Towards Smart Systems of Systems*

Holger Giese, Thomas Vogel, and Sebastian Wätzoldt

Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany
{Holger.Giese,Thomas.Vogel,Sebastian.Waetzoldt}@hpi.de

Abstract. Systems of Systems (SoS) have started to emerge as a consequence of the general trend toward the integration of beforehand isolated systems. To unleash the full potential, the contained systems must be able to operate as elements in open, dynamic, and deviating SoS architectures and to adapt to open and dynamic contexts while being developed, operated, evolved, and governed independently. We name the resulting advanced SoS to be *smart* as they must be self-adaptive at the level of the individual systems and self-organizing at the SoS level to cope with the emergent behavior at that level. In this paper we analyze the open challenges for the envisioned smart SoS. In addition, we discuss our ideas for tackling this vision with our SMARTSOS approach that employs open and adaptive collaborations and models at runtime. In particular, we focus on preliminary ideas for the construction and assurance of smart SoS.

1 Introduction

Systems of Systems (SoS) [1, 2] nowadays become a highly relevant challenge as the general trend can be observed that beforehand isolated systems are integrated into larger federations of systems. To unleash the full potential of such federations, SoS must be *smart* such that the contained systems are able to operate as elements in open, dynamic, and deviating SoS architectures and to adapt to open and dynamic contexts while being developed, operated, evolved, and governed independently.[1] Therefore, the resulting smart SoS must be self-adaptive at the level of the individual systems and self-organizing at the SoS level to cope with the emergent behavior at that level.

For a smart SoS holds that each of its systems has to be *independent* in the sense that it is developed, operated, evolved, and governed independently from

* This work was partially developed in the course of the project "Quantitative analysis of service-oriented real-time systems with structure dynamics" (Quantum) at the Hasso Plattner Institute at the University of Potsdam, published on its behalf, and funded by the Deutsche Forschungsgemeinschaft. See http://www.hpi.de/en/giese/projects/quantum.html

[1] Similar needs are observed for specific cases of SoS, such as *ultra-large-scale systems* [3] focusing on issues arising from the complexity of SoS or *cyber-physcial systems* [4] emphasizing the integration of the physical and cyber world.

M. Dastani and M.Sirjani (Eds.): FSEN 2015, LNCS 9392, pp. 1–29, 2015.
DOI: 10.1007/978-3-319-24644-4_1

the other systems in the SoS. Furthermore, these systems interact with each other in an open and dynamic world (cf. [5]), which causes individual systems to dynamically join or leave the SoS over time, the SoS architecture to deviate, and emergent behavior at the *SoS level*. Moreover, each independent system must adapt its behavior autonomously according to its own needs and peer systems in the SoS (*self-adaptation* [6, 7]) while considering the interplay between its own behavior, the other systems' behavior, and the required SoS-level behavior (*self-organization* [8]). The envisioned interaction between the systems involves independently developed systems and requires means for exchanging knowledge between these systems at runtime. This knowledge covers aspects of a running system itself and the context or the requirements of a running system. While the outlined rich interaction is key, the development of the systems has to scale and thus only can take into account the publically available knowledge about possible collaborations with the other systems. Finally, the systems as part of a smart SoS will evolve in order to adjust to new needs or changing regulations (cf. *software evolution* [9]). Since it seems today improbable or even impossible that all the required evolution steps can be covered automatically by a system itself (e.g., by self-adaptation or self-organization), it must still be possible that required evolution steps for each individually governed system are performed during the operation of the SoS. These challenges for engineering smart SoS are currently hardly covered by the available approaches.

The current state of the art suggests constructing SoS using an architecture perspective and services (cf. [10]) employing, for example, the *Service oriented architecture Modeling Language* (SoaML) [11] and the *Unified Modeling Language* (UML) [12] to specify the cooperation of systems by means of service contracts and 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.

The use of collaborations for the modeling of services [13,14], the use of class diagrams for the structure and graph transformations for the behavior modeling [15,16], and a formal model of ensembles [17] have been proposed. However, none of these approaches supports the construction of dynamic collaborations as required for smart SoS where systems dynamically join or leave the federation. And even though well-established formal approaches such as π-calculus [18] or bigrahs [19] tackle such structural dynamics, to the best of our knowledge no work exists that especially covers the problem of providing assurances for dynamic collaborations of arbitrary size. 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) [15,20–24] or they lack the expressive power to describe typical problems concerning the structural dynamics [25,26].

In our own Mechatronic UML approach (mUML) [27] 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. For mUML and its collaboration concepts an overall

assurance scheme has been presented in [28]: it combines a modular verification approach [29] for the component hierarchies of the autonomous systems, the compositional verification [30] of ad hoc real-time collaborations between the autonomous systems, and a fully automatic checker for inductive invariants of graph transformation system rules [31] 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 an SoS to study how commitments between the collaborating systems can be modeled and analyzed [32]. Therefore, with MUML an approach exists that provides assurances for systems that combine self-adaptive autonomous systems similar to an SoS. However, in contrast to the challenges of smart SoS, which are discussed in the next section, 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. Moreover, the self-adaptation is limited to pre-planned reconfigurations in hierarchical architectures.

Our Executable Runtime Megamodels approach (EUREMA) [33] for the model-driven engineering of self-adaptive systems supports – in contrast to MUML – the flexible specification of self-adaptation by allowing us to employ abstract runtime models (cf. [34]) of the context and the system itself such that the self-adaptation behavior can be specified by rules operating on such abstractions. However, EUREMA is so far limited to centralized and non-distributed systems and does not address collaborations or the self-organizing SoS level.

In this paper, we will first analyze open challenges for engineering smart SoS. We will then discuss our ***S**oftware with **M**odels **a**t **R**un**t**ime for **S**ystems **o**f **S**ystems* (SMARTSOS) vision that employs collaborations and generic models at runtime for trustworthy self-organization and evolution of the systems at the SoS level and self-adaptation within the systems while taking the independent development, operation, management, and evolution of these systems into account. We will particularly outline the formal foundations underlying SMARTSOS based on graph transformation systems [35,36] which cover in principle the identified challenges for the construction and assurance of smart SoS.

The rest of the paper is structured as follows: In Section 2, we discuss open challenges for smart SoS. Afterwards, our SMARTSOS vision is outlined in Section 3. Then, the concepts for constructing smart SoS with collaborations, components, and runtime models are outlined in Section 4. Afterwards, the results that enable the assurance for the smart SoS are presented in Section 5. The paper closes with a discussion of these results in Section 6 and provides in Section 7 some concluding remarks and an outlook on future work.

2 Challenges

SoS as a composition of systems that are operationally and managerially independent from each other [1] is characterized by uncoordinated evolution steps and geographic distribution of these systems (cf. [37]). Additionally, dynamic

Table 1. Summary of the Identified Challenges.

Construction/Assurance of Self-Adaptation (C1/A1)
Construction/Assurance of SoS-Level Interactions for Self-Organization (C2/A2)
Construction/Assurance of SoS-Level Structural Dynamics (C3/A3)
Construction/Assurance of SoS-Level Runtime Knowledge Exchange (C4/A4)
Construction/Assurance of Evolution of Smart SoS (C5/A5)
Scalable Construction/Assurance of Smart SoS (C6/A6)
Construction/Assurance of Smart SoS with Restricted Knowledge (C7/A7)

configuration capabilities, resilience, the ability to dynamically adapt and absorb deviations in the SoS structure, and the ability to deal with emergent behavior in context of self-organization that goes beyond developing contractual descriptions are crucial [2] for smart SoS. These issues are challenging since each system as part of a smart SoS is *independent* in the sense that it is developed, operated, evolved, and governed independently from the other systems in the same SoS. Furthermore, the systems interact with each other in an open and dynamic world (cf. [5]), which causes individual systems to dynamically join or leave the smart SoS over time, the SoS architecture to deviate, and emergent behavior at the *SoS level*. Consequently, the SoS-level architecture and behavior are not controlled by a single, centralized authority.

As example in this paper we consider a large-scale transport system where different organizations operate fleets of autonomously driving shuttles that share a track system. This constitutes an SoS since the shuttles are operated, managed, and evolved by different authorities, they dynamically interact with each other (e.g., to build convoys), and they adapt to their own and the other shuttle's states and behavior (e.g., to decrease the speed when the shuttle's own battery level is low or when the speed of the shuttle running ahead decreases).[2]

Engineering such smart SoS imposes several challenges that we outline in Table 1. All of them aim for means for the construction and assurance of smart SoS and its individual systems, which must take the operational and managerial independence of the individual systems and the emergent behavior at the SoS level into account.

The first challenge considers the *Construction/Assurance of Self-Adaptation (C1/A1)* of the individual systems as each system within a smart SoS must adapt its behavior according to its own needs and the behavior of other systems as well as the emergent SoS-level behavior (cf. [38, 39]). For instance, a shuttle adapts its speed to its battery level, to the speed of the shuttle running ahead, or to an agreement established by all shuttles in a convoy. Such changes of the behavior must be systematically constructed and assured to enable trustworthy operation and in particular self-adaptation [6, 7].

Due to the operational and managerial independence of the systems, the *Construction/Assurance of SoS-Level Interactions for Self-Organization (C2/A2)* challenge captures that the interactions among these systems must be self-organizing (cf. [8]) to achieve the SoS-level goals. For example, shuttles from different organization must interact to avoid collisions or they may even

[2] See `http://www.railcab.de` for an example with the outlined characteristics.

cooperate to build convoys in order to save energy. Such interplays of autonomous shuttles must be systematically constructed and assured to provide confidence for satisfying SoS-level goals (e.g., reliable and safe rail traffic).

Furthermore, since the operational context changes or the systems dynamically join or leave the SoS such that the SoS-level architecture dynamically changes, the challenge of the *Construction/Assurance of SoS-Level Structural Dynamics (C3/A3)* must be supported. For instance, autonomous shuttles may join or leave a convoy and the other shuttles already part of the convoy must account for it. Such dynamics must be constructed and assured to guarantee certain SoS-level behavior resulting from the interactions.

Additionally to the structural dynamics, the *Construction/Assurance of SoS-Level Runtime Knowledge Exchange (C4/A4)* is required since no system in the SoS typically has *all* the knowledge needed to achieve the SoS-level goals at runtime. This calls for exchanging knowledge between interacting systems. This knowledge refers to the current state, context, and requirements of individual systems. For instance, shuttles establishing a convoy may exchange their driving modes to agree on a common mode for the whole convoy. Such a knowledge exchange must be constructed and assured when engineering interactions to gain confidence that the SoS-level goals can be achieved.

In parallel to the self-adaptive and self-organizing behavior of systems in the SoS, the systems evolve in an uncoordinated way as they are managed independently from each other by different organizations. Evolution is caused by the permanent need to change the software in response to changing requirements of the stakeholders, no longer valid assumptions, or changing regulations the software has to adhere to (cf. *software evolution* [9]). With evolution, we refer to introducing new or removing existing types of systems or interactions (and therefore, also behavior) from the SoS. For example, new shuttle versions that support new cooperation mechanisms are integrated into the SoS and they must not interfere with the already existing versions. To handle such radical changes in a trustworthy manner and to support the long-term existence of the SoS, the *Construction/Assurance of Evolution of Smart SoS (C5/A5)* is a main challenge.

In general, engineering smart SoS is challenging due to the ultra-large scale and complexity of such systems and due to the different authorities managing such systems. Therefore, the *Scalable Construction/Assurance of Smart SoS (C6/A6)* and the *Construction/Assurance of Smart SoS with Restricted Knowledge (C7/A7)* are also important aspects for the engineering. The first aspect is motivated by the fact that it is not feasible to construct the whole SoS upfront before its deployment, or to analyze *all* possible SoS configurations or architectures that grow exponentially with the number of participating systems. For instance, any number of shuttles of arbitrary types may run on the track system or may cooperate in a convoy. Thus, the construction and assurances for SoS must scale with the size of the SoS. The second aspect refers to the construction and assurances for smart SoS, which must work despite the restricted knowledge that participants of the SoS have. An organization responsible for an individual system in the SoS or the system itself might have no global view of the SoS and

no concrete information about the other participants and possible interactions among participants. Nevertheless, assurances for each system and the SoS must be provided to enable trustworthy operations. For example, when constructing and assuring new shuttle versions supporting a certain interaction, details about other organizations' shuttle versions that are potential cooperators for the interaction might not be available. However, the construction and assurance must cope with the limited knowledge available.

These open challenges, each with a construction and assurance dimension as summarized in Table 1, reveal the difficulty of engineering and ruling smart SoS due to the complexity, dynamics, emergence, and decentralized management and governance.

3 SMARTSOS

In our vision SMARTSOS, we suggest combining the benefits of MUML and EUREMA to tackle the challenges for smart SoS. However, we do not suggest simply integrating the ideas of both approaches. Instead we developed a radically different and more abstract perspective on the SoS-level interactions to overcome the limitations of the state-of-the-art and former approaches and to master the complexity of smart SoS. This novel perspective is based on the combination of runtime models and collaborations.

3.1 Runtime Models

To realize the challenge of the Construction/Assurance of SoS-Level Runtime Knowledge Exchange (C4/A4), SMARTSOS employs *models at runtime* [34] that suggest following model-driven engineering principles to engineer abstract runtime representations of running systems or their contexts and requirements. Such models are said to be *causally connected* to the running system, which means that changes in the system are reflected in the model and vice versa. Thus, "change agents (e.g., software maintainers, software-based agents) use [abstract] runtime models to modify executing software in a controlled manner" [40, p. 39] rather than directly adapting the running software at the code level.

With EUREMA we have extended this perspective by providing runtime models at different levels of abstraction [41] and by specifying the adaptation itself, that is, software-based agents, using runtime models [33]. The latter aspect leverages the flexibility of runtime models for managing the change agents at runtime in addition to the running systems. In SMARTSOS, we go another step further and suggest using *generic* in contrast to highly specific and optimized runtime models, which eases interoperability and evolution of systems in an SoS, as it excludes individual optimization and specific solution for the runtime models.

For an individual system in a smart SoS, such generic runtime models may capture the system's state, context, requirements, and adaptation logic. These models may be used locally for self-adaptation and assurances that each system

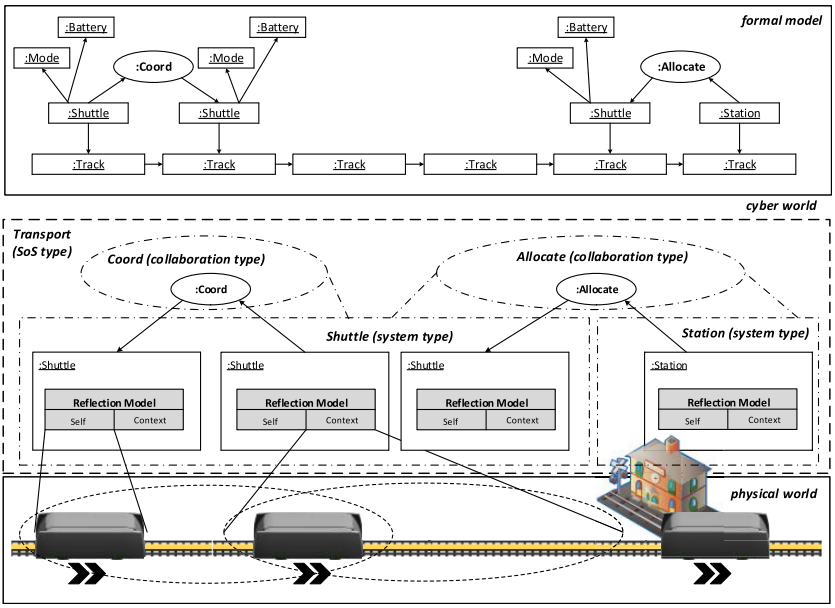

Fig. 1. Local and shared runtime models of a complex SoS architecture.

fulfills its requirements. As discussed in the following, they may also be used for collaborations among systems leading to the self-organization of the SoS, which requires the exchange of runtime knowledge.

The basic idea to integrate runtime models and collaborations for the construction of smart SoS is depicted in Fig. 1. At first, each running system living in the cyber world may have a view on its physical context and its own state in the cyber and physical world by means of runtime models of the *context* and the *self*. In general, we consider runtime descriptions reflecting the running system and its context as *Reflection Models*. Particularly, *System Models* (Self in Fig. 1) reflect about architectural and behavioral key aspects of the system and they are causally connected to the system. *Context Models* (Context in Fig. 1) describe the environmental situation of a system (cf. [42, 43]). In our example, models of the Context and Self are depicted in the individual Shuttle systems in Fig. 1. This supports designing the self-adaptation of the shuttles as MAPE-K feedback loops (**M**onitor/**A**nalyze/**P**lan/**E**xecute-**K**nowledge) [44] while the knowledge part is implemented by the runtime models. Such feedback loops are realized by analyze and plan activities that operate on the basis of the runtime models while linking the runtime models to the system and context is realized by the monitor and execute activities. The Self and Context runtime models can refer to the local state of a shuttle (e.g., the Mode and Battery status) as well as the available information about the context (e.g., whether there is another shuttle driving on the tracks ahead of the shuttle). This context does not only consist of the physical context such as the shuttle's position, the topology of the

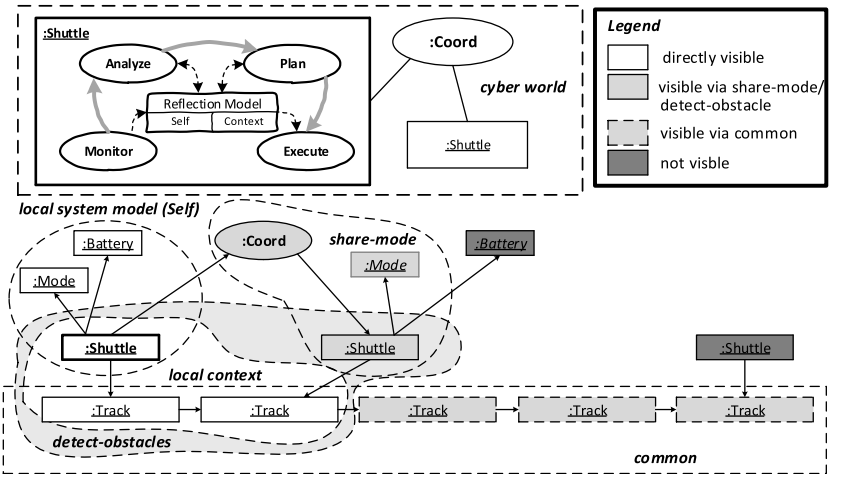

Fig. 2. Visibility of local and shared runtime models.

track system, and the positions of other shuttles nearby, but additionally the context in the cyber world. This cyber-world context covers, for instance, the established collaboration instances and context shared with other roles of these collaboration instances.

As depicted in Fig. 1, these runtime models of the context and self constitute an overall local runtime model, that is, the Reflection Model, for each :Shuttle system. As described in Fig. 2 in more detail, each system does not only operate on information about its own local context and itself, but can also get access to the information stored in runtime models of other systems concerning their context or themselves.[3] In our example, the left most Shuttle system has also access to information in form of runtime models in quite different ways. The directly visible elements are the information available local to the shuttle by means of runtime models (white with solid frame). Additional information like the position of the shuttle in front of it, which is encoded by the on edge between that shuttle and its current track, is accessible for shuttles that knows the Coord collaboration type (gray with dashed frame). Finally, information about the mode of the shuttle system in front of it is accessible for a shuttle, if it is connected to this via a Coord collaboration instance (gray with solid frame).

Likewise to EUREMA, the behavior of the systems in reaction to particular situations can now be directly described based on such overall local runtime models. This is illustrated at the top of the left-hand side of Fig. 2 showing the feedback loop (i.e., the monitor, analyze, plan, and execute activities) of a shuttle realizing the self-adaptation and therefore, addressing the Construction

[3] In the case of heterogeneous local runtime models, efficient incremental model synchronization techniques such as triple graph grammars [45] can be employed. Such techniques realize required translation steps between runtime models that are specified in different modeling languages but that capture similar content. We already applied them to create and maintain multiple runtime models of a system in [41].

of Self-Adaptation (C1) challenge. In the following, we will elaborate the use of such generic runtime models in the context of collaborations. In general, such models as employed by SMARTSOS provide an idealization of the systems and contexts, which is discussed in Section 4.3.

3.2 Collaborations

To approach the challenges of the Construction/Assurance of SoS-Level Interactions for Self-Organization (C2/A2) and the Construction/Assurance of SoS-Level Structural Dynamics (C3/A3) while taking aspects of the Scalable Construction/Assurance of Smart SoS (C6/A6) and the Construction/Assurance of Smart SoS with Restricted Knowledge (C7/A7) into account, *SoaML* and *UML* provide basic concepts of modeling collaborations. They support specifying abstract collaboration types and corresponding roles. The interaction of roles is defined by sequence or activity diagrams and *UML* interface descriptions (in form of class diagrams). Role behavior may be also covered by protocol state machines. The mUML approach goes beyond the ideas of *SoaML* and *UML* by describing the possible interaction always via real-time variant of state machines for each role and the communication medium. Due to the well-defined semantics for all employed formalism such as the real-time variant of state machines, mUML enables the basic verification of the interactions through model checking.

SMARTSOS supports a more flexible concept for collaborations compared to mUML and *SoaML*/*UML*. In SMARTSOS a richer language to specify the collaborations is provided, which also covers the exchange of complex information as well as specifying structural dynamics covering, for example, how systems can join the collaboration, leave the collaboration, or how the structure of the collaboration may change at runtime. In order to ensure interoperability and achieve trustworthy behavior at the SoS level, we furthermore need also more sophisticated analysis capabilities, for example, to investigate the impact from one system part to another through collaborations.

The basic idea of separating the required interactions into separated collaborations is depicted in Fig. 1. The autonomous systems of the SoS can connect with each other as specified by the collaboration types and can establish collaboration instances to cooperate as needed. For example, the two left most Shuttle instances in Fig. 1 are linked by a Coord collaboration instance to build a temporary convoy and the right most Shuttle instance is linked to a Station instance by an Allocate collaboration instance.

It has to be noted that different collaboration types and their involved views on the physical or cyber world may not be disjoint. In such a case, the construction and assurance for the collaboration types that share some of their elements of the physical or cyber world have to take such overlapping into account. The required concepts for abstract shared collaborations types that includes runtime models with overlapping entities are discussed in Section 4.3.

The elements of the Coord collaboration type are defined in the class diagram depicted on the left hand side of Fig. 3. The Coord collaboration element as well as the Shuttle role with its Mode and Battery status are introduced using the

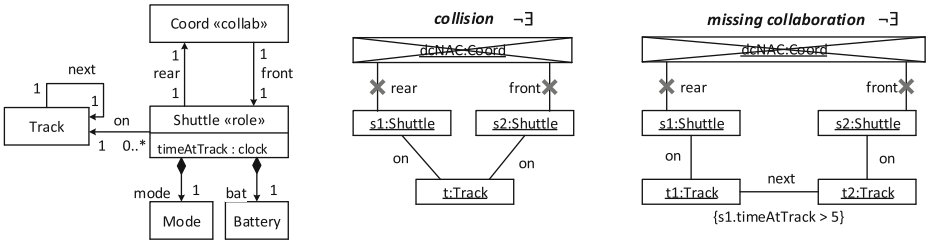

Fig. 3. Class diagram of the collaboration type Coord (left); properties of the collaboration type Coord as forbidden SPs (middle and right).

stereotypes ≪collab≫ for collaborations and ≪role≫ for roles. The class diagram also defines the track topology, that is, multiple Track elements connected with each other by the next relationship, as well as the positioning of the Shuttles on the tracks by the on relationship.

The class diagram of a collaboration type implicitly specifies in form of all valid object configurations for the class diagram the possible states the collaboration may be in. Later on, we define rules that refer to such object configurations to capture the system behavior. We formally define these object configurations as attributed graphs in Section 4.

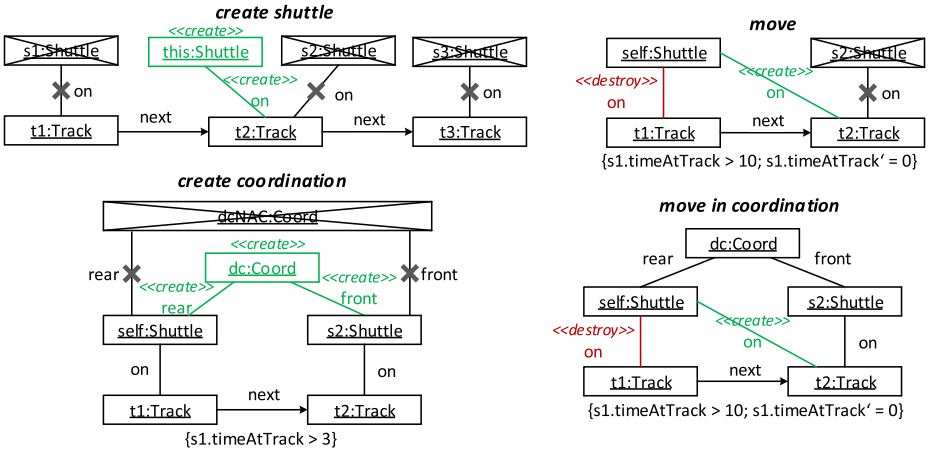

Fig. 4. Behavior rules for the Shuttle role of the Coord collaboration as SPs.

To capture the laws that should hold for the different collaboration types, we specify *behavior rules* with Story Patterns (SPs) [45] that define the permitted and mandatory behavior of each role, *read rules* with SPs having no side effects that define the visibility of shared runtime models, and the *properties* the collaboration has to ensure. These properties are described by SPs without side effects in the case of simple state properties and with Timed Story Sequence Diagrams (TSSDs) [46] in the case of sequence properties. A major duty for the SoS-level assurance is to ensure that the properties are the guaranteed outcome of the roles' behavior.

Fig. 4 depicts the *behavior rules*, that is, the SPs defining the permitted and mandatory behavior of each role of the Coord collaboration (we refer to [47] for the complete set of behavior rules). The SP called move in coordination on the lower right-hand side of Fig. 4, for example, describes that after the building of a platoon encoded by the existence of the Coord instance, the rear shuttle instance can move with a reduced distance behind the front shuttle instance and therefore, both shuttle instances may even be positioned on the same Track instance. The instance name self used in a rule determines the role of the collaboration, for which this rule is intended, in this example, for the shuttle role (cf. self:Shuttle element). Furthermore, the create coordination SP defines that shuttles must create a collaboration if they are on neighboring tracks and do not yet collaborate (i.e., there is no Coord instance yet).

A simple SP denotes two graphs at once. The first one is the left-hand-side graph L that you try to find in the current instance situation encoded in the graph G and that consists of all unmarked elements and those marked with ≪destroy≫. The second one is right-hand-side graph R that consists of all unmarked elements and those marked with ≪create≫. If L can be matched in G, the SP rule can be applied. A rule application on G results in the replacement of the match of L in G by R. In the case of complex SPs that have a negative application condition (NAC) that defines a graph L' resulting from extending L by all the crossed-out elements. Then, besides finding the left-hand-side graph L in G there must exist *no* match for the NAC L' in G that extends that match for L. Otherwise, the rule cannot be applied.

On the upper right-hand side of Fig. 4, the move SP specifies using such a NAC that shuttles can move forward along the track over time if there is no other shuttle on the next track. Additionally, the SP defines a temporal condition that a shuttle must stay on a track at least ten time units before it can move to the next track. This temporal condition ensures that the move SP can only be applied with respect to realistic physical movement conditions of the shuttles. If the shuttle moves, the on reference is created on the t2 Track and the clock attribute timeAtTrack of the shuttle is reseted to zero. As a consequence, the shuttle has to stay on the next track again at least for ten time units before the move SP can be applied again.

Fig. 5 shows an example of an application of a graph transformation rule. If we consider the instance situation (start graph) on the left-hand side and apply the move SP rule from Figure 4, first a match for the move SP has to be found.

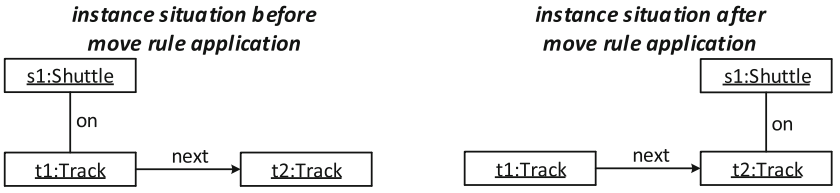

Fig. 5. Application of the move SP from Figure 4 on an exemplary instance situation.

Thereby, we can find a match, where the shuttle instance *self* from Figure 4 is matched to the *s1* shuttle instance on the left side in Figure 5. Because there is no other shuttle on the next track t2, the move SP can be applied for the found match with the consequence that the on link from shuttle s1 to track t1 is deleted and a new on link from shuttle s1 to track t2 is created. Thus, we obtain the instance situation (result graph) as depicted on the right-hand side on Figure 5.

The *properties* the collaboration type Coord should guarantee are depicted in Fig. 3. We have the forbidden situations collision and missing collaboration that must not happen. Hence, these properties are marked with $\neg\exists$. For example, the collision SP in the middle of Fig. 3 shows the situation where two shuttles that are not collaborating with each other are on the same track, that is, these two shuttles may collide. Furthermore, the missing collaboration SP on the right-hand side of Fig. 3 reflects the faulty situation where two shuttles are on neighboring tracks but they do not collaborate with each other. Both situations have to be excluded to ensure a proper operation of the collaboration. For both instance situations in Figure 5 must hold that all specified properties are fulfilled (that the forbidden collision and missing collaboration SP cannot be matched).

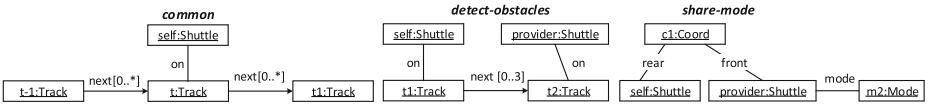

Fig. 6. Read rules for the Shuttle role of the Coord collaboration type as SPs.

The *read rules* for the collaboration type Coord are depicted in Fig. 6 using SPs and so called *path expressions* that allow us to describe the fraction of the runtime models that can be accessed in a more compact form than standard SPs without such path expressions. In general, read rules on the one hand describe what the roles can access (read) but also imply on the other hand that the other roles somehow have to provide the related information. In any case, the self instance name in a read rule that must always be present determines for which role this rule specifies access to other runtime models. The optional provider instance name if present in a rule indicates whether a related role is in charge of providing that information.

The first read rule, called common, describes that it is assumed that a shuttle (marked with the self instance name) has access to the complete track topology because the path expression next$[0..*]$ denotes any finite path between two Tracks with arbitrary many omitted and thus not visible nodes in between. Looking at the class diagram in Fig. 3, it reveals that the omitted nodes must always be of type Track and thus, the path expression represents all sequences of Tracks connected by next edges. As this read rule does not require any instance of the Coord collaboration, it denotes that each shuttle knows the track topology on its own (a possible implementation would be that each shuttle has a map of the track topology).

The read rule detect-obstacle for the Shuttle role (cf. self instance name) employs the path expression next[0..3] to denote a path between two Tracks with 0 to 3 not visible Tracks in between. The provider instance name for the other Shuttle role indicates that this role must provide this fragments of its runtime model. As this read rule does not require any instance of the Coord collaboration, it defines that the shuttles are able to see other shuttles nearby even though no collaboration has been established yet (a possible implementation can be based on GPS and a related protocol to broadcast position data to the shuttle's vicinity [48]). The read rule share-mode for the Shuttle role (cf. self instance name) is different as it always requires an instance of the Coord collaboration. Otherwise, it does not allow a shuttle to retrieve (read) the mode of the other Shuttle role. The provider instance name for that Shuttle role indicates that this role must provide this data.

4 Construction

To cover in particular the challenges of the Construction of SoS-Level Interactions for Self-Organization (C2), Construction of SoS-Level Structural Dynamics (C3), Construction of SoS-Level Runtime Knowledge Exchange (C4), Construction of Evolution of Smart SoS (C5), and Scalable Construction of Smart SoS (C6), SMARTSOS supports collaborations with a dynamic number of roles, runtime models, the independent evolution of the autonomous systems and their collaborations, and the specification of individual autonomous systems without having complete knowledge about the overall SoS. These aspects require at first a solid foundation for the concepts used for the construction and assurance of smart SoS. Based on this foundation we then can formally introduce types and instances of collaborations, systems, and SoS as well as the notions of runtime models and overlapping collaborations. Finally, we cover the evolution of smart SoS, that is, the set of types and instances of the SoS evolve (e.g., new types and instances are introduced).

4.1 Foundation

The formal foundation of SMARTSOS can be based on our former results in the context of MUML and experience in formal models for self-adaptive systems [49] and model transformations [50] based on graph transformation systems. In this context, graphs serve as a *formal model* to represent object configurations capturing the SoS-level architecture and runtime models (see the formal model depicted in Fig. 1) while the graph transformation rules such as SPs denote the behavior of the collaboration roles and graph conditions the required properties. To address various software aspects, we have developed extended attributed graph transformations systems covering real-time [51], probabilistic [52], and hybrid [47,53,54] behavior. These extensions can be used in the formal foundation of SMARTSOS.

In this paper, we will only introduce the basic ideas of the underlying formal model and refer to [54] for more details. The formal model we use can be described as follows:

Typed attributed graphs G with attributes describe the states of our model elements (system of systems, systems, and collaborations). They relate to the possible object configurations that a class diagram CD defines (cf. Figure 3 for the Coord collaboration). We use $\mathcal{G}_{\emptyset}(CD)$ to denote all those possible typed attributed graphs that fit to a class diagram CD and do not contain any role objects. We will further refer to the empty graph as $G_{\emptyset}$. An example of an object configuration is depicted in Figure 2, where two shuttles are linked by a Coord collaboration. The corresponding elements can also be found in the formal model depicted on top of Fig. 1.

Sets of graph transformation rules $\mathcal{R}$ define the behavior. They related to the SPs we employed earlier (cf. Figure 4 for the Coord collaboration). Rules $r \in \mathcal{R}$ can match a certain fragment of a graph representing the state and in addition describe how the graph changes when the rule is applied. Given a start graph and a number of rule applications we get a path π.

Furthermore, a suitable logic for state and sequence properties is assumed and we can describe whether a sequence property ϕ holds for a path π ($\pi \models \phi$) or for all paths generated by a start graph G and a set of rules $\mathcal{R}$ applied on G ($G, \mathcal{R} \models \phi$). Simple state conditions are specified by SPs without side effects (cf. Fig. 3), while TSSDs [46] can be employed to describe sequence properties.

Additionally, we use a refinement notion for graph transformation rule sets $\mathcal{R}' \sqsubseteq \mathcal{R}$ that guarantees preservation of safety properties while allowing us to extend the rules unless guaranteed behavior will be blocked.

We further exploit the fact that the different elements in our formal model are by construction separated either by their types or so-called pseudo types[4] For two rules separated by their types or pseudo types holds that the behavior cannot interfere in unexpected ways.

4.2 Collaborations, Systems, and System of Systems

Collaborations. Similar to *SoaML* and mUML, collaborations are the main elements to address the interactions among individual systems in SMARTSOS. However, we need an extended formalization as presented in more detail in [54] to cover also the structural dynamics such as joining a collaboration, leaving a collaboration, or changing the structure of a collaboration.[5] At first and in addition to the simplified view depicted in Fig. 1, we have to add extra nodes in the graphs for the role instances of a collaboration:

Definition 1 (see [54]). *A* role type ro^i *equals a node type* ro_i.

[4] Their types separate two rules if they have no node and edge type in common. Their pseudo types separate them, if for all nodes of shared type holds that always a single link to a special node with not shared type at the instance level are demanded.

[5] The terminology used in [54] has been adjusted and extended in this paper to better fit the concepts of SoS.

In our application example, the Shuttle element of the Coord collaboration is such a role type.

In addition to the basic notion of collaborations used in *SoaML/UML* and the extended one provided by MUML, we have to cover more information for a collaboration type as depicted in Fig. 1, 2, 3, and 4.

Definition 2 (see [54]). *A* collaboration type $\mathsf{Col}_i = (\mathsf{col}_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, R_i, \Phi_i)$ *consists of a collaboration type node* col_i*, a number of role types* ro_i^j*, an UML class diagram* CD_i*, a function* $R_i : \{\mathsf{col}_i, ro_i^1, \ldots, ro_i^{n_i}\} \mapsto 2^{\mathcal{R}}$ *assigning rules to role types, and a guaranteed property* Φ_i*.*

A collaboration instance of collaboration type Col_i is represented by a node of type col_i. In our example we have the Coord and Allocate collaboration types (see related dashed oval shapes in Fig. 1) as well as single :Coord and :Allocate collaboration instances (see related solid oval shapes in Fig. 1).

For two different role types ro_i^k and ro_i^l the set of assigned rules has to be disjoint $R_i(ro_i^k) \cap R_i(ro_i^l) = \emptyset$. The creation of collaboration instances of collaboration type Col_i is only possible through the collaboration type's roles ro_i^k and their assigned behavior $R_i(ro_i^k)$. E.g., see the create shuttle SP in Fig. 4.

The relation among the collaboration Col_i's role types $ro_i^1, \ldots, ro_i^{n_i}$ and any additional data types that are used within the collaboration are specified by the class diagram CD_i. The class diagrams of different collaborations have to be separated by different name spaces.

In our example, we have a role type Shuttle and its behavior rules are given by the set { create shuttle, move, create coordination, move in coordination } of SPs as depicted in Fig. 4. The corresponding class diagram CD_i defining the roles and all the other elements in the rules is depicted on the left-hand side of Fig. 3. The guaranteed property Φ_i is in our example the and-combination of two forbidden properties collision and missing collaboration depicted in the middle and right-hand side of Fig. 3. The read rules that are depicted in Fig. 6 are not explicitly covered in the formal model but the behavioral rules of the systems that realize the roles have to take them into account. This issue will be discussed in more detail in Section 4.3.

Within a collaboration many styles of interactions, particularly, synchronous or asynchronous ones can be used. For asynchronous message passing, the following scheme can be employed: an instance of the Shuttle role creates a new message (i.e., a node in the graph) and links it to another instance of the Shuttle role that should be the receiver of the message. The latter shuttle instance can afterwards process the message that has been linked to this instance. For synchronous interactions, an instance of a role may directly modify links and data of another instance. For example, an instance of the Shuttle role may if permitted directly change the mode of another instance of the Shuttle role (cf. Fig. 1).

Systems. Similar to *UML* and *SoaML*, we employ components to represent systems that interact through collaborations by realizing the related roles. Our

specification of a system further comprises safety properties that have to be fulfilled by the system's implementation.

Definition 3 (see [54]). *A* system type $\mathsf{Sys}_i = (\mathsf{sys}_i, (ro_i^1, \ldots, ro_i^{m_i}), CD_i, R_i, I_i, \Psi_i)$ *consists of a system type node* sys_i*, a number of role types* ro_i^j*, a class diagram* CD_i*, a function* $R_i : \{\mathsf{sys}_i, ro_i^1, \ldots, ro_i^{m_i}\} \mapsto 2^{\mathcal{R}}$ *assigning rules to role types, a set of initial rules* $I_i \subseteq R_i(\mathsf{sys}_i)$*, and a safety property* Ψ_i*.*

A system as an instance of system type Sys_i is represented by a node of type sys_i, which also fulfills the pseudo-typing requirements and thus separates elements from each other that belong to different systems. In our example we have the Shuttle and Station system types (see related dashed boxes in Fig. 1) as well as several :Shuttle and one :Station system of the corresponding types (see related solid boxes in Fig. 1).

All rules of Sys_i preserve a pseudo-typing linking of all nodes to sys_i. The function R_i is defined as for collaboration types (see Definition 2). The only way a system/instance of type Sys_i can be created is through the execution of any of the creation rules in I_i. The system type's class diagram CD_i contains all class diagrams of the collaboration types that are used by the system type.[6] Additionally, the system itself represented by a class sys_i (node type) and all data types required by the system are contained in CD_i. We further write $R_i(ro_i^k) \subseteq R_i(\mathsf{sys}_i)$ to refer to the set of all rules that belong to the system Sys_i's implementation of role ro_i^k.

System of Systems. To cover SoS, we employ system of system types and instances. System of systems combine collaboration and system types to a conceptual unit (depicted by the outer dashed box in Fig. 1).

Definition 4 (see [54]). *A* system of system type $\mathsf{SoS} = ((\mathsf{Col}_1, \ldots, \mathsf{Col}_n), (\mathsf{Sys}_1, \ldots, \mathsf{Sys}_m))$ *consists of a number of collaborations types* Col_i *and a number of system types* Sys_j*.*

Definition 5 (see [54]). *A* system of system instance *is a pair* $\mathsf{sys} = (\mathsf{SoS}, G_{\mathsf{sys}})$ *with system of system type* $\mathsf{SoS} = ((\mathsf{Col}_1, \ldots, \mathsf{Col}_n), (\mathsf{Sys}_1, \ldots, \mathsf{Sys}_m))$ *and an initial configuration* G_{sys} *that is type conform to* SoS*.*

As this paper does not include the details of a system type and system of system type for our example, we refer to [47, 54] to obtain these details and an example that cover abstract system specifications and a system of system type.

4.3 Runtime Models

As depicted in Fig. 1, in our formal model an idealized view of the context is directly visible and accessible by the shuttle systems. This idealization reflects

[6] A system type uses a collaboration type if it implements a role that has been defined for this collaboration type.

that the systems handle the related information about the physical and cyber world by means of runtime models and their exchange. The idea of the read rules depicted in Fig. 6 generalizes the concept of [32] to capture the capabilities of sensors and actuators for the physical world. Consequently, the formal model describes what can be read directly by local sensors or indirectly by the exchange of runtime models reflecting the context or the other systems' internal states.

The read rules are not formalized but the behavior rules of the systems have to adhere to them, that is, the rules must not access information that should not be visible to them through local runtime models or the exchange/sharing of runtime models. In the example of Fig. 1, this visible information of the local and shared runtime models relates to the local context as well as the mode and battery status of the shuttle itself, the topology as given by the common read rule, the position of the shuttles nearby as given by the detect-obstacle read rule, and the mode of the other shuttles that are connected by a Coord collaboration instance as given by the share-mode read rule (cf. Fig. 6).

It has to be noted that the outlined formal model is an *idealization*. It assumes that the systems operate on consistent and not delayed observations and ignore the risk of partial failures. However, it many cases the outlined idealization is quite reasonable. At first, any solution that would not work for the idealization will also likely not work under more realistic assumptions. Secondly, a more detailed design would in particular acknowledge that the effects due to partial failures and delayed and inconsistent observations are limited to the extent which can be tolerated for the considered problem addressed by the collaboration type (e.g., see the protocol developed in [48] that covers the loss of connection while preserving a basis for a safe behavior).

Another issue that has to be taken into account is that even though different collaborations can be employed to talk about different required interactions, as soon as they refer to the same phenomena of the physical or cyber world, the observations in the different collaborations must be consistent. Therefore, we require that in these cases an initial collaboration has to cover the interrelated phenomena of the domain that should be considered in a consistent manner and share these phenomena with the other collaborations.

If other and more specific collaboration types take a subset of the phenomena of the physical or cyber world covered by such an initial collaboration type into account, they have to extend the initial collaboration type. Then, these more specific collaboration types cannot be specified completely separated from each other and the shared one.

Definition 6. *An* overlapping collaboration type $\mathsf{Col}_i = (\mathsf{col}_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, R_i, \Phi_i)$ *extending a shared collaboration type* $\mathsf{Col}_j = (\mathsf{col}_j, (ro_j^1, \ldots, ro_j^{n_j}), CD_j, R_j, \Phi_j)$ *consists of a collaboration type node* col_i*, a number of roles* ro_i^j *with* $n_i \geq n_j$ *and for all* $1 \leq l \leq n_j$ $ro_i^l = ro_j^l$*, an UML class diagram* CD_i *extending* CD_j*, a function* $R_i : \{\mathsf{col}_i, ro_i^1, \ldots, ro_i^{n_i}\} \mapsto 2^{\mathcal{R}}$ *extending* R_j *assigning rules to roles, and a guaranteed property* Φ_i*.*

In our example depicted in Fig. 1, the Allocate collaboration type refines the Coord collaboration type and therefore is aware that the shuttles may move.

4.4 Evolution

One aspect of our motivation for this work is that individual systems in a smart SoS are subject to independent changes (evolution), which has to be handled by construction and assurance. In the following, we will explicitly consider the modeling of evolution, which is not addressed by *SoaML* or MUML.

Definition 7 (see [54]). *An* extended evolution sequence *is a sequence of system of systems* $(\mathsf{SoS}_1, G_S^1), \ldots, (\mathsf{SoS}_n, G_S^n)$ *such that (1)* SoS_{i+1} *only extends* SoS_i *by additional collaboration and system types, (2)* G_S^{i+1} *is also type conform to* SoS_i*, and (2)* G_S^{i+1} *can be reached from* G_S^i *in the system of system* (SoS_i, G_S^i)*.*

An evolution sequence *is a sequence of system of system types* $\mathsf{SoS}_1, \ldots, \mathsf{SoS}_n$ *such that at least one related extended evolution sequence* $(\mathsf{SoS}_1, G_S^1), \ldots, (\mathsf{SoS}_n, G_S^n)$ *exists.*

As presented in [54], type conformance for the introduced evolution concepts can be defined that ensure a proper typing of collaborations, systems, and system of systems.

5 Assurance

Existing instance-based formal approaches do not scale and are often not applicable to the specific settings of SoS such as openness, dynamic structures, and independent evolution. Thus, the challenges of establishing Assurance of SoS-Level Interactions for Self-Organization (A2), Assurance of SoS-Level Structural Dynamics (A3), and Assurance of Evolution of Smart SoS (A5) and in particular the Scalable Assurance of Smart SoS (A6) and the Assurance of Smart SoS with Restricted Knowledge (A7) for the assurance for smart SoS are not covered.

Therefore and similar to the MUML approach, we propose establishing the required guarantees for the assurance by referring only to the collaboration and system types rather than to the instance level. For the verification at the type level we show that the correctness proven for the collaboration and system types and only type conformance for the system of systems type will by construction imply that the related correctness also holds at the instance level for any possible configurations of the related system of systems. The scalability of our approach comes from the fact that the size of the type level is independent of the size of the instance level. However, we have to show as a general property of our approach that the results we yield for the type level are also valid for the instance level.

To tackle assurance for the envisioned SMARTSOS approach, we will first address the correctness at the type level looking into collaboration and system types. Then, we will look at the instances of collaboration and system types and show that the correctness established for the types can be transfered to the

instances. Afterwards, we outline how the special case of collaborations with overlapping runtime models can be handled. Finally, we cover evolution where the set of types and instances of a SoS may evolve.

5.1 Collaboration and System Types

We start our considerations with defining what we mean by correct types for collaborations and systems (see dashed inner elements in Fig. 1).

Definition 8 (see [54]). *A collaboration type* $\mathsf{Col}_i = (\mathsf{col}_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, R_i, \Phi_i)$ *is* correct *if for all initial configurations* $G_I \in \mathcal{G}_\emptyset(CD_i)$ *holds that for* $R_i(\mathsf{Col}_i) = R_i(ro_i^1) \cup \cdots \cup R_i(ro_i^n) \cup R_i(\mathsf{col}_i))$ *the overall behavior of the collaboration the reachable collaboration configurations are correct:* $G_I, R_i(\mathsf{Col}_i) \models \Phi_i$.

Please note that looking only at the behavior of all roles and to consider only the initial object configurations G_I without any roles is sufficient to cover all possible behavior, as we have a closed model where only the behavior of the roles is allowed to create or delete roles or any other considered elements.

For our example and the behavior rules of the Coord collaboration type as depicted in Fig. 4, it can be formally verified that the collaboration type is correct employing an automated checker (cf. [47, 51]). These checks only work for state properties and operate at the level of the types. Therefore, they do not have to consider the instance situation that would require checking infinite many and arbitrary large object configurations over arbitrary long sequences of steps. Another option that would allow us to cover sequence properties might be to use incomplete techniques such as simulation/testing or bounded or statistical model checking to establish a certain confidence for the correctness of a specific collaboration type.

A correct system type requires that the resulting behavior ensures the guarantees and that the system's implementation refines the combined role behavior.

Definition 9 (see [54]). *A system type* $\mathsf{Sys}_i = (\mathsf{sys}_i, (ro_i^1, \ldots, ro_i^{m_i}), CD_i, I_i, \Psi_i)$ *is* correct *if for all initial configurations* $G_I \in \mathcal{G}_\emptyset(CD_i)$ *holds that (1) the reachable configurations are correct* $G_I, R_i(\mathsf{sys}_i) \cup COMP(\mathsf{Sys}_i) \cup I_i \models \Psi_i$ *and that (2) the system behavior* $R_i(\mathsf{sys}_i)$ *refines the orthogonally combined role behavior and creation behavior* $R_i(\mathsf{sys}_i) \sqsubseteq R_i(ro_i^1) \cup \cdots \cup R_i(ro_i^{m_i}) \cup I_i$. *To add the collaboration behavior to the system behavior for each role without the role itself, we employ here* $COMP(\mathsf{Sys}_i) = \bigcup_{1 \leq l \leq m_i} COMP(\mathsf{Sys}_i, ro_i^l)$ *with* $COMP(\mathsf{Sys}_i, ro_i^l) = R_j(\mathsf{Col}_j)$ *which is covered by* $R_i(\mathsf{sys}_i)$ *to derive a related closed behavior.*

Due to lack of space, we do not discuss an example for a correct system type and refer to [47,54] for such an example and its formal verification. Again, another option might be to employ incomplete techniques such as simulation/testing or bounded or statistical model checking to establish a certain confidence for the correctness of a specific system type.

5.2 Collaborations and System Instances

After defining our notion of correctness for the types, we have to define the related notion of correctness at the instance level (cf. the solid elements in Fig. 1).

Definition 10 (see [54]). *A concrete system of system* $\mathsf{sos} = (\mathsf{SoS}, G_S)$ *with system of system type* $\mathsf{SoS} = ((\mathsf{Col}_1, \ldots, \mathsf{Col}_n), (\mathsf{Sys}_1, \ldots, \mathsf{Sys}_m))$ *is* correct *if it holds:*

$$G_S, R(\mathsf{sys}_1) \cup \cdots \cup R(\mathsf{sys}_m) \cup R(\mathsf{col}_1) \cup \cdots \cup R(\mathsf{col}_n) \models \Phi_1 \wedge \cdots \wedge \Phi_n \wedge \Psi_1 \wedge \cdots \wedge \Psi_m.$$

Then, we can show in the following Theorem 1 that the type conformance of the system of system type and the correctness of collaboration types and system types ensures correctness at the instance level for the system of system.

Theorem 1 ([54]). *A system of systems* $\mathsf{sos} = (\mathsf{SoS}, G_\emptyset)$ *with system of system type* $\mathsf{SoS} = ((\mathsf{Col}_1, \ldots, \mathsf{Col}_n), (\mathsf{Sys}_1, \ldots, \mathsf{Sys}_m))$ *is* correct *if (1) the system of system type* SoS *is* type conform*, (2) all collaboration types* $\mathsf{Col}_1, \ldots, \mathsf{Col}_n$ *are* correct*, and (3) all system types* $\mathsf{Sys}_1, \ldots, \mathsf{Sys}_m$ *are* correct*.*

Theorem 1 provides sufficient but not necessary conditions to ensure the correctness. It permits us to straightforward establish the required correctness of the types by checking refinement and the guarantees for the properties using the rule sets as employed in condition (2) and (3).[7]

Due to lack of space, we do not present an example for a correct system of system type here and refer to [47, 54] for such an example. In general, at the system of systems level, we only have to collect the evidence for correctness that is provided for the collaboration and system types being part of this system of systems.

5.3 Runtime Models

The sharing of runtime models by a single collaboration type as depicted in Fig. 1 for the Coord collaboration can be covered with the introduced basic concepts for collaborations. Thus, the results of Theorem 1 also apply in such cases and permit us to provide the required assurance. However, this does not hold for overlapping collaborations.

For collaboration types that refine a shared collaboration type we can exploit the following Definition 12 and Lemma 1 that outline under which circumstances the correctness of the composition of all overlapping collaboration types can be derived only on the basis of the correctness of all the overlapping collaboration types, the correctness of the refined shared collaboration type, and the compatibility of their roles.

[7] As outlined in [54] in detail, based on the refinement of the rule sets for the involved roles the result of Theorem 1 can also be extended to abstract system and collaboration types.

Definition 11. *An overlapping collaboration type* $\mathsf{Col}_i = (\mathsf{col}_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, R_i, \Phi_i)$ *extending the shared collaboration type* $\mathsf{Col}_j = (\mathsf{col}_j, (ro_j^1, \ldots, ro_j^{n_j}), CD_j, R_j, \Phi_j)$ *is* correct *if for all initial configurations* $G_I \in \mathcal{G}_\emptyset(CD_i)$ *holds that the reachable collaboration configurations are correct* $G_I, R_i(\mathsf{Col}_i) \models \Phi_j \wedge \Phi_i$ *for* $R_i(\mathsf{Col}_i) = R_i(ro_i^1) \cup \cdots \cup R_i(ro_i^n) \cup R_i(\mathsf{col}_i))$ *the overall behavior of the collaboration and that all added roles refine roles of the refined shared collaboration type:* $\forall l \in [n_j + 1, n_i] \exists k \in [1, n_j] R_i(ro_i^l) \sqsubseteq R_i(ro_j^k)$.

We can combine a set of overlapping collaboration types of the same shared refined collaboration type to obtain the related resulting collaboration type.

Definition 12. *For a set of overlapping collaboration types* $\mathsf{Col}_{i_1}, \ldots, \mathsf{Col}_{i_m}$ *extending a shared collaboration type* $\mathsf{Col}_0 = (\mathsf{col}_0, (ro_0^1, \ldots, ro_0^{n_0}), CD_0, R_0, \Phi_0)$ *the* resulting collaboration type *is defined as* $\mathsf{Col}_i = (\mathsf{col}_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, R_i, \Phi_i)$ *with a collaboration type node* col_i*, a set of roles that unites the roles sets of* $\mathsf{Col}_0, \mathsf{Col}_{i_1}, \ldots, \mathsf{Col}_{i_m}$*, an UML class diagram* $CD_i = CD_0 \cup \bigcup_{1 \leq k \leq m} CD_{i_k}$*, a function* $R_i : \{\mathsf{col}_i, ro_i^1, \ldots, ro_i^{n_i}\} \mapsto 2^{\mathcal{R}}$ *extending* R_0 *and all* R_{i_k} *for* $1 \leq k \leq m$ *assigning rules to roles, and a guaranteed property* $\Phi_i = \Phi_0 \wedge (\wedge_{1 \leq k \leq m} \Phi_{i_k})$.

In a next step we can show with the following Lemma that the resulting collaboration type is correct, if all overlapping collaboration types of the related shared collaboration types are correct.

Lemma 1. *If all overlapping collaboration types* $\mathsf{Col}_{i_1}, \ldots, \mathsf{Col}_{i_m}$ *and the shared refined collaboration type* $\mathsf{Col}_0 = (\mathsf{col}_0, (ro_0^1, \ldots, ro_0^{n_j}), CD_0, R_0, \Phi_0)$ *are correct, then the resulting collaboration type* $\mathsf{Col}_i = (\mathsf{col}_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, R_i, \Phi_i)$ *is also correct.*

Proof. For any correct overlapping collaboration type $\mathsf{Col}_{i_k} = (\mathsf{col}_{i_k}, (ro_{i_k}^1, \ldots, ro_{i_k}^{n_{i_k}}), CD_{i_k}, R_{i_k}, \Phi_{i_k})$ *of the shared refined collaboration type* $\mathsf{Col}_0 = (\mathsf{col}_0, (ro_0^1, \ldots, ro_0^{n_j}), CD_0, R_0, \Phi_0)$ *holds that* $G_I, R_{i_k}(\mathsf{Col}_{i_k}) \models \Phi_0 \wedge \Phi_{i_k}$*. As the extension of each overlapping collaboration type are disjoint and the shared behavior is refining the roles of the shared collaboration, we can conclude that also* $G_I, R_i(\mathsf{Col}_i) \models \Phi_{i_k}$ *will hold. As* $\Phi_i = \Phi_0 \wedge (\wedge_{1 \leq k \leq m} \Phi_{i_k})$ *we only have to combine this finding for all* $1 \leq k \leq m$ *and get* $G_I, R_i(\mathsf{Col}_i) \models \wedge_{1 \leq k \leq m} \Phi_{i_k}$ *and thus* $G_I, R_i(\mathsf{Col}_i) \models \Phi_i$ *such that* Col_i *is correct.* □

Due to Lemma 1 we can now employ Theorem 1 to cover overlapping collaborations.

5.4 Evolution

So far the presented results for assurance do not cover the evolution of SoS. Therefore, we will extend the former results to cover typical evolution scenarios such as adding new collaboration or system types. If we look at our former results in more detail, we can notice that the assumption has been made that all types are known at verification time. This assumption is not true for a steadily

evolving system where new type definitions are added over time. Furthermore, the different organizations involved in an SoS will only have a partial view and thus do not know all currently existing types in the SoS. For a given *extended evolution sequence* (cf. Definition 7) we can define correctness as follows:

Definition 13 (see [54]). *An extended evolution sequence* $(\mathsf{SoS}_1, G_S^1), \ldots, (\mathsf{SoS}_n, G_S^n)$ *with* $\mathsf{SoS}_n = ((\mathsf{Col}_1, \ldots, \mathsf{Col}_p), (\mathsf{Sys}_1, \ldots, \mathsf{Sys}_q))$ *is* correct *if for any combined path* $\pi_1 \circ \cdots \circ \pi_n$ *such that* π_i *is a path in* SoS_i *leading from* G_S^i *to* G_S^{i+1} *for* $i < n$ *and that* π_n *is a path in* SoS_n *starting from* G_S^n *holds:* $\pi_1 \circ \cdots \circ \pi_n \models \Phi_1 \wedge \cdots \wedge \Phi_p \wedge \Psi_1 \wedge \cdots \wedge \Psi_q$. *An evolution sequence* $\mathsf{SoS}_1, \ldots, \mathsf{SoS}_n$ *is* correct *if all possible related extended evolution sequence* $(\mathsf{SoS}_1, G_S^1), \ldots, (\mathsf{SoS}_n, G_S^n)$ *are correct.*

A first observation is that SoS_n contains all types defined in any SoS_i. However, for a combined path $\pi_1 \circ \cdots \circ \pi_n$ such that π_i is a path in SoS_i leading from G_S^i to G_S^{i+1} for $i < n$ does not hold in general that an equal path π in SoS_n exists that goes through all G_S^i, as the rules added by later added types may influence the possible outcomes if added at the start.[8] Another observation is that the properties guaranteed for newly introduced collaboration or system types have to be true as long as the types have not yet been introduced as otherwise the evolution cannot be correct. We can exploit these observations and construct related collaboration types $E(\mathsf{Col}_i)$ and system types $E(\mathsf{Sys}_j)$ encoding that the types come into existence later. Based on this we can then define $E(\mathsf{SoS}_1, \mathsf{SoS}_n)$ as that system of system type where the types of SoS_n not present in SoS_1 can come into existence later. $E(\mathsf{SoS}_1, \mathsf{SoS}_n)$ therefore includes all possible combined paths of any possible extended evolution sequences for a given evolution sequence $\mathsf{SoS}_1, \ldots, \mathsf{SoS}_n$.

We can then use the fact that the related dynamically evolving system of system type includes all possible extended evolution sequences to check also the correctness for all possible evolution sequences.

Theorem 2 (see [54]). *An evolution sequence of systems* $\mathsf{SoS}_1, \ldots, \mathsf{SoS}_n$ *is* correct *if the related dynamic evolving system of system type* $E(\mathsf{SoS}_1, \mathsf{SoS}_n)$ *is correct.*

Lemma 2 (see [54]). *For a correct collaboration type Col holds also that its dynamic extension* $E(Col)$ *is correct. For a correct system type* Sys *holds also that its dynamic extension* $E(\mathsf{Sys})$ *is correct.*

Due to Lemma 2, it is sufficient to simply check the collaboration and system types and this already guarantees that any extended evolution sequence will also show correct behavior.

Moreover, due to Theorem 2 and Lemma 2, an organization that wants to extend the system of system type accordingly does not require any knowledge about all the other types besides those which are refined or where an overlap

[8] For example, in a refined model there may be urgent rules that have to be executed if enabled and thus may preempt other rules when added during the evolution.

exists. Furthermore, if two independent extensions are done which do not refer to each other, the concrete order of these extensions does not matter as the checks remain the same. Therefore, each organization can simply check its own extension by means of added collaboration and system types without considering when the other extensions are enacted.

Due to lack of space, we do not present an example for a correct system of system type with evolution here and refer to [47, 54] for such an example.

6 Discussion

In SMARTSOS, we combine ideas from mUML and EUREMA to tackle the challenges for smart SoS that are discussed in Section 2. A radically different and more abstract perspective on the SoS-level interactions based on runtime models and collaborations is employed to overcome the limitations of the state-of-the-art and our former approaches and to master the complexity of smart SoS. In the following, we will discuss which challenges are addressed by the proposed ideas, particularly, by the concepts of collaborations and runtime models, the novelty of these ideas, and the additional benefits of these ideas.

6.1 Runtime Models

As discussed for the envisioned SMARTSOS approach in Section 3 and its formal model in Sections 4 and 5, SMARTSOS employs *generic runtime models.*

On the one hand, this supports the engineering of the self-adaptation for individual systems in the smart SoS as required by the challenge of Construction/Assurance of Self-Adaptation (C1/A1) (cf. Section 2). Similar to EUREMA, the self-adaptation for each system is implemented in SMARTSOS by a feedback loop with monitor, analyze, plan, and execute activities that operate on the generic runtime models. For instance, the self-adaptive behavior of each shuttle in the large-scale transport system is specified by such feedback loops operating on partially shared runtime models that reflect the shuttle itself and the shuttle's context (cf. top of the left-hand side of Fig. 2).

On the other hand, SMARTSOS uses the generic runtime models to exchange information between individual collaborating systems, which addresses the challenge of Construction/Assurance of SoS-Level Runtime Knowledge Exchange (C4/A4). This aspect distinguishes SMARTSOS from the state of the art in engineering SoS that employ specific and optimized runtime models without exchanging them among individual systems.

Therefore, SMARTSOS goes beyond the mUML approach and the state of the art by supporting *generic runtime models* of the contexts and the systems in the SoS. SMARTSOS also goes beyond EUREMA and the state of the art by supporting the runtime exchange of these models between the individual collaborating systems in an SoS. In the context of collaborations, such runtime models can also reflect agreements between the collaborating systems which is a prerequisite for jointly achieving the SoS-level goals. Thereby, such collaborations

may be established in a self-organizing manner while each system still evolves and self-adapts independently from the other systems in the SoS.

While the *generic* nature of the runtime models used in SMARTSOS clearly leads to a higher complexity of the models compared to the state of the art, it also leverages a number of benefits: (B1) As a "success in regulation implies that a sufficiently similar model must have been built," [55], it is consequently unavoidable that the software captures all the relevant variety of the controlled system itself, the requirements, and the context to be able to control their variability effectively. (B2) Herbert A. Simon observed for the example of an ant that "[t]he apparent complexity of its behavior over time is largely a reflection of the complexity of the environment in which it finds itself" [56, p. 52]. Thus, it can be expected that including the physical and cyber environment by runtime models will in fact help to reduce the complexity of the remaining software solution that operates on the basis of these runtime models. Finally, (B3) while specialized solutions that only capture a minimal amount of information about the environments lead to simpler software in the short run, it can be expected that the envisioned generic runtime models without such optimizations result in a *direct-mapping* [57] between the original (e.g., the system or context) and the model. Such a mapping is usually more stable in the long run and considerably eases interoperability. The latter aspect is a critical issue to achieve open and dynamic collaborations in smart SoS.

6.2 Collaborations

Based on the generic runtime models, SMARTSOS employs *open and dynamic collaborations* (cf. Section 3) to achieve self-organizing interactions between individual systems of the smart SoS. The collaboration concept of SMARTSOS is also covered in the formal model for the construction and assurance of smart SoS (cf. Sections 4 and 5). In contrast to state of the art approaches, this perspective on the SoS-level interactions addresses the challenges of Construction/Assurance of SoS-Level Interactions for Self-Organization (C2/A2) and Construction/Assurance of SoS-Level Structural Dynamics (C3/A3) (cf. Section 2). Moreover, it is key to address the challenges of Construction/Assurance of Evolution of Smart SoS (C5/A5), Scalable Construction/Assurance of Smart SoS (C6/A6), and Construction/Assurance of Smart SoS with Restricted Knowledge (C7/A7).

In this context, SMARTSOS extends EUREMA that does not consider collaborations at all since EUREMA focuses on centralized and non-distributed systems. In contrast to MUML, SMARTSOS employs *open and more dynamic collaborations* that are governed by laws supporting self-organization at the SoS level and that support the structural dynamics of smart SoS where, for example, systems may dynamically join or leave the SoS. The collaboration concept of SMARTSOS supports abstracting details of individual systems in the SoS by means of roles, runtime models, and behavioral contracts while distinguishing the type and instance levels (cf. Section 3). This collaboration concept leverages the independent development, operation, management, and evolution of

these systems (cf. Sections 4 and 5). Thus, the evolution of smart SoS and its contained systems with respect to construction (cf. Section 4.4) and assurance (cf. Section 5.4) aspects is supported. By abstraction and explicitly distinguishing the type and instance levels of smart SoS, the construction and assurance are scalable as they mainly work at the type level – hence abstracting from the sheer scale and number of all possible instance situations of a smart SoS. In the same line of reasoning, the construction and assurance of smart SoS works despite not considering the complete instance situation and thus all details of the SoS. Therefore, SMARTSOS can handle the restricted knowledge of SoS caused by multiple authorities governing the SoS.

In general, the collaboration concept of SMARTSOS is motivated by the beneficial observations that (B4) in our society we have established legal domains, which we consider independent of each other. We can expect that the individuals behave according to the laws of each of these legal domains independent of the other domains. This approach allows us to cooperate even though the systems and legal domains evolve and adapt in principle independently from each other (cf. law-governed interaction [58]). In the traffic domain, for example, rules for driving vehicles and related regulations and laws impose what individual drivers are allowed to do while within these bounds the individuals are free to act. In our example, the Coord collaboration in Fig. 1 establishes such a solution with respect to the driving behavior of the shuttles. As another example, individuals in the traffic domain may establish contracts with each other to allocate parking slots. In our example, a Shuttle may establish such a contract to have the privilege to stop at a specific platform of a Station by means of an Allocate collaboration instance as depicted in Fig. 1.

7 Conclusion and Future Work

In this paper we analyzed the open challenges for the envisioned smart SoS looking in particular into construction and assurance of such SoS. In this context, we presented our ideas how to tackle this vision with our SMARTSOS approach, specifically, by employing open and adaptive collaborations and generic models at runtime. We discussed that by supporting generic runtime models at the SoS level, the challenge of Construction/Assurance of SoS-Level Runtime Knowledge Exchange (C4/A4) can be covered by SMARTSOS. Furthermore, based on such runtime models, the SMARTSOS collaboration concept directly covers the challenges of Construction/Assurance of SoS-Level Interactions for Self-Organization (C2/A2) and Construction/Assurance of SoS-Level Structural Dynamics (C3/A3). Moreover, it provides the required foundation to tackle the challenges of Construction/Assurance of Evolution of Smart SoS (C5/A5), Scalable Construction/Assurance of Smart SoS (C6/A6), and Construction/Assurance of Smart SoS with Restricted Knowledge (C7/A7). While SMARTSOS addresses the challenges related to the SoS level, the challenge of Construction/Assurance of Self-Adaptation (C1/A1) of individual systems in the SoS is mainly addressed by our former work on MUML and EUREMA.

Our plans for future work are to further elaborate SMARTSOS by extending the model-driven EUREMA approach [33] with open and adaptive collaborations and means for the distributed management of runtime models. Additionally, we plan to further strengthen the links between runtime and development-time models [59] and model-driven techniques in runtime scenarios [45].

Acknowledgment. We thank Basil Becker for his contribution to the presented results and his comments on draft versions of this paper.

References

1. Maier, M.W.: Architecting principles for systems-of-systems. Systems Engineering 1(4), 267–284 (1998)
2. Valerdi, R., Axelband, E., Baehren, T., Boehm, B., Dorenbos, D., Jackson, S., Madni, A., Nadler, G., Robitaille, P., Settles, S.: A research agenda for systems of systems architecting. Intl. Journal of System of Systems Engineering 1(1-2), 171–188 (2008)
3. Northrop, L., Feiler, P.H., Gabriel, R.P., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D.: Ultra-Large-Scale Systems: The Software Challenge of the Future. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2006)
4. Broy, M., Cengarle, M.V., Geisberger, E.: Cyber-Physical Systems: Imminent Challenges. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 1–28. Springer, Heidelberg (2012)
5. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward Open-World Software: Issue and Challenges. Computer 39(10), 36–43 (2006)
6. Cheng, B.H.C., et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
7. de Lemos, R., et al.: Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013)
8. Di Marzo Serugendo, G., Gleizes, M.P., Karageorgos, A. (eds.): Self-organising Software. Natural Computing Series. Springer (2011)
9. Mens, T., Demeyer, S.: Software Evolution. Springer (2008)
10. Mittal, S., Risco Martin, J.: Model-driven systems engineering for netcentric system of systems with DEVS unified process. In: Simulation Conference (WSC), pp. 1140–1151 (Winter 2013)
11. Object Management Group (OMG): Service oriented architecture Modeling Language (SoaML) Specification, Version 1.0.1. (2012)
12. UML 2.4 Superstructure Specification, Version 2.4, ptc/2010-11-14 (2010)
13. Sanders, R.T., Castejón, H.N., Kraemer, F., Bræk, R.: Using UML 2.0 Collaborations for Compositional Service Specification. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 460–475. Springer, Heidelberg (2005)
14. Broy, M., Krüger, I., Meisinger, M.: A formal model of services. ACM Trans. Softw. Eng. Methodol. 16 (2007)

15. Baresi, L., Heckel, R., Thöne, S., Varró, D.: Modeling and Validation of Service-Oriented Architectures: Application vs. Style. In: Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-11, pp. 68–77. ACM, New York (2003)
16. Baresi, L., Heckel, R., Thöne, S., Varró, D.: Style-based modeling and refinement of service-oriented architectures. Software and Systems Modeling 5(2), 187–207 (2006)
17. Hölzl, M., Wirsing, M.: Towards a System Model for Ensembles. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 241–261. Springer, Heidelberg (2011)
18. Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge University Press, New York (1999)
19. Milner, R.: The Space and Motion of Communicating Agents. Cambridge University Press (2009)
20. Varró, D.: Automated formal verification of visual modeling languages by model checking. Software and System Modeling 3(2), 85–113 (2004)
21. Rensink, A.: Towards model checking graph grammars. In: Proc. of the 3rd Workshop on Automated Verification of Critical Systems, AVoCS, University of Southampton, pp. 150–160 (2003)
22. Frias, M.F., Galeotti, J.P., López Pombo, C.G., Aguirre, N.M.: DynAlloy: Upgrading Alloy with actions. In: Proceedings of the 27th International Conference on Software Engineering. ICSE 2005, pp. 442–451. ACM (2005)
23. Ölveczky, P.C., Meseguer, J.: Specification and Analysis of Real-Time Systems Using Real-Time Maude. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 354–358. Springer, Heidelberg (2004)
24. Zhang, J., Goldsby, H.J., Cheng, B.H.: Modular verification of dynamically adaptive systems. In: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, AOSD 2009, pp. 161–172. ACM, New York (2009)
25. Baldan, P., Corradini, A., König, B.: A Static Analysis Technique for Graph Transformation Systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 381–395. Springer, Heidelberg (2001)
26. Bauer, J., Wilhelm, R.: Static Analysis of Dynamic Communication Systems by Partner Abstraction. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 249–264. Springer, Heidelberg (2007)
27. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., Scheideler, P.: Tool Support for the Design of Self-Optimizing Mechatronic Multi-Agent Systems. International Journal on Software Tools for Technology Transfer (STTT) 10(3), 207–222 (2008)
28. Giese, H., Schäfer, W.: Model-Driven Development of Safe Self-Optimizing Mechatronic Systems with MechatronicUML. In: Cámara, J., de Lemos, R., Ghezzi, C., Lopes, A. (eds.) Assurances for Self-Adaptive Systems. LNCS, vol. 7740, pp. 152–186. Springer, Heidelberg (2013)
29. Giese, H., Burmester, S., Schäfer, W., Oberschelp, O.: Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, SIGSOFT 2004/FSE-12, pp. 179–188. ACM (2004)

30. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the compositional verification of real-time uml designs. In: Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-11, pp. 38–47. ACM, New York (2003)
31. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, pp. 72–81. ACM (2006)
32. Giese, H., Klein, F.: Systematic verification of multi-agent systems based on rigorous executable specifications. Int. J. Agent-Oriented Softw. Eng. 1(1), 28–62 (2007)
33. Vogel, T., Giese, H.: Model-Driven Engineering of Self-Adaptive Software with EUREMA. ACM Trans. Auton. Adapt. Syst. 8(4), 18:1–18:33 (2014)
34. Blair, G., Bencomo, N., France, R.B.: Models@run.time. Computer 42(10), 22–27 (2009)
35. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation: Foundations, vol. 1. World Scientific Pub. Co. (1997)
36. Rozenberg, G., Ehrig, H., Engels, G., Kreowski, H. (eds.): Handbook of graph grammars and computing by graph transformation. applications, languages, and tools, vol. 2. World Scientific (1999)
37. The Open Group Architectural Framework (TOGAF), version 9.1. Open Group Standard (2011)
38. Vassev, E., Hinchey, M.: The Challenge of Developing Autonomic Systems. Computer 43(12), 93–96 (2010)
39. Marconi, A., Bucchiarone, A., Bratanis, K., Brogi, A., Camara, J., Dranidis, D., Giese, H., Kazhamiakink, R., de Lemos, R., Marquezan, C., Metzger, A.: Research challenges on multi-layer and mixed-initiative monitoring and adaptation for service-based systems. In: 2012 Workshop on European Software Services and Systems Research - Results and Challenges (S-Cube), pp. 40–46. IEEE (2012)
40. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: 2007 Future of Software Engineering, FOSE 2007, pp. 37–54. IEEE Computer Society, Washington, DC (2007)
41. Vogel, T., Giese, H.: Adaptation and Abstract Runtime Models. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010, pp. 39–48. ACM (2010)
42. Vogel, T., Seibel, A., Giese, H.: The Role of Models and Megamodels at Runtime. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 224–238. Springer, Heidelberg (2011)
43. Wätzoldt, S., Giese, H.: Classifying Distributed Self-* Systems Based on Runtime Models and Their Coupling. In: Proceedings of the 9th International Workshop on Models@run.time. CEUR Workshop Proceedings, vol. 1270, pp. 11–20. CEUR-WS.org (2014)
44. Kephart, J.O., Chess, D.: The Vision of Autonomic Computing. Computer 36(1), 41–50 (2003)
45. Giese, H., Lambers, L., Becker, B., Hildebrandt, S., Neumann, S., Vogel, T., Wätzoldt, S.: Graph Transformations for MDE, Adaptation, and Models at Runtime. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) SFM 2012. LNCS, vol. 7320, pp. 137–191. Springer, Heidelberg (2012)
46. Klein, F., Giese, H.: Joint Structural and Temporal Property Specification using Timed Story Sequence Diagrams. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 185–199. Springer, Heidelberg (2007)

47. Becker, B.: Architectural modelling and verification of open service-oriented systems of systems. PhD thesis, Hasso-Plattner-Institut für Softwaresystemtechnik, Universität Potsdam (2014)
48. Giese, H., Burmester, S., Klein, F., Schilling, D., Tichy, M.: Multi-Agent System Design for Safety-Critical Self-Optimizing Mechatronic Systems with UML. In: Henderson-Sellers, B., Debenham, J. (eds.) OOPSLA 2003 - Second International Workshop on Agent-Oriented Methodologies, Anaheim, CA, USA, pp. 21–32. Center for Object Technology Applications and Research (COTAR), University of Technology, Sydney, Australia (2003)
49. Becker, B., Giese, H.: Modeling of Correct Self-Adaptive Systems: A Graph Transformation System Based Approach. In: Proceedings of the 5th International Conference on Soft Computing As Transdisciplinary Science and Technology, CSTST 2008, pp. 508–516. ACM (2008)
50. Giese, H., Hildebrandt, S., Lambers, L.: Bridging the gap between formal semantics and implementation of triple graph grammars. Software and Systems Modeling 13(1), 273–299 (2014)
51. Becker, B., Giese, H.: On Safe Service-Oriented Real-Time Coordination for Autonomous Vehicles. In: Proc. of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), pp. 203–210. IEEE Computer Society Press (2008)
52. Krause, C., Giese, H.: Probabilistic Graph Transformation Systems. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 311–325. Springer, Heidelberg (2012)
53. Becker, B., Giese, H.: Cyber-Physical Systems with Dynamic Structure: Towards Modeling and Verification of Inductive Invariants. Technical Report 64, Hasso Plattner Institute at the University of Potsdam, Germany (2012)
54. Giese, H., Becker, B.: Modeling and Verifying Dynamic Evolving Service-Oriented Architectures. Technical Report 75, Hasso Plattner Institute at the University of Potsdam, Germany (2013)
55. Conant, R.C., Ashby, W.R.: Every good regulator of a system must be a model of that system. Intl. J. Systems Science 1(2), 89–97 (1970)
56. Simon, H.A.: The Sciences of the Artificial, 3rd edn. The MIT Press (1996)
57. Meyer, B.: 30. In: Concurrency, Distribution, Client-Server and the Internet, 2nd edn., pp. 951–1036. Prentice Hall (1997)
58. Minsky, N.H., Ungureanu, V.: Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. ACM Transactions on Software Engineering and Methodology (TOSEM) 9(3), 273–305 (2000)
59. Vogel, T., Giese, H.: On Unifying Development Models and Runtime Models. In: Proceedings of the 9th International Workshop on Models@run.time. CEUR Workshop Proceedings, vol. 1270, pp. 5–10. CEUR-WS.org (2014)