounouse and J. Sifakis. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*. Volume 3436. LNCS. Springer, 2005.

[19] A. W. Brown. “Model Driven Architecture: Principles and Practice”. In: *Software and Systems Modeling* 3.4 (Dec. 2004), pages 314–327.

[20] M. Broy, M. Cengarle, and E. Geisberger. “Cyber-Physical Systems: Imminent Challenges”. In: *Large-Scale Complex IT Systems. Development, Operation and Management*. Volume 7539. LNCS. Springer, 2012, pages 1–28.

[21] M. Broy, I. Krüger, and M. Meisinger. “A Formal Model of Services”. In: *ACM Transactions on Software Engineering and Methodology* 16.1 (Feb. 2007), pages 1–40.

[22] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. A. Müller, M. Pezzè, and M. Shaw. “Engineering Self-Adaptive Systems through Feedback Loops”. In: *Software Engineering for Self-Adaptive Systems*. Volume 5525. LNCS. Springer, 2009, pages 48–70.

[23] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, and F. Plasil. "DEECO: An Ensemble-based Component System". In: *Proceedings of the 16th International Symposium on Component-based Software Engineering*. CBSE. ACM, 2013, pages 81–90.

[24] S. Burmester and H. Giese. "Visual Integration of UML 2.0 and Block Diagrams for Flexible Reconfiguration in Mechatronic UML". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*. VL/HCC. IEEE, Sept. 2005, pages 109–116.

[25] S. Burmester, H. Giese, E. Münch, O. Oberschelp, F. Klein, and P. 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), pages 207–222.

[26] S. Burmester, H. Giese, and W. Schäfer. "Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code". In: *Proceedings of the European Conference on Model Driven Architecture - Foundations and Applications*. Volume 3748. LNCS. Springer, Nov. 2005, pages 25–40.

[27] S. Burmester, H. Giese, and M. 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*. MDAFA. Springer, 2005, pages 47–61.

[28] R. Calinescu, S. Gerasimou, and A. Banks. "Self-Adaptive Software with Decentralised Control Loops". In: *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering*. FASE. Springer, 2015, pages 1–15.

[29] C. Cetina, P. Giner, J. Fons, and V. Pelechano. "Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes". In: *Computer* 42.10 (2009), pages 37–43.

[30] E. Chan, P. Gilhead, P. Jelínek, and T. Robinson. "Cooperative Control of SARTRE Automated Platoon Vehicles". In: *Proceedings of the 19th ITS World Congress*. 2012, pages 22–26.

[31] B. H. Cheng et al. "Software Engineering for Self-Adaptive Systems: A Research Roadmap". In: *Software Engineering for Self-Adaptive Systems*. Volume 5525. LNCS. Springer, 2009, pages 1–26.

[32] J.-S. Choi, T. McCarthy, M. Yadav, M. Kim, C. Talcott, and E. Gressier-Soudan. "Application Patterns for Cyber-Physical Systems". In: *IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications*. CPSNA. Aug. 2013, pages 52–59.

[33] P. C. Clements. "A Survey of Architecture Description Languages". In: *Proceedings of the 8th International Workshop on Software Specification and Design*. IWSSD. IEEE, Mar. 1996, pages 16–25.

[34] M. Cossentino, N. Gaud, V. Hilaire, S. Galland, and A. Koukam. "ASPECS: An Agent-oriented Software Process for Engineering Complex Systems". In: *Autonomous Agents and Multi-Agent Systems* 20.2 (2010), pages 260–304.

[35] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. "A journey to highly dynamic, self-adaptive service-based applications". In: *Automated Software Engineering* 15.3–4 (Dec. 2008), pages 313–341.

[36] acatech (Ed.). *Cyber-Physical Systems: Driving force for innovation in mobility, health, energy and production*. Springer, 2011.

[37] G. Edwards, J. Garcia, H. Tajalli, D. Popescu, N. Medvidovic, S. Gaurav, and B. 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, pages 142–151.

[38] E. T. S. I. (ETSI). *Intelligent Transportation Systems (ITS); Vehicular Communications; Basic Set of Applications; Definitions*. Technical report V1.1.1. European Telecommunications Standards Institute, June 2009, page 81.

[39] Y. P. Fallah, C. Huang, R. Sengupta, and H. Krishnan. "Design of cooperative vehicle safety systems based on tight coupling of communication, computing and physical vehicle dynamics". In: *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*. ICCPS. ACM, 2010, pages 159–167.

[40] J.-M. 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.

[41] F. Fleurey and A. 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*. Volume 5795. MODELS. Springer, 2009, pages 606–621.

[42] R. France and B. Rumpe. "Model-driven Development of Complex Software: A Research Roadmap". In: *Future of Software Engineering*. FOSE. IEEE, 2007, pages 37–54.

[43] A. L. de Freitas Francisco and F. J. Rammig. "Fault-Tolerant Hard-Real-Time Communication of Dynamically Reconfigurable, Distributed Embedded Systems". In: *Proceedings of the 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. ISORC. IEEE, May 2005, pages 275–283.

[44] S. Frey, A. Diaconescu, D. Menga, and I. Demeure. "A Holonic Control Architecture for a Heterogeneous Multi-Objective Smart Micro-Grid". In: *Proceedings of the 7th International Conference on Self-Adaptive and Self-Organizing Systems*. SASO. IEEE, 2013, pages 21–30.

[45] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure". In: *Computer* 37.10 (2004), pages 46–54.

[46] I. Georgiadis, J. Magee, and J. Kramer. "Self-organising Software Architectures for Distributed Systems". In: *Proceedings of the First Workshop on Self-healing Systems*. WOSS. ACM, 2002, pages 33–38.

[47] T. Gezgin, C. Etzien, S. Henkler, and A. Rettberg. "Towards a Rigorous Modeling Formalism for Systems of Systems". In: *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. ISORCW. Apr. 2012, pages 204–211.

[48] H. Giese. "Modeling and Verification of Cooperative Self-adaptive Mechatronic Systems". In: *Proceedings of the 12th Monterey Conference on Reliable Systems on Unreliable Networked Platforms*. Volume 4322. LNCS. Springer, 2007, pages 258–280.

[49] H. Giese, N. Bencomo, L. Pasquale, A. Ramirez, P. Inverardi, S. Wätzoldt, and S. Clarke. "Living with Uncertainty in the Age of Runtime Models". In: *Models@run.time*. Volume 8378. LNCS. Springer, 2014, pages 47–100.

[50] H. Giese, S. Burmester, W. Schäfer, and O. 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, pages 179–188.

[51] H. Giese and F. Klein. "Systematic Verification of Multi-Agent Systems based on Rigorous Executable Specifications". In: *International Journal on Agent-Oriented Software Engineering* 1.1 (Apr. 2007), pages 28–62.

[52] H. Giese and W. Schäfer. "Model-Driven Development of Safe Self-Optimizing Mechatronic Systems with MechatronicUML". In: *Assurances for Self-Adaptive Systems*. Volume 7740. LNCS. Springer, Jan. 2013, pages 152–186.

[53] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. "Towards the Compositional Verification of Real-Time UML Designs". In: *Proceedings of the 9th European Software Engineering Conference together with the 11th International Symposium on Foundations of Software Engineering*. ESEC/FSE. ACM, Sept. 2003, pages 38–47.

[54] D. Gil de la Iglesia and D. 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, pages 83–92.

[55] O. M. 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.

[56] O. M. Group. *Systems Modeling Language (SysML)*. Version 1.3, `http://www.omg.org/spec/SysML/` visited: 19th of February 2015. 2012.

[57] O. M. 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.

[58] R. Hebig, H. Giese, and B. 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, pages 21–28.

[59] R. Hebig, A. Seibel, and H. Giese. "On the Unification of Megamodels". In: *Proceedings of the 4th International Workshop on Multi-Paradigm Modeling*. Volume 42. Electronic Communications of the EASST. 2011.

[60] T. Henzinger and J. Sifakis. "The Embedded Systems Design Challenge". In: *14th International Symposium on Formal Methods*. Volume 4085. LNCS. Springer, 2006, pages 1–15.

[61] M. Hirsch, S. Henkler, and H. 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, May 2008, pages 33–40.

[62] M. Hölzl and M. Wirsing. "Towards a System Model for Ensembles". In: *Formal Modeling: Actors, Open Systems, Biological Systems*. Volume 7000. LNCS. Springer, 2011, pages 241–261.

[63] M. U. Iftikhar and D. 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, pages 125–134.

[64] X. Jian, G. Bing-feng, Z. Xiao-ke, Y. Ke-wei, and C. 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, pages 141–147.

[65] J. O. Kephart and D. Chess. "The Vision of Autonomic Computing". In: *Computer* 36.1 (Jan. 2003), pages 41–50.

[66] N. Kilicay-Ergin and C. Dagli. "Executable Modeling for System of Systems Architecting: An Artificial Life Framework". In: *Proceedings of the 2nd Annual IEEE Systems Conference*. IEEE, Apr. 2008, pages 1–5.

[67] K.-D. Kim and P. Kumar. "Cyber-Physical Systems: A Perspective at the Centennial". In: *Proceedings of the IEEE* 100.Special Centennial Issue (May 2012), pages 1287–1308.

[68] M. Kim, M.-O. Stehr, J. Kim, and S. Ha. "An Application Framework for Loosely Coupled Networked Cyber-Physical Systems". In: *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing* (2010), pages 144–153.

[69] F. Klein and M. 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, pages 51–58.

[70] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[71] E. A. Lee. "Cyber-Physical Systems - Are Computing Foundations Adequate?" In: *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*. Oct. 2006.

[72] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. `http://leeseshia.org` visited: 12th of March 2015, 2011.

[73] R. de Lemos et al. "Software Engineering for Self-Adaptive Systems: A second Research Roadmap". In: *Software Engineering for Self-Adaptive Systems II*. Volume 7475. LNCS. Springer, Jan. 2013, pages 1–32.

[74] P. Maes. "Concepts and Experiments in Computational Reflection". In: *Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA. ACM, 1987, pages 147–155.

[75] S. Malek, M. Mikic-Rakic, and N. Medvidovic. "A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems". In: *Component Deployment*. Volume 3798. LNCS. Springer, 2005, pages 99–114.

[76] A. Marconi, A. Bucchiarone, K. Bratanis, A. Brogi, J. Camara, D. Dranidis, H. Giese, R. Kazhamiakink, R. de Lemos, C. Marquezan, and A. 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, pages 40–46.

[77] *MDA Guide revision 2.0*. `http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf` visited: December 2014. Object Management Group. June 2014.

[78] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[79] R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[80] S. Mittal and J. L. 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, pages 1140–1151.

[81] R. Morrison, G. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cîmpan, B. Warboys, B. Snowdon, and R. M. Greenwood. "Support for Evolving Software Architectures in the ArchWare ADL". In: *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture*. WICSA. IEEE, June 2004, pages 69–78.

[82] D. J. Musliner, R. P. Goldman, M. J. Pelican, and K. D. Krebsbach. "Self-Adaptive Software for Hard Real-Time Environments". In: *Intelligent Systems and their Applications, IEEE* 14.4 (July 1999), pages 23–29.

[83] L. Northrop, P. H. Feiler, R. P. Gabriel, R. Linger, T. Longstaff, R. Kazman, M. Klein, and D. Schmidt. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University, 2006.

[84] P. C. Ölveczky and J. Meseguer. "Specification and Analysis of Real-Time Systems Using Real-time Maude". In: *Proceedings on Fundamental Approaches to Software Engineering*. Volume 2984. LNCS. Spinger, 2004, pages 354–358.

[85] C. L. Ortiz, R. Vincent, and B. 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, pages 860–867.

[86] A. Rajhans, A. Bhave, I. Ruchkin, B. H. Krogh, D. Garlan, A. Platzer, and B. Schmerl. "Supporting Heterogeneity in Cyber-Physical Systems Architectures". In: *Transactions on Automatic Control* 59.12 (Dec. 2014), pages 3178–3193.

[87] T. Reenskaug, P. Wold, and O. A. Lehne. *Working With Objects - The OOram Software Engineering Method*. Prentice Hall, 1996.

[88] D. Riehle and T. 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, pages 117–133.

[89] M. Salehie and L. Tahvildari. "Self-Adaptive Software: Landscape and Research Challenges". In: *Transactions on Autonomous and Adaptive Systems* 4.2 (2009), pages 1–42.

[90] R. T. Sanders, H. N. Castejón, F. Kraemer, and R. Bræk. "Using UML 2.0 Collaborations for Compositional Service Specification". In: *Model Driven Engineering Languages and Systems*. Volume 3713. LNCS. Springer, 2005, pages 460–475.

[91] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. "Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems". In: *IEEE International Conference on Requirements Engineering*. IEEE, 2010, pages 95–103.

[92] W. Schäfer and H. Wehrheim. "The Challenges of Building Advanced Mechatronic Systems". In: *Future of Software Engineering*. FOSE. IEEE, 2007, pages 72–84.

[93] D. C. Schmidt. "Model-Driven Engineering". In: *IEEE Computer* 39.2 (Feb. 2006).

[94] R. Sengupta, S. Rezaei, S. E. Shladover, D. Cody, S. Dickey, and H. Krishnan. "Cooperative Collision Warning Systems: Concept Definition and Experimental Implementation". In: *Intelligent Transportation Systems* 11.3 (July 2007).

[95] J. A. Stankovic, I. Lee, A. Mok, and R. Rajkumar. "Opportunities and Obligations for Physical Computing Systems". In: *Computer* 38.11 (Nov. 2005), pages 23–31.

[96] D. Sykes, J. Magee, and J. 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, pages 100–109.

[97] J. Sztipanovits, G. Karsai, and T. Bapty. "Self-adaptive software for signal processing". In: *Communication* 41.5 (1998), pages 66–73.

[98] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic. "PLASMA: A Plan-based Layered Architecture for Software Model-driven Adaptation". In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ASE. ACM, 2010, pages 467–476.

[99] R. Valerdi, E. Axelband, T. Baehren, B. Boehm, D. Dorenbos, S. Jackson, A. Madni, G. Nadler, P. Robitaille, and S. Settles. "A Research Agenda for Systems of Systems Architecting". In: *International Journal of System of Systems Engineering* 1.1-2 (2008), pages 171–188.

[100] D. Varró. "Automated Formal Verification of Visual Modeling Languages by Model Checking". In: *Software and System Modeling* 3.2 (May 2004), pages 85–113.

[101] E. Vassev and M. Hinchey. "The Challenge of Developing Autonomic Systems". In: *Computer* 43.12 (2010), pages 93–96.

[102] T. Vogel and H. Giese. "Adaptation and Abstract Runtime Models". In: *Proceedings of the 5th Workshop on Software Engineering for Adaptive and Self-Managing Systems at the 32nd IEEE/ACM International Conference on Software Engineering*. SEAMS. ACM, May 2010, pages 39–48.

[103] T. Vogel and H. Giese. "Model-Driven Engineering of Self-Adaptive Software with EUREMA". In: *ACM Transactions on Autonomous and Adaptive Systems* 8.4 (Jan. 2014), 18:1–18:33.

[104] P. Vromant, D. Weyns, S. Malek, and J. 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, pages 202–207.

[105] V. S. C. C. (VSCC). *Vehicle Safety Communications Project: Task 3 Final Report: Identify Intelligent Vehicle Safety Applications Enabled by DSRC*. National Highway Traffic Safety Administration, Office of Research and Development, 2005.

[106] S. Wätzoldt and H. 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, pages 11–20.

[107] D. Weyns, S. Malek, and J. Andersson. "FORMS: Unifying Reference Model for Formal Specification of Distributed Self-adaptive Systems". In: *ACM Transactions on Autonomous and Adaptive Systems* 7.1 (May 2012), 8:1–8:61.

[108] D. Weyns, S. Malek, and J. 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, pages 84–93.

[109] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. Goeschka. "On Patterns for Decentralized Control in Self-Adaptive Systems". In: *Software Engineering for Self-Adaptive Systems II*. Volume 7475. LNCS. Springer, 2013, pages 76–107.

[110] M. 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.

[111] W. Wolf. "The Good News and the Bad News". In: *Computer* 40.11 (2007), pages 104–105.

[112] P. R. Wurman, R. D'Andrea, and M. Mountz. "Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses". In: *Proceedings of the 19th National Conference on Innovative Applications of Artificial Intelligence - Volume 2*. IAAI. AAAI, 2007, pages 1752–1759.

[113] M. Zeller and C. Prehofer. "A Multi-layered Control Approach for Self-adaptation in Automotive Embedded Systems". In: *Advances in Software Engineering* 2012.10 (Jan. 2012), pages 1–15.

[114] J. Zhang, H. J. Goldsby, and B. H. Cheng. "Modular Verification of Dynamically Adaptive Systems". In: *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*. AOSD. ACM, 2009, pages 161–172.

ISBN 978-3-86956-324-4
ISSN 1613-5652