Softw Syst Model (2019) 18:157–189
https://doi.org/10.1007/s10270-017-0583-z

THEME SECTION PAPER

Transactional execution of hierarchical reconfigurations in cyber-physical systems

Christian Heinzemann[1] · **Steffen Becker**[2] · **Andreas Volk**[3]

Received: 15 March 2016 / Revised: 18 December 2016 / Accepted: 10 January 2017 / Published online: 1 February 2017

Abstract Cyber-physical systems reconfigure the structure of their software architecture, e.g., to avoid hazardous situations and to optimize operational conditions like their energy consumption. These reconfigurations have to be safe so that the systems protect their users or environment against harmful conditions or events while changing their structure. As software architectures are typically built on components, reconfiguration actions need to take into account the component structure. This structure should support vertical composition to enable hierarchically encapsulated components. While many reconfiguration approaches for cyber-physical and embedded real-time systems allow the use of hierarchically embedded components, i.e., vertical composition, none of them offers a modeling and verification solution to take hierarchical composition, i.e., encapsulation, into account thus limiting reuse and compositional verification. In this paper, we present an extension to our existing modeling language, MECHATRONICUML, to enable safe hierarchical reconfigurations. The three extensions are (a) an adapted variant of the 2-phase-commit protocol to initiate reconfigurations that maintain component encapsulation, (b) the integration of feedback controllers during reconfiguration, and (c) a verification approach based on (timed) model checking for instances of our model. We illustrate our approach on a case study in the area of smart railway systems by showing two different use cases of our approach. We show that using our approach the systems can be easily designed to reconfigure safely.

Keywords CPS · Safe reconfiguration · Correctness-by-construction · Runtime reconfiguration · Component model · Reconfiguration behavior · Feedback controller exchange · Transactions · Atomicity · Consistency · Isolation · Timed model checking

1 Introduction

Cyber-physical systems (CPSs) are systems that operate in physical environments in real time but are driven by software. Examples of CPSs include smart cars, smart railway systems, or smart grids. As such systems interact with humans, they have to be safe. Safety is not easy to achieve as the concrete physical conditions these systems operate in are often highly dynamic and unknown at design time. Despite these environmental characteristics, CPSs shall operate at near-optimal conditions (e.g., with minimum use of resources) while never entering hazardous situations. One (technical) approach to achieve these objectives is to safely adapt the systems behavior to match its current environment by restructuring the systems software architecture.

This software architecture typically consists of software components and their connections. These components encapsulate their implementation to gain better maintainability, reusability, and analyzability. They can be composed either horizontally, e.g., components on the same hierarchy level

Communicated by Dr. ' F. Ciccozzi, J. Carlson, P. Pelliccione, and M. Tivoli.

✉ Christian Heinzemann
christian.heinzemann@de.bosch.com

Steffen Becker
steffen.becker@informatik.tu-chemnitz.de

Andreas Volk
andreas.volk@bosch-softtec.com

[1] Robert Bosch GmbH, Corporate Research, Renningen, Germany

[2] Technical University Chemnitz, Chemnitz, Germany

[3] Bosch SoftTec GmbH, Hildesheim, Germany

cooperate via messages to provide some functionality, or vertically, i.e., components embed reused components as children [1]. Particularly, the latter composition type requires that reconfiguration actions take into account this encapsulation. Hence, reconfiguration requests need to propagate recursively through the software components vertical hierarchy using properly defined component interfaces.

The focus of this paper is on the design of correct reconfigurations and their safe execution in real-time CPS that are composed of hierarchical components. Hence, reconfigurations in such systems have to fulfill three essential properties: First, they have to respect real-time properties (i.e., respect tight deadlines). Second, they have to execute according to ACI properties [2]: atomicity, i.e., either all or no reconfigurations throughout the vertical component composition have to execute; consistency, i.e., while and after executing a reconfiguration the system has to be in a consistent state; and isolation, i.e., reconfigurations need to be mutually exclusive so that one reconfiguration cannot interfere with another leading to invalid configurations. Third, a reconfiguration has to respect the continuous nature of the physical environment where discrete changes cannot be implemented. For example, we cannot stop the control system of the car for executing a reconfiguration while the car is still moving.

While many reconfiguration approaches for embedded, real-time systems take horizontal composition into account, most of them do not offer a comprehensive modeling and verification solution to take hierarchical composition, i.e., encapsulation, into account. As a consequence they lose the advantages of hierarchical compositions, i.e., reusing components to implement composed components and scalable hierarchical verifications. Such verification approaches verify each component separately, thereby relying on the verified child component properties. From those approaches which consider vertical composition [3,4], none fulfill the combination of real-time, ACI properties, and controller component support.

In this paper, we present an extension to our existing modeling language, MechatronicUML [5,6]. Our extension enables to specify platform-independent models of hierarchical reconfigurations. These models help software designers to control all aspects of designing reconfigurations in the context of a continuous physical environment which imposes additional requirements and constraints on the system and its timing. Our extension advances our previous article [7] on how to implement proper reconfiguration interfaces with ACI properties in our component model. Compared to what we presented in [7], i.e., a verifiable reconfiguration protocol which guarantees ACI properties and real-time constraints by design, this paper additionally addresses the continuous physical environment. Interaction with this environment is performed by utilizing feedback controllers, i.e., components that try to get a target metric of the CPS as fast and as close as possible to a set value. For example, they control the current speed of a smart vehicle and take into account that acceleration and breaking takes a while in a real physical environment. We extended our reconfiguration actions, so that also reconfigurations are supported that have an impact on target values of reconfigured feedback controller components.

Fig. 1 RailCab prototype in scale 1:2.5 on the university test track

We illustrate our novel approach on a proof-of-concept case study in the area of smart railway systems. However, our example system, called a RailCab,[1] is a real-world research prototype of such a smart train (cf. Fig. 1) that has been built at the University of Paderborn. In our case study, we intentionally focus on a slice of the software of this system to be modeled and analyzed. In particular, we model and analyze the reconfigurations of its drive control logic. This logic may be requested to enter convoy mode by adapting the speed controller to rely on external speed information and the distance to the train ahead instead of a manually defined target speed. The RailCab's software is also a case study for a reconfiguration which impacts a controller's set value (i.e., the target speed). For both use cases, we show their realization in our model and we demonstrate the model's correctness via (timed) model checking. Throughout the remainder of this paper, we will use the RailCab case study as a running example to illustrate our concepts. Therefore, it is described in the course of the paper, without having a dedicated section.

The paper is structured as follows. Section 2 introduces the MechatronicUML design method that is the basis for our contributions. Section 3 gives an overview on our approach to extend MechatronicUML. The following sections detail this approach. Section 4 introduces reconfiguration controllers, Sect. 5 our reconfiguration protocol, and Sect. 6 our declarative modeling concepts to ease the task of the software

[1] http://www.railcab.de.

architect. Section 7 explains how the declarative specification is translated by a model transformation into the system's implementation which is then verified as illustrated in Sect. 8. Section 9 contains remarks on our proof-of-concept tool support. Section 10 lists remaining assumptions and limitations of our approach. After a review of related work in Sect. 11, we conclude and outline future work.

2 MechatronicUML component model

MechatronicUML [5,8] is a domain-specific modeling language (DSML) for specifying the software of a CPS. MechatronicUML specifically targets the specification of (i) a component-based software architecture, (ii) the real-time behavior of the components, and (iii) the reconfiguration of the software architecture at run time. MechatronicUML distinguishes between platform-independent models (PIM) and platform-specific models (PSM) as defined by the model-driven architecture approach [9]. In this paper, we focus on the PIM level.

In this section, we introduce the basic parts of the MechatronicUML component model [8,10] that are required for the contribution of this paper and refer to the given literature for a complete definition. The MechatronicUML component model enables to specify a component-based software architecture including the reconfiguration of the software architecture at run time.

In the remainder of this section, we first review the specification of components (Sect. 2.1) and their instantiation to component instances (Sect. 2.2). Thereafter, we describe the specification of reconfigurations of the software architecture based on component story diagrams (Sect. 2.3) and the definition of architectural constraints (Sect. 2.4). Along with the component model, we also introduce the main parts of the RailCab model that we will use as a running example throughout the remainder of this paper.

2.1 Components

"A [..] *component* is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard." [11, p. 7] In accordance with UML [12], components are either implemented directly or assembled from other components, thereby forming a hierarchy of components. We refer to the former as *atomic components* and to the latter as *structured components*. In both cases, the internals of a component are hidden from the outside world. This is denoted as component encapsulation [1]. Access to the capabilities or data of a component is only allowed via its ports.

Figure 2 shows a slightly simplified version of the structured component RailCabDriveControl that implements the driving functionality of the RailCab. For a full version of the example, we refer to [10]. The behavior of RailCabDriveControl is defined by the concurrent execution of the six embedded components. All of the referenced components are atomic components except the VelocityController, which we show in detail in Fig. 3.

The MechatronicUML component model distinguishes atomic components and ports based on their purpose and the information they process or exchange. We introduce these kinds of atomic components and ports based on our running example in the following.

The embedded components ConvoyCoordination, MemberControl, and OperationStrategy are *discrete atomic components*. Discrete atomic components define the discrete, event-based real-time behavior of the system. As a result, a discrete component operates on time-discrete values and implements message-based communication. In our example, ConvoyCoordination and MemberControl implement the behavior of a RailCab being a coordinator or member of a convoy, respectively. The behavior of discrete components is defined by hierarchical state machines called *real-time statecharts* (RTSCs). In essence, they are UML state machines [12] that are extended by the clock concept of UPPAAL timed automata [13]. RTSCs are event-triggered and their operational semantics is defined by a mapping to UPPAAL timed automata [14].

The embedded components SpeedSensor and DistanceSensor are *continuous atomic components*. Continuous components represent the sensors, actuators, and feedback controllers of the system. Thus, they operate on time-continuous signals and their correctness depends on the physical behavior of the controlled system. In our example, SpeedSensor and DistanceSensor are sensors that measure the current speed of the RailCab and its distance to a preceding RailCab. The behavior of continuous components is specified in a control engineering tool such as MATLAB/Simulink.[2]

In our example, we use three kinds of ports for connecting components. These are discrete ports, continuous ports, and hybrid ports. Discrete ports send and receive asynchronous messages and enable to implement communication protocols for the interaction of different components and systems. Therefore, they define messages that they may send and receive including a protocol state machine specified by an RTSC. The semantics of discrete ports is defined by the operational semantics of the RTSC, i.e., via timed automata. In our example, the ports coordinator and member of RailCabDriveControl are discrete ports that implement the communication protocol for the convoy drive. The small embedded triangles indicate the communication direction. Continuous and

[2] http://www.mathworks.com/products/simulink.

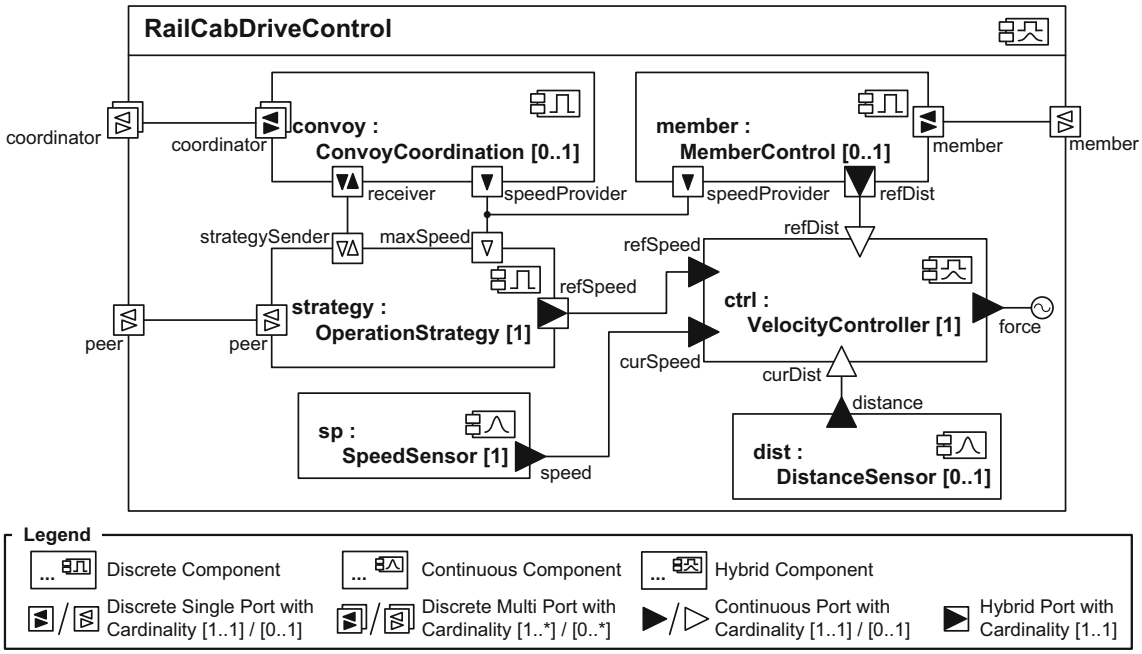

Fig. 2 Component type RailCabDriveControl

hybrid ports send or receive a time-continuous signal. Hybrid ports sample the time-continuous signal with a fixed period to enable interaction between discrete components and continuous components. In our example, the port speed of SpeedSensor is a continuous port while refSpeed of OperationStrategy is a hybrid port that enables to set the reference speed of the VelocityController. Continuous ports connected to a circle with an embedded sine wave such as force of ctrl in Fig. 2 are directly connected to the digital hardware upon deployment.

Ports define a cardinality that defines the minimum and maximum number of instances that a component may have of that port. Ports are optional if the lower bound of the cardinality is 0. Optional ports are visualized with unfilled triangles such as member of RailCabDriveControl or refDist of VelocityController. Ports with an upper bound greater than 1 are called multi-ports. They are visualized with a cascaded border such as coordinator of RailCabDriveControl.

Figure 3 shows the structured component VelocityController. The VelocityController embeds three components. The continuous component StandaloneDrive implements the feedback controller for driving alone or as a coordinator of a convoy. It controls the speed of the RailCab only based on the current speed received via curSpeed and a reference speed received via refSpeed. The continuous component ConvoyDrive implements the feedback controller for driving as a member of a convoy. It additionally considers the distance to the preceding RailCab for controlling the speed.

Finally, ConvoyFading is a so-called fading component. A fading component enables to switch between continuous component instances as part of a reconfiguration (cf. Sect. 5.2 for details) if they produce the same output signal such as force in Fig. 3. In general, continuous component instances must not be replaced instantaneously because this may cause a jump in the value of the controlled variable force at the motor that may damage it. For preventing such jumps, a fading component implements fading functions based on cross-fading [15] or flatness-based switching [16] for smoothing the output signal while replacing continuous component instances. The fading functions are implemented directly in a control engineering tool as for continuous components.

2.2 Instantiating components to component instances

The components introduced in Sect. 2.1 are instantiated to stateful *component instances* for defining a software architecture of a system. Components may be instantiated multiple times in a system. In particular, each structured component instance creates its own instances for the components that are embedded by the component parts. Upon instantiation, the number of port instances for each port, the number of embedded component instances for each component part, and the connector instances need to be determined. By default, all ports and component parts are instantiated with minimum cardinality and may not exceed the maximum cardinality when reconfigured [10].

Each component instance has a configuration that is defined by its currently instantiated port instances, embedded component instances, and connector instances. The *component instance configuration* (CIC) of the system [8] is defined by the configurations of all component instances.

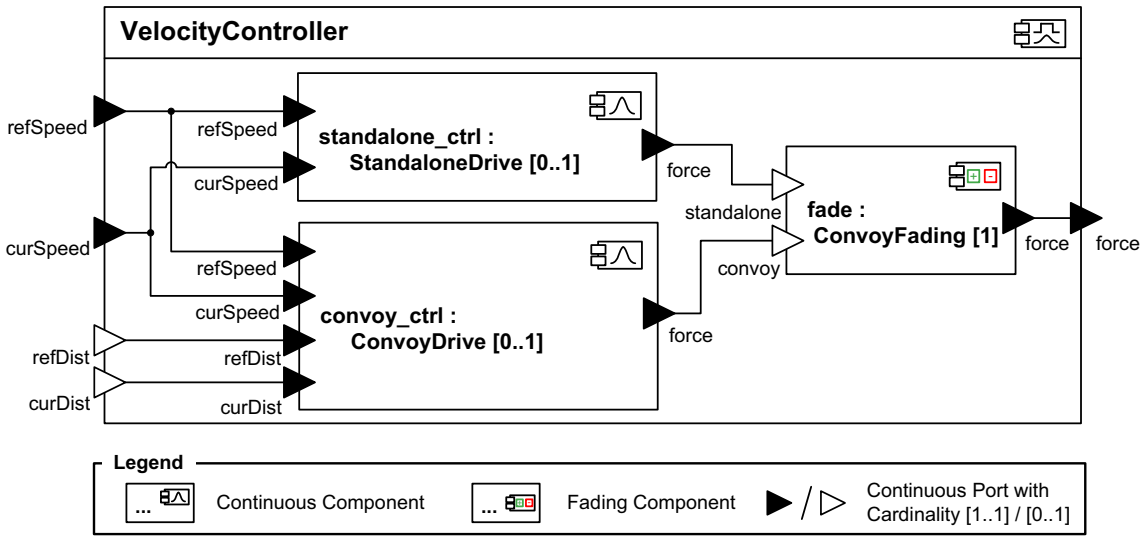

Fig. 3 Component type VelocityController

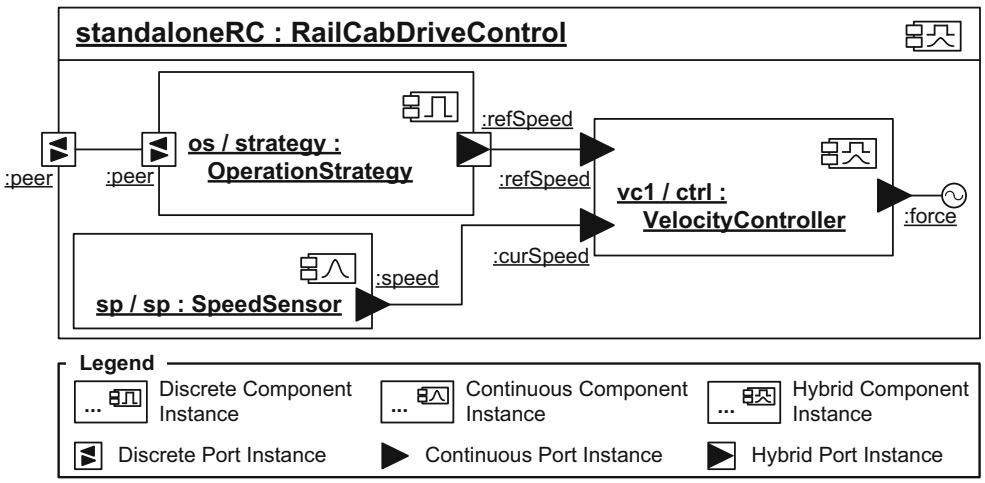

Fig. 4 Component instance of component RailCabDriveControl for a RailCab driving alone

Figure 4 shows the CIC of RailCabDriveControl for a RailCab driving alone. The standaloneRC only embeds three component instances: os of type OperationStrategy, vc1 of type VelocityController, and sp of type SpeedSensor. Consequently, the reference speed for the RailCab is defined solely by os and provided to vc1. vc1 controls the force of the electric motor only based on this reference speed and the current speed of the RailCab provided by sp. Thus, vc1 only uses the feedback controller implemented in StandaloneDrive (cf. Fig. 3).

In the following, we introduce a second CIC of RailCabDriveControl for a RailCab driving as a convoy member. We introduce this CIC because we use the corresponding reconfiguration from driving alone to driving as a convoy member as a running example throughout the remainder of this paper. The Member shown in Fig. 5 embeds five component instances. In addition to standaloneRC, Member has instances mc of type MemberControl and ds of type DistanceSensor. mc is connected to the coordinator of the convoy via its member port and propagates the reference speed of the convoy to os. In addition, it sets the reference distance to vc2. The instance vc2 of VelocityController now executes the feedback controller implemented in ConvoyDrive and, thus, controls the force of the electric motor based on the reference speed and the reference distance. The current distance to the preceding RailCab is provided by the instance ds of DistanceSensor.

2.3 Specifying reconfigurations

In our approach, we distinguish between *functional behavior* and *reconfiguration behavior* [17,18]. The functional behavior is defined as the behavior that is executed by the current CIC. The reconfiguration behavior defines possible modifications of the CIC that may be executed at run time. During

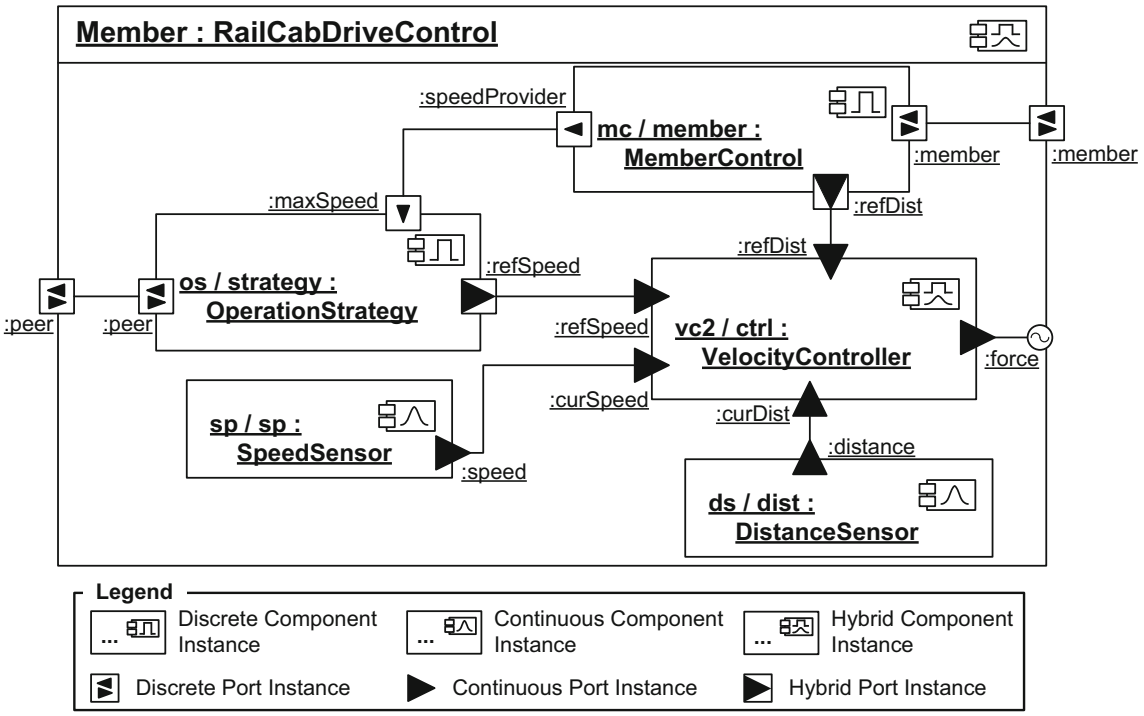

Fig. 5 Component instance of component RailCabDriveControl for a RailCab driving as a convoy member

a reconfiguration, the system switches from one functional behavior to another functional behavior [18].

We specify reconfigurations using *component story diagrams* (CSDs, [19]). Each component can contain a set of CSDs that define possible reconfigurations of the component. The allowed modifications are the addition and removal of embedded component instances, of ports, and of connectors. Essentially, CSDs are UML activity diagrams [12] where each action describes a step of the reconfiguration. The semantics of the actions is formally defined based on graph transformations [20]. By construction, CSDs ensure that component instances remain syntactically correct after applying a reconfiguration because a CSD can only be executed if its modifications do not violate the cardinalities of ports and component parts.

For the specification of the graph transformation, CSDs use a short-hand notation that depicts left-hand side and right-hand side of the graph transformation in a single, annotated graph (cf. Fig. 6). Unmodified elements are depicted in black without further annotations. Created elements are depicted in green and carry the annotation «create», while deleted elements are depicted in red and carry the annotation «destroy». All variables and links of the actions are typed by the components, ports, and connectors that are defined by the component model. Each action contains exactly one this component variable. At run time, the this variable is automatically bound to the component instance that invoked the CSD on itself [19].

Figure 6 shows the CSD becomeMember of the component RailCabDriveControl. The CSD reconfigures an instance of RailCabDriveControl of a RailCab driving alone (cf. Fig. 4) to an instance of a RailCab driving as a member of a convoy (cf. Fig. 5).

The CSD has two actions. In the first action, we match the embedded component instances of types OperationStategy and VelocityController. We invoke a reconfiguration applyMemberStrategy on os that creates the maxSpeed port instance. The invocation of the CSD is directly attached to the corresponding component variable. In addition, we invoke the reconfiguration switchToConvoy on vc that reconfigures the feedback controllers for driving as a convoy member. We introduce this CSD in more detail below. In the second action, we create an instance of MemberControl. In addition, we create an instance of DistanceSensor and connect it to vc by an assembly connector instance. Finally, we create a port instance of member on this and connect all port instances of mc.

In this paper, we define a method that explicitly restricts reconfigurations such that they respect component encapsulation. In particular, we forbid that a CSD directly creates or destroys port instances of its embedded component instances. Such port instances may only be created by the embedded component instance itself. Therefore, a corresponding CSD that creates the port instance needs to be invoked on the embedded component instance as shown in Fig. 6. Since we need to create these port instances before connecting them,

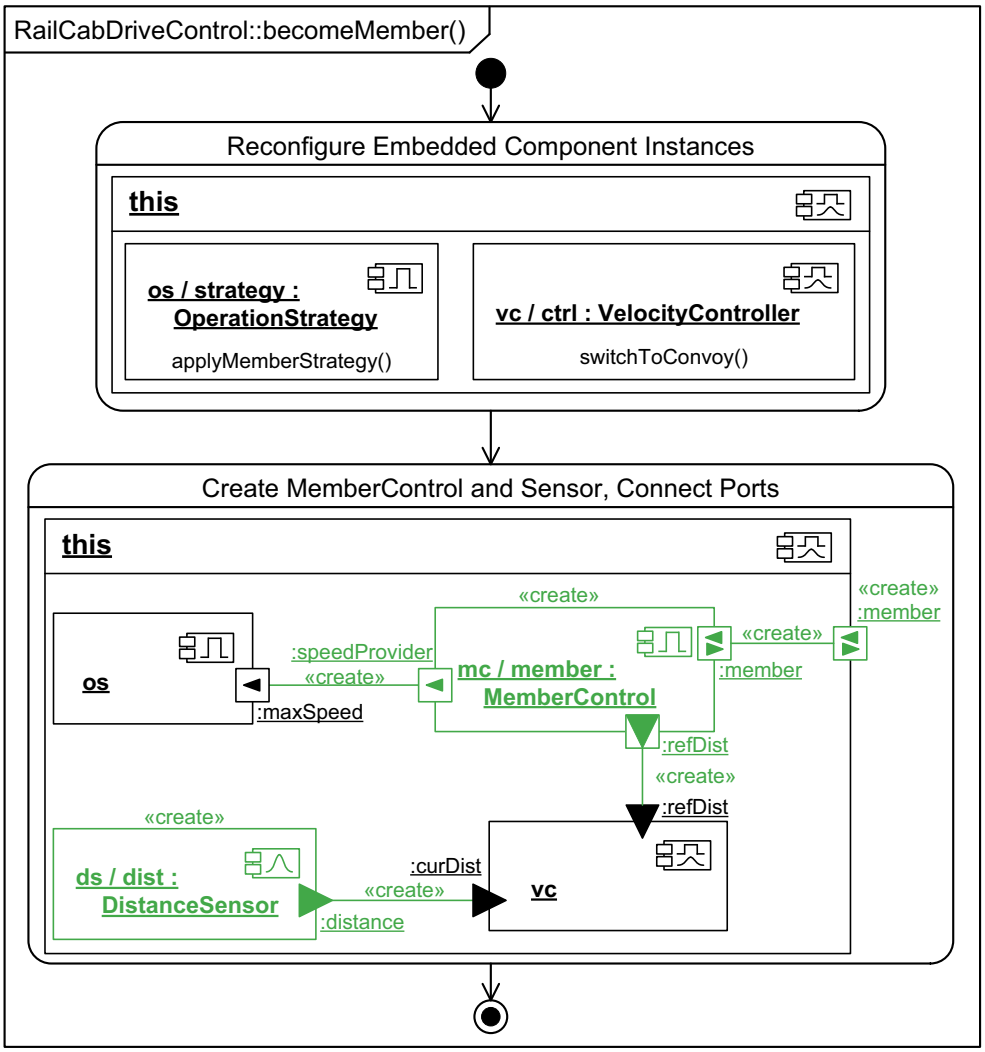

Fig. 6 Component story diagram specifying the reconfiguration becomeMember of RailCabDriveControl

we need to call the corresponding CSDs before creating the connector.

In contrast to discrete components, we cannot instantaneously replace feedback controllers because this may cause a discontinuity in the controlled value. This discontinuity, in turn, may damage the physical system because, in our example, the force being set to the electric motor would be too high. As a solution, we need to fade smoothly from the outputs of the destroyed continuous component to the outputs of the created continuous component. This, in turn, requires transferring the internal state that is contained in the control algorithm. Therefore, we need to wait until the feedback controller in the created continuous component has correctly initialized, based on the current values and the reference value. The fading to the outputs of the new continuous component is then performed by a fading function [15,16] that is executed by the fading component f shown in Fig. 7. The main design challenges for the control engineer when developing such fading functions are guaranteeing stability of the control algorithms and finishing in time.

In CSDs, we use controller exchange actions for specifying the replacement of continuous components. As an example, Fig. 7 shows the CSD switchToConvoy of the VelocityController component that is invoked by becomeMember. In essence, the CSD replaces the instance of StandaloneDrive by an instance of ConvoyDrive. In our example, initializing the ConvoyDrive and executing the fading function takes 150 ms to 180 ms.

2.4 Defining architectural constraints

An architectural constraint defines a condition on the configurations of a (structured) component instance [21]. Architectural constraints may be defined as invariants or conditions. An *invariant* needs to evaluate to true for any configuration of the component, whereas a *condition* may evaluate to false for

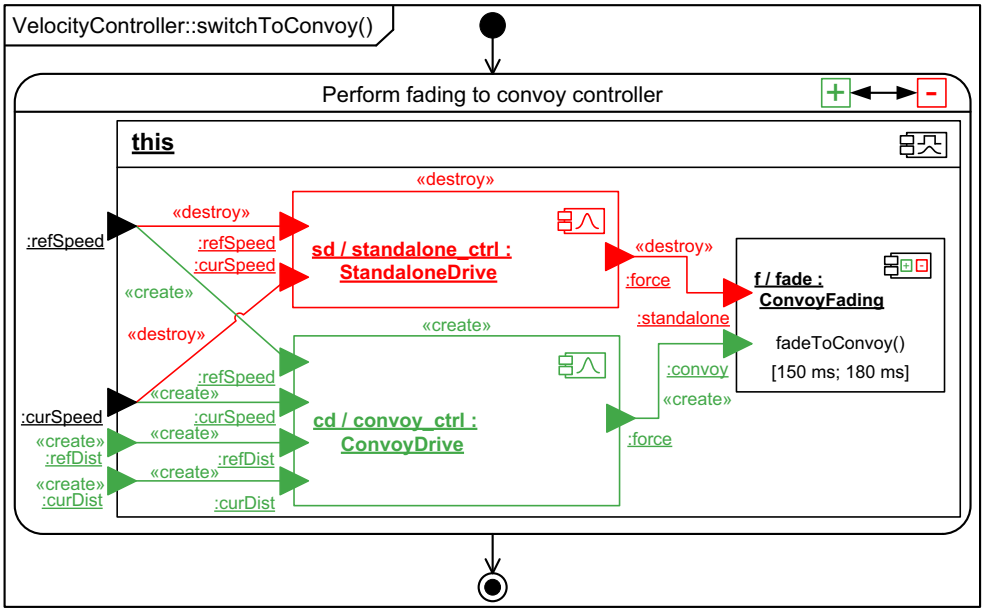

Fig. 7 Component story diagram specifying the reconfiguration switchToConvoy of VelocityController

some configurations. Invariants enable to define valid configurations of a component, which we exploit for verifying the correctness of the reconfiguration behavior in Sect. 8. Conditions enable to restrict the applicability of reconfigurations.

In the MECHATRONICUML component model, we use *component story decision diagrams* (component SDDs, [22]) for modeling architectural constraints. They are, in essence, a syntactically restricted form of CSDs that always return a Boolean value. Therefore, we omit a detailed description of component SDDs in this paper and refer to our technical report [22].

3 Overview of our approach

Reconfigurations in a hierarchical component model often require the reconfiguration of several components that are located on different levels inside the hierarchy. As an example, the reconfiguration of a structured component instance may require the upfront reconfiguration of one or more of its children as it has been shown in Fig. 6. In this example, creating the instance mc of MemberControl requires reconfiguring the instances of OperationStrategy and VelocityController first. Then, the port instances created by these reconfigurations are connected by RailCabDriveControl. In general, we distinguish two use cases for such reconfigurations.

In **Use Case 1**, an embedded component instance, in the following referred to as *child*, detects a situation that requires a reconfiguration that it cannot handle solely by itself. In our example in Fig. 2, the OperationStrategy component negotiates that the RailCab enters a convoy, but it does not know how to do this itself. Thus, it needs to send a request to the embedding structured component instance of type RailCabDriveControl to handle that situation and to execute the necessary reconfiguration. We will refer to the embedding structured component instance as *parent* in the following.

In **Use Case 2**, a structured component instance executes a reconfiguration that requires the reconfiguration of one or more of its children. In our example, becoming a member of a convoy requires a reconfiguration of the RailCabDriveControl (cf. Fig. 5). Executing this reconfiguration, however, requires that the OperationStrategy changes its port instances and that the VelocityController switches to the ConvoyDrive component instance. Therefore, RailCabDriveControl needs to trigger the corresponding reconfigurations on its children.

For both use cases, executing such reconfigurations safely demands that all component instances, which need to reconfigure, perform their reconfiguration in a coordinated way. The necessary conditions for executing a hierarchical reconfiguration safely are given by the *ACI properties* (atomicity, consistency, and isolation) of database systems [2,23], a correct timing, and, if necessary, correct fading functions. A correct timing demands that if a (hard) deadline for executing a reconfiguration exists, the system shall only start the reconfiguration if it is able to finish it before the deadline. Therefore, the deadline puts an upper bound on the maximum duration that the reconfiguration may take. Fading functions are integrated as "black box" into the execution of a reconfiguration as explained in Sect. 5 such that we only need to consider its timing properties for the specification and analysis of the behavior of the reconfiguration controller. As a result, we need to consider ACI properties plus timing, referred to as *ACI-T properties*, for the remain-

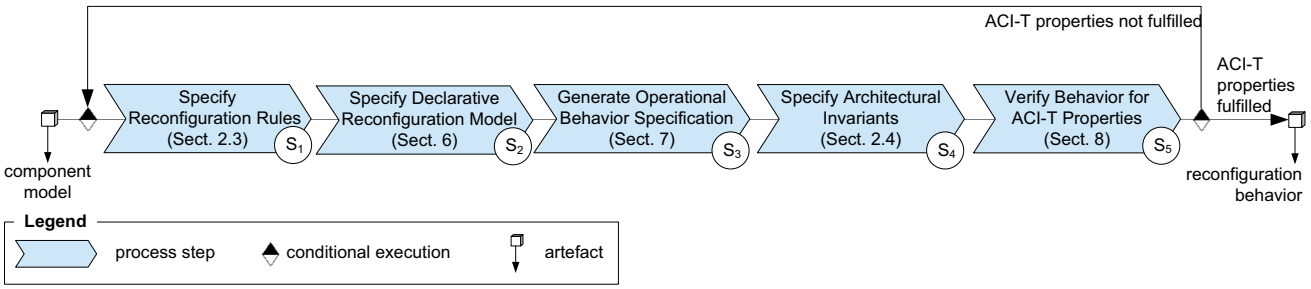

Fig. 8 Process for specifying reconfiguration behavior (cf. [24])

der of this paper. If a reconfiguration is executed according to ACI-T properties, we denote this as *transactional execution*.

For realizing transactional execution of reconfigurations, we adapt the 2-phase-commit protocol for distributed database systems [2, ch. 7] to the domain of CPS. To this end, we focus on structured components in this paper. In accordance with the 2-phase-commit protocol, a structured component instance asks all children that are required to reconfigure whether they can execute the required reconfiguration *before* starting the reconfiguration. Only if all children confirm and if the reconfiguration can be finished in time, the reconfiguration is started and the children are notified to execute their reconfiguration.

In contrast to related approaches, we may not start to reconfigure optimistically and roll back to a preexisting configuration if the reconfiguration fails as, for example, proposed for reliable reconfiguration of Fractal components in [23]. This is for two reasons: First, the system might come into an inconsistent state that causes it to malfunction if a reconfiguration is only executed partially. Second, it is not guaranteed that returning to the configuration before reconfiguration has started is even possible and safe.

Figure 8 summarizes our process for specifying reconfigurations based on our variant of the 2-phase-commit protocol [24]. In the first Step S_1, the developer specifies the reconfiguration rules using CSDs as introduced in Sect. 2.3. Thereafter, the developer creates a declarative, table-based specification of hierarchical reconfigurations in Step S_2. These tables define in which situation which CSD is to be executed, but they relieve the developer from specifying how the reconfiguration is carried out [7]. In addition, these tables define timing requirements that formalize our notion of a correct timing mentioned above. Then, we automatically generate an operational behavior specification based on RTSCs from the declarative table-based specification in Step S_3. The operational behavior specification additionally specifies *how* reconfigurations are executed based on the 2-phase-commit protocol. In Step S_4, the developer specifies architectural invariants (cf. Sect. 2.4) that define the set of valid configurations for instances of a component. In Step S_5, we use the architectural invariants as well as the generated operational behavior specification for verifying at design time that the reconfiguration specification fulfills ACI-T properties. With respect to timing, we verify the necessary conditions for a correct execution w.r.t. the timing requirements that are contained in our declarative specification. In Sect. 4 and 5, we describe the foundations of our modeling approach. In the subsequent sections, we then describe the introduced method steps.

4 MechatronicUML reconfiguration controller

In the MechatronicUML component model as introduced in Sect. 2, a developer defines a set of CSDs that specify the possible reconfigurations of a component. This does not enable to specify in which situation which reconfiguration is to be executed. In addition, CSDs offer no means for executing a reconfiguration hierarchically according to ACI-T properties while preserving component encapsulation.

As a solution, we syntactically extend each reconfigurable component with a dedicated *reconfiguration controller* that is inspired by the reconfiguration controller of the Fractal component model [25,26]. While these controllers might be specified by developers manually, we aim at generating them as described in Sect. 7. Our reconfiguration controller as shown in Fig. 9 introduces two syntactic elements, namely a *manager* and an *executor*. The executor is responsible for executing reconfigurations respecting hierarchy and ACI-T properties based on the 2-phase-commit protocol. The manager decides which reconfiguration is executed in which situation, which is not supported by the Fractal reconfiguration controller. By using a dedicated reconfiguration controller, we retain separation of concerns between functional behavior and reconfiguration behavior as advised by McKinley et al. [17].

In addition, we introduce two new port types called *reconfiguration message ports*, short RM ports, and reconfiguration execution ports, short RE ports. A component uses its RM ports for sending information in situations that may require a reconfiguration to its parent. Consequently, RM ports are

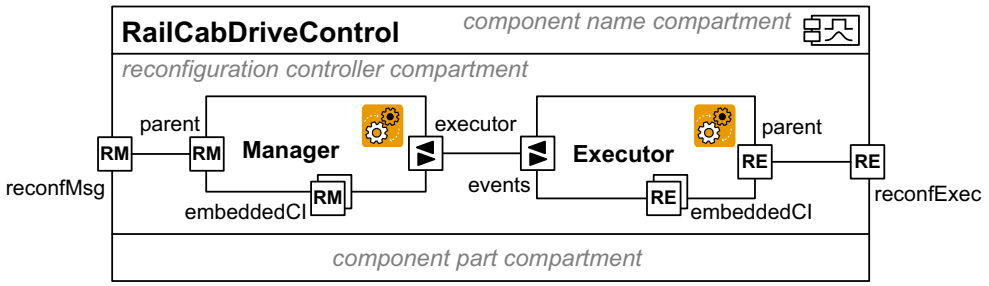

Fig. 9 Reconfiguration controller of a structured component (cf. [7])

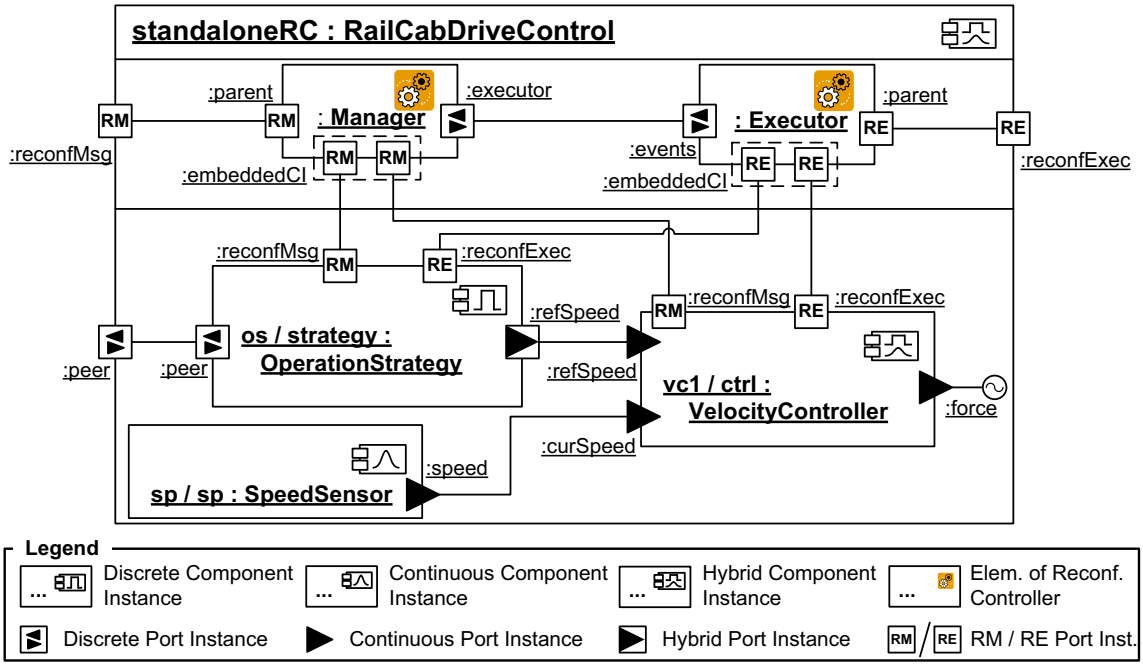

Fig. 10 Component instance RailCabDriveControl with reconfiguration controller

used for bottom-up information passing and to provide the necessary message flow for realizing Use Case 1. A component uses its RE port for offering reconfigurations to its parent. The parent may trigger a reconfiguration on a child by sending a message to the RE port of that child. Thus, RE ports are primarily used for top-down reconfiguration initiation and to provide the necessary message flow for realizing Use Case 2.

RM ports and RE ports are essentially discrete ports with an extended interface specification (cf. Sect. 6) that enables to execute reconfigurations across different levels of hierarchy without violating component encapsulation. This interface specification enables to generate the protocol state machine for RM ports and RE ports (cf. Sect. 7). Since they are discrete ports, they have the same operational semantics as discrete ports (cf. Sect. 2.1).

For enabling message flow across different levels of hierarchy at run time, we connect the manager (and executor) to the parent and all embedded component instances using the RM ports (or RE ports). Figure 10 illustrates these connections for an instance of the RailCabDriveControl component for driving alone.

As shown in Fig. 9, the manager specifies two RM ports named parent and embeddedCI. The RM port parent implements the RM port of the structured component and is used for sending messages to the parent. The RM multi-port embeddedCI connects the manager to the RM port instances of the embedded component instances for receiving their messages. At run time, one subport instance of this port exists for each child of the structured component instance as shown in Fig. 10. Since standaloneRC has three embedded component instances, the embeddedCI port of the manager contains three subport instances. The executor is connected to the parent and the embedded component instances in the same fashion.

Since the reconfiguration controller has the same structure for any structured component and introduces additional visual complexity, we typically use the short-hand notation

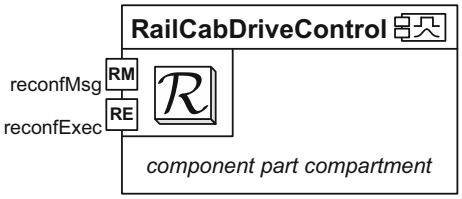

Fig. 11 Short-hand notation for reconfigurable components

shown in Fig. 11 for visualizing reconfigurable structured components [7].

For executing the reconfiguration at run time, the structured component needs to maintain a model@run.time [27] of its own architecture. For deployment, a platform-specific realization of the system needs to be implemented, i.e., by using a generator for a specific platform. This realization has to be a valid refinement of the platform-independent model, i.e., conformance of the timing specifications has to be verified. As this realization will execute on a distributed hardware platform later, the model@run.time needs to be maintained in a distributed fashion as well. This is facilitated by component encapsulation because encapsulation imposes that a component may only have information about itself and about its direct children. As a consequence, each structured component only needs to maintain a small, local part of the model@run.time that contains information about its own instantiated ports, about its instantiated child components including their ports, and about the assembly and delegation connectors that connect the children. A discussion regarding the encoding and the management of the model@run.time, however, is beyond the scope of this paper. For more information on this subject, we refer to our paper [28] and the thesis [10, chap. 6]. There, we present an approach in MATLAB/Simulink for a distributed encoding of the model@run.time for an embedded system. The approach utilizes control signals for activating and deactivating component instances, port instance, and connector instances.

5 Executing reconfigurations

Using our reconfiguration controller, we can execute reconfigurations with respect to hierarchy considering our two use cases. As mentioned above, we provide a variant of the 2-phase-commit protocol [2, ch. 7]. The 2-phase-commit protocol starts with a *voting phase*. In the voting phase, a structured component instance queries all of its children, which are required to participate in the reconfiguration, whether they actually can reconfigure. The children then evaluate concurrently whether they can execute the requested reconfiguration or not. In case a child component is able to reconfigure itself in case a reconfiguration command would be issued by the component's parent until a certain deadline determined by the component itself, it sends a positive vote including its deadline to its parent. Only if all queried children reply with a positive vote, the structured component can execute the reconfiguration in the *execution phase*. If at least one children responds with a negative vote, the reconfiguration is aborted.

For executing a reconfiguration in the execution phase of our 2-phase-commit protocol, we need to distinguish between purely discrete reconfigurations and reconfigurations that involve continuous components. In the former case, all affected children need to be quiescent as explained in Sect. 5.3 and, therefore, we may reconfigure the system bottom-up in a single pass as explained in Sect. 5.1. We refer to this as *single-phase execution*. If the reconfiguration replaces continuous components, we need to execute fading functions (cf. Sect. 2.1). These fading functions require that all port instances of the destroyed and created continuous component instance are properly connected. This requires splitting the execution phase into three subphases. We refer to this as *three-phase execution* as explained in Sect. 5.2. In general, single-phase execution is faster and requires less messages to be exchanged between the executor of a structured component instance and the executors of the children. Therefore, single-phase execution should be preferred whenever possible.

5.1 Single-phase execution

Using single-phase execution, the reconfiguration of a structured component instance is performed in a single, bottom-up pass over the component hierarchy. That means, we start with the children that are nested at the deepest level of the hierarchy. The reconfiguration of a structured component instance is then executed after the concurrent execution of the reconfigurations of all children. In the following, we describe the message flow and responsibilities in our reconfiguration controller for realizing the two use cases mentioned above with our 2-phase-commit protocol and single-phase execution.

Figure 12 illustrates Use Case 1 (i.e., a reconfiguration triggered by a child component) for an instance dc of RailCabDriveControl (cf. Fig. 10). First, the OperationStrategy component instance sends a message via its RM port to the Manager, requesting the reconfiguration for becoming a convoy member. Then, the Manager decides whether to execute the reconfiguration and, if so, triggers the executor. The executor initiates the 2-phase-commit protocol and collects the votes of the children as outlined above. In our example, the instances of VelocityController and OperationStrategy are affected by the reconfiguration. If at least one child sends a negative vote, the reconfiguration will be aborted. If a child sends a positive vote, it provides a commit time bound. The commit time bound denotes how long after the end of the voting the child can assure to execute the reconfigu-

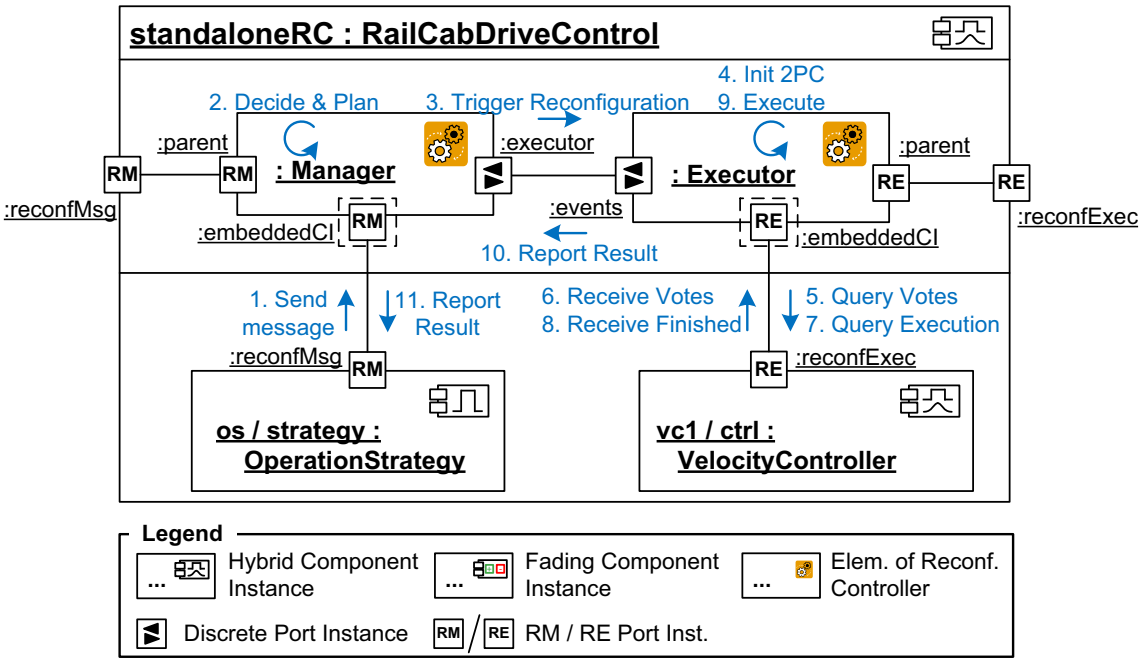

Fig. 12 Use Case 1: Reconfiguration after child request (cf. [7])

ration successfully. After that time, the child is no longer bound to its vote (cf. Sect. 6.4). If all children have voted, the executor computes the minimum of all commit time bounds provided by the children. This time is relevant as after passing the minimum commit time bound, the component that replied with that time bound can no longer be assumed to successfully complete a reconfiguration command. Having the minimum commit time bound, the executor can decide whether a reconfiguration is feasible or not. In case the maximum time for executing the reconfiguration is less than the minimum commit time bound, the executor knows for sure that the reconfiguration is feasible in the available time and therefore commands all children to execute their reconfigurations. We provide details on the computation of these time values in Sect. 8.2. After all children have finished, the executor performs the reconfiguration of the structured component instance itself and reports the result to the manager. Since the reconfiguration originated from a request of a child, the result is reported to that child, i.e., OperationStrategy in our case study.

Figure 13 illustrates Use Case 2 in the same fashion. Continuing our example, we consider the VelocityController instance and assume that it is triggered by its parent for switching into member mode. The VelocityController receives the message via its RE port. The message is propagated to the manager which decides upon the request. The manager then reports the decision to the executor. If the manager has decided not to execute the reconfiguration, the executor immediately sends a negative vote to the parent. If the manager has decided to execute the reconfiguration, the executor initiates the 2-phase-commit as in Use Case 1. After it has collected the votes of the children, it checks whether the reconfiguration can be executed in time using the commit time bounds of the children. Then, it sends the resulting vote to the parent. After sending the vote, the executor waits for the answer of the parent, but no longer than the commit time. If the parent decides to execute (or abort), the executor commands the execution (or abortion) of the reconfiguration on the children. After all children have finished, the executor performs the reconfiguration of the structured component and reports to its parent that the reconfiguration has been finished.

The use cases for reconfiguration are designed such that they propagate recursively to all children. If a structured component reconfigures based on Use Case 1 and invokes a child, that child reconfigures based on Use Case 2. If the child is a structured component instance itself and needs to invoke reconfigurations of its children as well, Use Case 2 propagates recursively. For an atomic component, only Use Case 2 may occur.

5.2 Three-phase execution

Single-phase execution of reconfigurations as described in the previous section cannot be applied if the reconfiguration involves replacing continuous components due to the issue of the discontinuities in the controlled variable. In addition, in order to execute fading functions, we need to respect the correct order of instantiating and deleting components as detailed below. As a solution to this problem, we split the

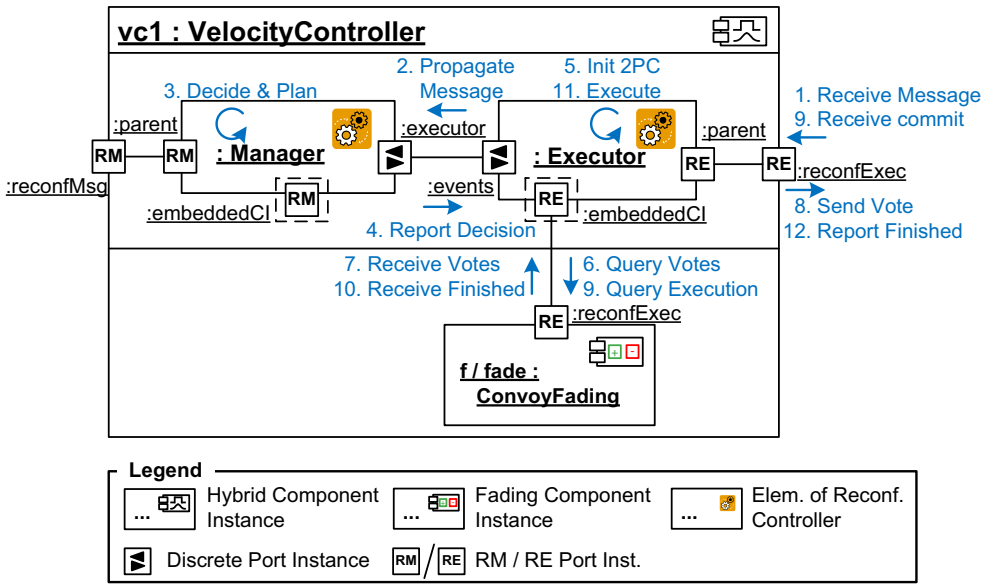

Fig. 13 Use Case 2: Reconfiguration as part of 2-phase-commit (cf. [7])

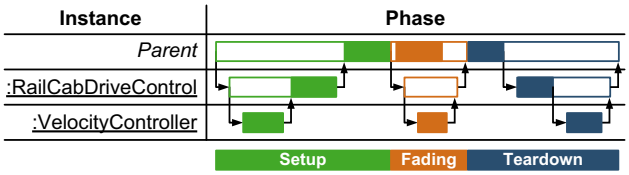

Fig. 14 Illustration of three-phase execution

execution phase of our 2-phase-commit protocol into three subphases if the reconfiguration replaces continuous components. These subphases are *setup*, *fading*, and *teardown*. Each of these subphases executes part of the reconfiguration.

Figure 14 illustrates how these subphases are executed in a structured component instance. A filled bar denotes that the instance is currently executing reconfiguration behavior, while an unfilled bar denotes that the instance is idle. The arrows denote the points in time where the parent triggers the child to start the corresponding phase (downwards arrow) and when the child execution returns (upwards arrow). The actual part of the reconfiguration that needs to be executed in each phase can be derived automatically from the CSDs.

In our case study, we needed to consider the reconfiguration for becoming a convoy member that requires the VelocityController to switch from the StandaloneDrive to the ConvoyDrive feedback controller (cf. Fig. 6).

Figure 15 shows an intermediate CIC that would occur if we executed this reconfiguration according to single-phase execution. In particular, this CIC occurs when rc1 queries vc1 to execute its reconfiguration. As a result, vc1 contains an instance of ConvoyDrive and it currently executes the fadeToConvoy fading function in the fading component. At this point in time, both continuous component instances are executed in parallel. However, the ConvoyDrive instance will not properly work because the input ports refDist and curDist have no defined values. The reason is that these port instances are delegated by vc1 and need to be connected in rc1 before starting to execute the fading function. In particular, we need to create instances of the SpeedSensor and the MemberControl in rc1 prior to executing the fading function. This issue is solved by the setup and teardown phases as detailed in the following Sects. 5.2.1 to 5.2.3. The voting phase of the 2-phase-commit is executed as for single-phase execution (cf. Sect. 5.1) and will not be described here.

5.2.1 Setup

The three-phase execution starts with the setup phase. The setup phase (hierarchically) reconfigures a component instance such that all preconditions for executing the fading functions are established. Therefore, it changes the software architecture of the CPS, but it does not yet change the exhibited behavior of the CPS. The setup phase is executed bottom-up as shown in Fig. 14, i.e., a component first triggers its children concurrently and executes its own setup behavior after all children are finished.

In the setup phase, each component instance that is affected by the reconfiguration creates all discrete, continuous, and hybrid port instances as specified by the reconfiguration rule. In addition, structured component instances create all embedded component instances and all connector instances between continuous and hybrid port instances. Discrete component instances and ports are kept in a suspended mode, i.e., their RTSCs are not being executed and their clocks do not yet progress. All hybrid port instances

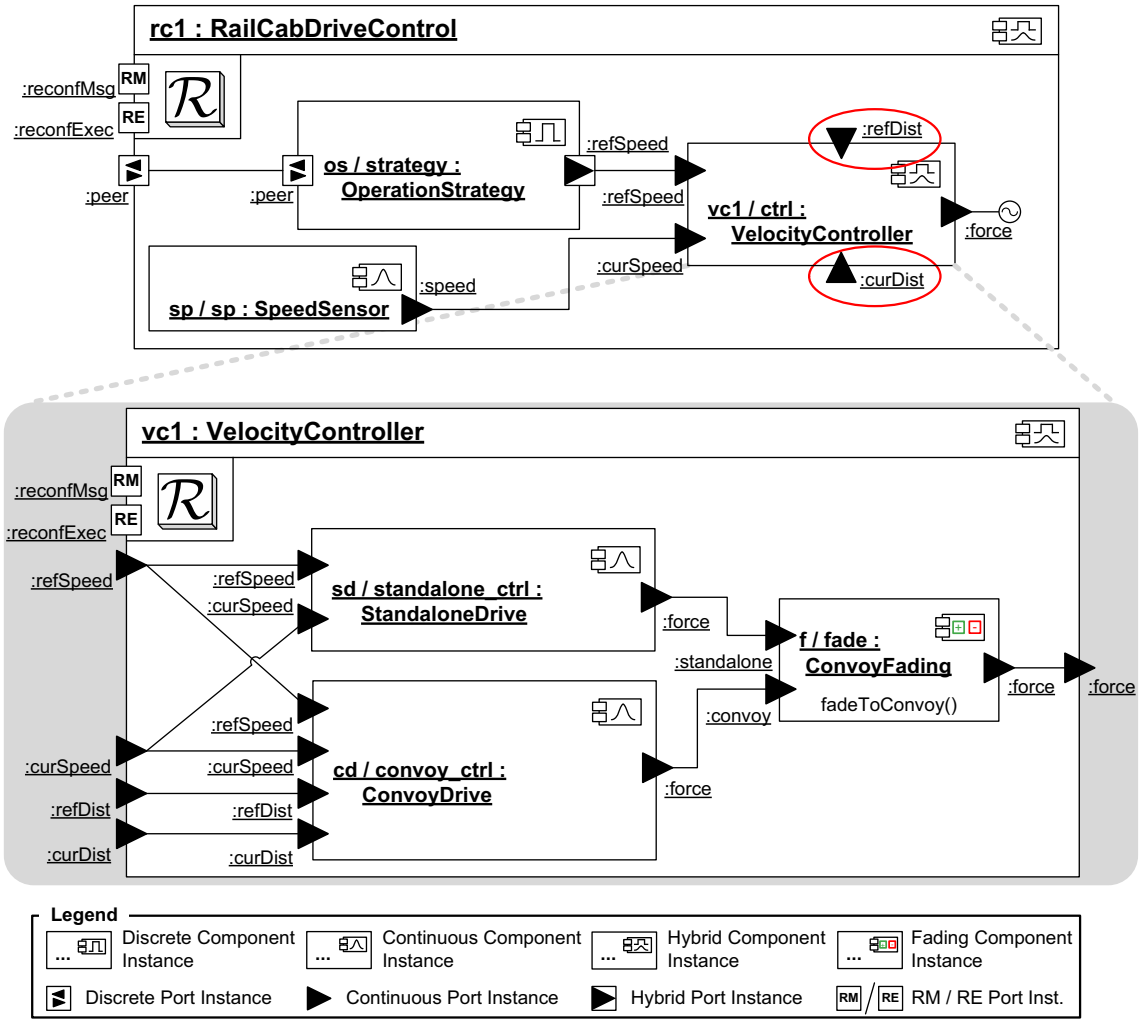

Fig. 15 Problems when replacing continuous component instances using single-phase execution

that have been created during setup already emit their default value though. All affected fading components still forward the unmodified value of the continuous component that is to be replaced.

Figure 16 shows component instances of RailCabDriveControl and VelocityController after performing the setup phase for the reconfiguration becomeMember shown in Fig. 6.

Since the setup phase is executed bottom-up, the execution starts at vc1. vc1 creates an instance of ConvoyDrive including the port instances refDist and curDist. In addition, it delegates all in-ports to the corresponding port instances of the new component instance cd. Finally, it creates the port convoy at the fading component including the assembly instance. As a result, vc1 contains all necessary component instances, port instances, and connector instances for executing the fading function.

After vc1 finished, rc1 executes its setup phase. According to the CSD in Fig. 6, rc1 creates instances of DistanceSensor and MemberControl. Since MemberControl is a discrete component, the instance mc remains suspended and only emits the default reference distance via its hybrid refDist port instance. In addition, rc1 creates the port instance member, but it does not yet create the corresponding delegation instance. Finally, rc1 creates the assembly connector instances between the continuous and hybrid port instances for connecting DistanceSensor and MemberControl to vc1.

As it can be inferred from Fig. 16, all in-ports of vc1 are properly connected. Thus, the fading function can now be executed and provide a meaningful result. Up to now, the behavior of rc1 and vc1 has not been changed because vc1 still emits the force value of sd and because the discrete connector instances in rc1 have not yet been modified and MemberControl is still suspended.

5.2.2 Fading

In the fading phase, we execute all fading functions in parallel as shown in Fig. 14. Thus, the behavior of the CPS changes,

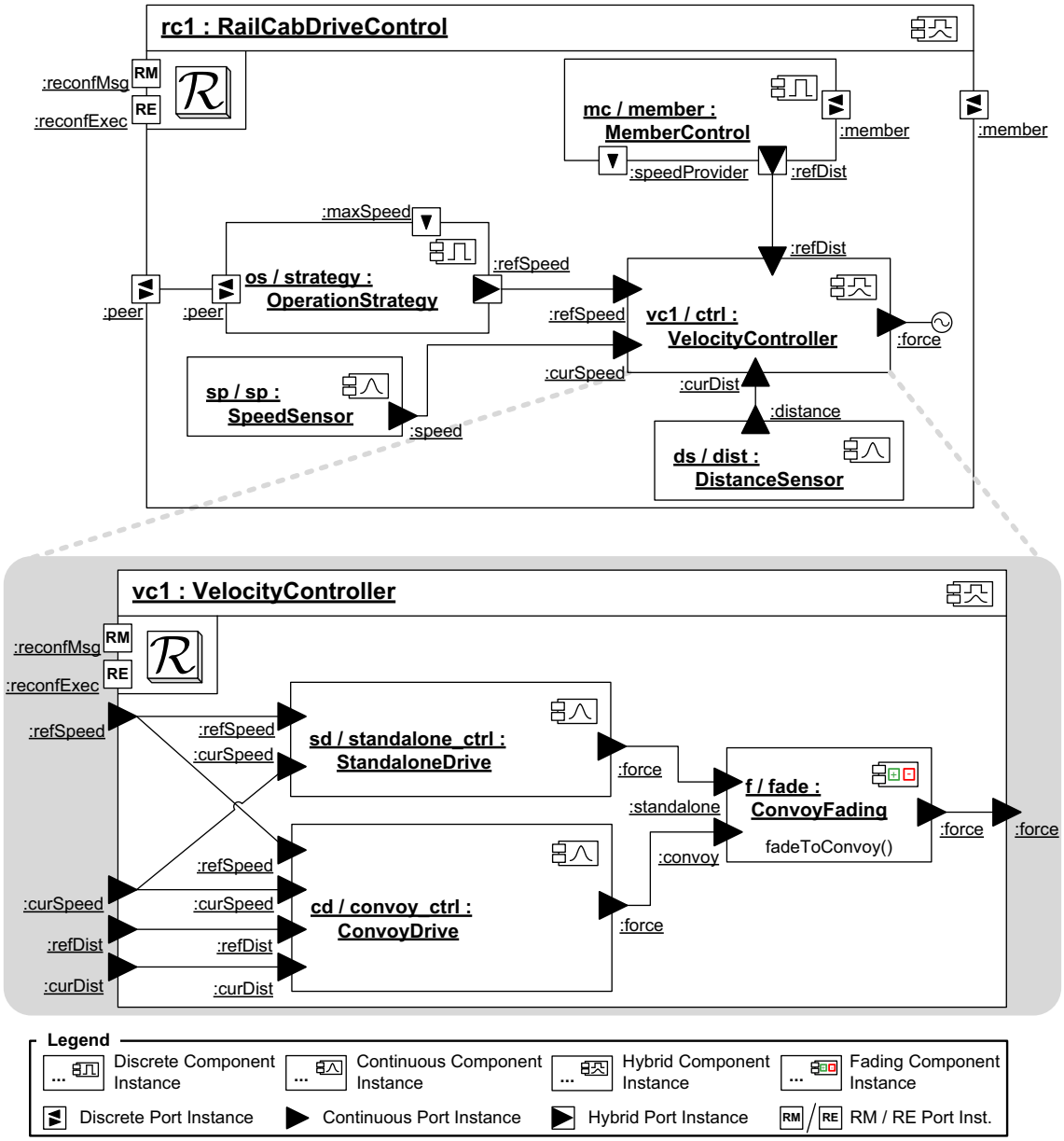

Fig. 16 RailCabDriveControl after executing the setup phase for the reconfiguration becomeMember

but its software architecture does not change. The duration of the fading phase is defined by the maximum duration of all involved fading functions.

5.2.3 Teardown

The execution of the reconfiguration finishes with the teardown phase. In this phase, both the behavior and the software architecture of the CPS change. The teardown phase is executed top-down as shown in Fig. 14, i.e., a component first executes its own teardown behavior before it triggers its children in parallel.

In the teardown phase, we destroy all component instances, port instances, and connector instances as specified by the reconfiguration rule. Furthermore, we activate all discrete component instances and port instances that were created in the setup phase including the connector instances between discrete port instances. The fading components now forward the unmodified value of the continuous component instance that has been created.

Continuing our example in Fig. 16, we now destroy the StandaloneDrive instance including all of its port instances and adjacent connector instances. In rc1, we create the delegation connector instance that delegates the port instance member of mc to the corresponding port instance of rc1.

Finally, we create the assembly connector instance between speedProvider of mc and maxSpeed of os. The result is, as expected, equivalent to the component instance Member shown in Fig. 5.

5.3 Quiescence

Component instances and port instances may not be deleted at any point in time. In particular, they may not be deleted if they currently perform a computation or if they are engaged in executing a communication protocol that is required for the safe operation of the system. *Quiescence* [18,29] defines whether it is safe to delete a component instance or one of its port instances at a certain point of time. Then, executing a reconfiguration safely demands that all affected component instances are quiescent.

As an example, consider a member RailCab that leaves a convoy. As a consequence, the RailCab will destroy its instance of MemberControl and it will switch back to the StandaloneDrive feedback controller. However, the RailCab may not perform this reconfiguration if it is still driving closely behind another RailCab. If it performs the reconfiguration, it will not be notified about braking maneuvers of the convoy and, thus, a crash is likely to occur.

In our approach, a structured component instance is quiescent with respect to a particular reconfiguration if and only if all children that are affected by this reconfiguration are quiescent. For continuous atomic component instances, the fading functions define how they may be safely replaced. For discrete atomic component instances, we currently leave it to the developer to specify the behavior for checking whether the component instance is quiescent. In any case, the check for quiescence is executed as part of the voting phase of the 2-phase-commit protocol. A preliminary concept for quiescence of discrete atomic component instances in MechatronicUML is introduced by Schubert et al. [30] but out of scope for this paper.

6 Declarative specification

Providing correct reconfiguration manager and executor component realizations in MechatronicUML is a manual cumbersome, labor-intensive, and error-prone task. Therefore, we introduce a declarative specification of the behavior of the reconfiguration controller based on tables in our approach. This declarative specification relieves the developer from manually providing correct reconfiguration manager and executor components (as they will be generated from the declarative specification) and make him more productive. These tables extend the MechatronicUML component model by additional syntactical elements that are tailored to the 2-phase-commit protocol.

Developers need to specify one table for each RM port and for each RE port of a reconfigurable component. This table enhances the interface of the port, i.e., which messages the port may send or receive, with timing constraints for the messages that are relevant for executing the 2-phase-commit protocol. In addition, developers need to specify one table for the manager and one table for the executor of each reconfigurable structured component. The entries in these tables define conditions that express when to execute which reconfiguration, but they do not specify how the conditions are checked and how reconfigurations are executed according to the 2-phase-commit protocol.

The timing constraints that are contained in our declarative, table-based specification are requirements for an execution of the reconfiguration behavior on a hardware platform. For a CPS, these requirements originate from three sources. First, they originate from conditions that are imposed by the physical environment. As an example, consider a convoy build-up at a switch. In this case, the reconfigurations for becoming a coordinator or member, respectively, need to be finished before approaching the switch too closely. Second, timing requirements are defined by the functional safety specification. In case of a hardware failure, a reconfiguration that implements a self-healing operation needs to be finished within a particular time in order to prevent a hazard. This particular time may be obtained by performing a timed hazard analysis [31]. Third, timing requirements originate from the quiescence criteria. The component instances that are affected by a reconfiguration need to remain quiescent throughout the reconfiguration (cf. Sect. 5.3). As a result, the reconfiguration needs to be finished before the component instance needs to execute some non-quiescent behavior that is necessary for safely operating the system.

In the following, we provide details regarding our declarative, table-based specifications of manager and executor as well as of the interfaces of RM ports and RE ports in Sects. 6.1 to 6.3.

6.1 Interface specification of RM ports

An RM port is a special kind of discrete port that is solely used for communication between the managers of reconfigurable components. Its interface is defined by a table with four columns where each interface entry corresponds to one row in the table. The first column defines the message types that may be sent to the parent. The second column gives additional information on the semantics of the message using a type. We distinguish two types of messages: info messages and requests. An info message is only provided for information and does not necessarily require a reconfiguration. A request is sent in situations where a reconfiguration is necessary from the perspective of the sending component and where it cannot solve the situation itself. In case of a request, the developer

Table 1 RM port specification of the RailCabDriveControl component (cf. [7])

	Message type	Type	Expected response time	Description
1	drivingAtHighSpeed	Info	—	RailCab travels at high speed.
2	drivingAtNormalSpeed	Info	—	RailCab travels at normal speed.
3	distanceSensorFailure	Request	200 ms	Distance sensor is broken.

of a component may specify an expected response time in the third column. It defines the point in time where the component needs the information whether a reconfiguration has been executed by the parent. The fourth column optionally contains a human readable description of the reported situation for a developer.

Table 1 shows the RM port specification for the RM port of the RailCabDriveControl component from our case study as introduced in Fig. 10. It contains three entries. First, the RailCabDriveControl informs its parent about its speed profile using the messages drivingAtHighSpeed and drivingAtNormalSpeed of type info. This information may be used to adapt the sensing of obstacles depending on the speed. If the speed is high, obstacles need to be sensed in larger distances to brake early enough. In addition, RailCabDriveControl sends a request positionSensorFailure, which denotes that the distance sensor is broken. This request triggers a self-healing operation (cf. [31]) and needs to be finished in 200 ms for guaranteeing the convoy safety.

6.2 Manager specification

As for the interfaces of the reconfiguration ports, specifying the reconfiguration manager component in MECHATRONICUML manually is a task we want to avoid effort-wise. Therefore, the behavior of a manager is specified declaratively using a table with eight columns in our approach (cf. Table 2). We refer to each row of the table as an entry of the manager specification. The entries of the manager specification define how the manager needs to react if it receives a particular message. In our approach, the manager only reacts to messages that it receives from the children or from the executor. We did not yet include dedicated monitoring capabilities for structured components in our approach. The manager specification needs to define at least one entry for each message that the manager may receive from the children or from the executor.

In the manager specification, the first column contains a message type that the manager may receive either from a child or from the executor. The second and third columns define whether the manager treats the message or whether it propagates the message to its parent. A message that is received from the executor always needs to be treated. We allow, however, that the manager operates as a sink with respect to messages sent by a child by neither treating nor propagating them. We do not allow treating and propagating a message at the same time because that may lead to conflicting reconfiguration decisions on different levels of hierarchy in the component model. All messages that are specified as propagated in the manager specification need to appear in the interface specification of the RM port parent of the corresponding reconfigurable component.

If the specification defines that a message is treated, the developer must specify a reconfiguration rule to be executed by the executor in the fourth column. Whether a reconfiguration may be executed at run time depends on (1) whether it is *allowed* to execute the reconfiguration (column Structural Condition) and (2) whether it is *useful* to execute the reconfiguration (column Invoke Planner). Only if both conditions evaluate to true during run time, the manager will trigger the executor to execute the reconfiguration. We explain these conditions in more detail in the following.

For each entry of the manager specification, the developer needs to define a structural condition. The structural condition specifies a condition on the embedded component instances that must be fulfilled for executing the reconfiguration. If a message appears in more than one entry, the entries associated to this message must have pairwise disjunct structural conditions. Currently, we only support specifying structural conditions based on component SDDs (cf. Sect. 2.4). It is only *allowed* to execute a reconfiguration if the structural condition is fulfilled. If the execution of the reconfiguration shall not be restricted, true may be used as a structural condition as for Entries 4, 5, and 6.

Second, we account for the usage of a planner component in our manager specification. The planner component will contain executable logic, which decides whether it is *useful* to execute the reconfiguration based on the goals of the system [32]. Although we have not explicitly added a planner component to our approach, yet, we enable the developer to specify whether to invoke such a planner component or not. A planner component may only be invoked if the message is treated. If a planner component is invoked, the developer needs to provide the maximum time that the planner component may run in the eighth column of the table. If no planner component shall be invoked, it is always considered to be useful to execute the reconfiguration if it is requested.

Table 2 shows the manager specification of the RailCabDriveControl component of our case study in Fig. 10. The messages in Entries 1 to 3 are sent by the child Opera-

Table 2 Manager specification of the RailCabDriveControl component (cf. [7])

	Message type	Treat	Propagate to parent	Reconfiguration rule	Structural condition	Invoke planner	Time for planning
1	becomeCoordinator	Yes	No	becomeCoordinator()	isStandalone()	Yes	20 ms
2	newMember	Yes	No	addConvoyMember()	isCoordinator()	No	—
3	becomeMember	Yes	No	becomeMember()	isStandalone()	Yes	20 ms
4	distanceSensorFailure	No	Yes	—	true	No	—
5	drivingAtHighSpeed	No	Yes	—	true	No	—
6	drivingAtNormalSpeed	No	Yes	—	true	No	—

tionStrategy that negotiates with other RailCabs whether to form a convoy. These messages are treated and not propagated. Therefore, we specify a reconfiguration rule that is contained in the executor specification (cf. Sect. 6.3) for each of them. Each of the reconfigurations specifies a structural condition. A RailCab may only become member of a convoy, if it is not yet engaged in a convoy. We specify this by an architectural constraint that checks whether ConvoyCoordination and MemberControl are both not instantiated. The same condition needs to be fulfilled for becoming a coordinator of a convoy. New members can only be added if RailCab already is the coordinator of a convoy, thus, having an instance of ConvoyCoordination. In addition, we foresee invoking a planner before building a convoy. For both reconfigurations, we permit the planner to run for 20 ms.

The messages in Entries 6 to 8 are sent by the VelocityController. These messages are propagated to the parent and not treated. Consequently, we neither specify a reconfiguration rule nor one of the three conditions for executing the reconfiguration.

6.3 Executor specification

The behavior of an executor is specified declaratively using a table with two columns. Again, we refer to each row of the table as an entry of the executor specification. Each entry of the executor specification contains a reference to a reconfiguration rule. In our approach, we use CSDs as introduced in Sect. 2.3 for specifying reconfiguration rules. The second column defines the maximum worst-case execution time (WCET, [33]) that is acceptable for executing the reconfiguration rule on a platform. Please note that this is not the actual WCET of the CSD on a particular platform but a requirement on the maximum WCET of the platform-specific system. For CSDs, worst-case execution time bounds can be computed in terms of elementary operations [34]. Translating these bounds to real platforms is a matter of refinement, i.e., to show that the execution time for the calculated amount of elementary operations on the real platform is lower than the maximum WCET.

Table 3 Executor specification of the RailCabDriveControl component (cf. [7])

	Reconfiguration rule	WCET
1	becomeCoordinatorQ()	50 ms
2	addConvoyMemberQ()	10 ms
3	becomeMemberQ()	50 ms

Table 3 shows the executor specification of the component RailCabDriveControl (cf. Fig. 10). RailCabDriveControl supports three reconfiguration rules. The first one creates the necessary components, ports, and connectors for operating as a convoy coordinator. If the component already operates as a coordinator, the second reconfiguration rule adds another member to the convoy. The third reconfiguration rule creates the necessary components, ports, and connectors for operating as a convoy member as specified by the CSD in Fig. 6.

6.4 Interface specification of RE ports

An RE port is a special kind of discrete port that is solely used for communication between the executors of reconfigurable components. Its interface is defined by a table with five columns. The first column defines a message type that it accepts from its parent. The second column contains a human readable description of the effect of sending a corresponding message to the component. The remaining columns contain time values that define timing requirements toward the execution of the 2-phase-commit protocol.

The third column contains the time for decision. This time value provides an upper bound for the time that the component needs for deriving a decision whether it may execute a reconfiguration or not. The fourth column contains a timing specification that defines an upper bound on how long the component needs for executing the reconfiguration that is associated with this message in the manager specification. If the reconfiguration may be executed according to single-phase execution, the time for execution is a single value. If the reconfiguration needs to be executed according to three-phase execution, then the timing specification contains

Table 4 RE port specification of the VelocityController component

	Message type	Description	Time for decision	Time for execution	Minimum commit time
1	switchToConvoy	The VelocityController operates as a convoy member and considers the distance to the preceding RailCab.	20 ms	Setup: 10 ms Fading: 50 ms Teardown: 5 ms	200 ms
2	switchToStandalone	The VelocityController operates as standalone or coordinator RailCab and will control speed solely based on a reference speed.	20 ms	Setup: 10 ms Fading: 50 ms Teardown: 5 ms	200 ms

separate time values for setup, fading, and teardown. We need to provide distinct time values for each phase for correctly computing how much time a hierarchical reconfiguration needs for being executed after deploying the component as explained below. Finally, the fifth column provides the minimum commit time that defines a lower bound on how long the component's decision on the execution of the reconfiguration can be considered valid (cf. Sec. 5, where this commit time bound is used during the voting phase).

Table 4 shows the interface specification of the RE port reconfExec of the VelocityController component in Fig. 10. The component offers two reconfigurations to its parent that correspond to the two entries in the table. The first one uses the message type switchToConvoy. Sending this message to the RE port of VelocityController triggers the switch to the ConvoyDrive feedback controller. Since this reconfiguration needs to be executed based on three-phase execution, the time for execution defines execution times for each phase. The second entry uses the message type switchToStandalone that causes the VelocityController to switch back to the StandaloneDrive feedback controller.

7 Generate operational behavior

The declarative, table-based specification of the reconfiguration controller introduced in the previous section cannot be verified or implemented directly. In order to verify or implement the reconfiguration behavior, we need a formal and operational behavior specification that additionally specifies *how* reconfigurations are executed according to the 2-phase-commit protocol. Therefore, we automatically generate a RTSC for both manager and executor, because RTSCs are formal and operational.

Using RTSCs for specifying the operational behavior of manager and executor enables to reuse the existing tool chain for MECHATRONICUML. That includes model checking support [14,35], WCET analyses [34], export to simulation environments as MATLAB/Simulink [28] or Modelica [36], and code generation [37].

We define the generation of the operational behavior specification based on generation templates for the RTSCs of manager and executor [7]. The generation templates define the 2-phase-commit protocol implementation as outlined in Sect. 5 and contain placeholders for the entries of the tables of our declarative, table-based reconfiguration specification introduced in Sect. 6. The placeholders are automatically filled by the information given in each row of the tables. By using the generation templates and an automatic generation process, we relieve the developer from specifying a large and complicated behavior specification for the 2-phase-commit protocol by himself. Our generator generates the 2-phase-commit behavior automatically from the declarative specification described in Sect. 6. Creating this specification is much less manual work than creating the declarative specification.

To convey an idea about the effort savings, we report here how many modeling elements are automatically generated for our example. For the manager RTSC given in Sect. 7.1 our generator relieves the developer from modeling 18 states and 30 transitions plus 2 states and 8 transitions for each entry of the manager specification. For the executor RTSC given in Sect. 7.2, our generator automates the effort to model another of 71 states and 102 transitions manually plus 1 state and 4 transitions for each entry in the RE port specification, 2 states and 5 transitions for each reconfiguration rule assuming single-phase execution, and 6 states and 7 transition for each reconfiguration assuming three-phase execution. In our case study with the RailCab system this actually relieves the developer from modeling more than 50 states and more than 90 transitions inside the reconfiguration component. Note that in addition to saving the effort of modeling these states and transitions, our generator also does not make introduce the mistakes a human would introduce when modeling manually leading to more reduced overall effort.

7.1 Manager specification

Figure 17 shows the generation template for the manager RTSC. The RTSC template is complex and provides many

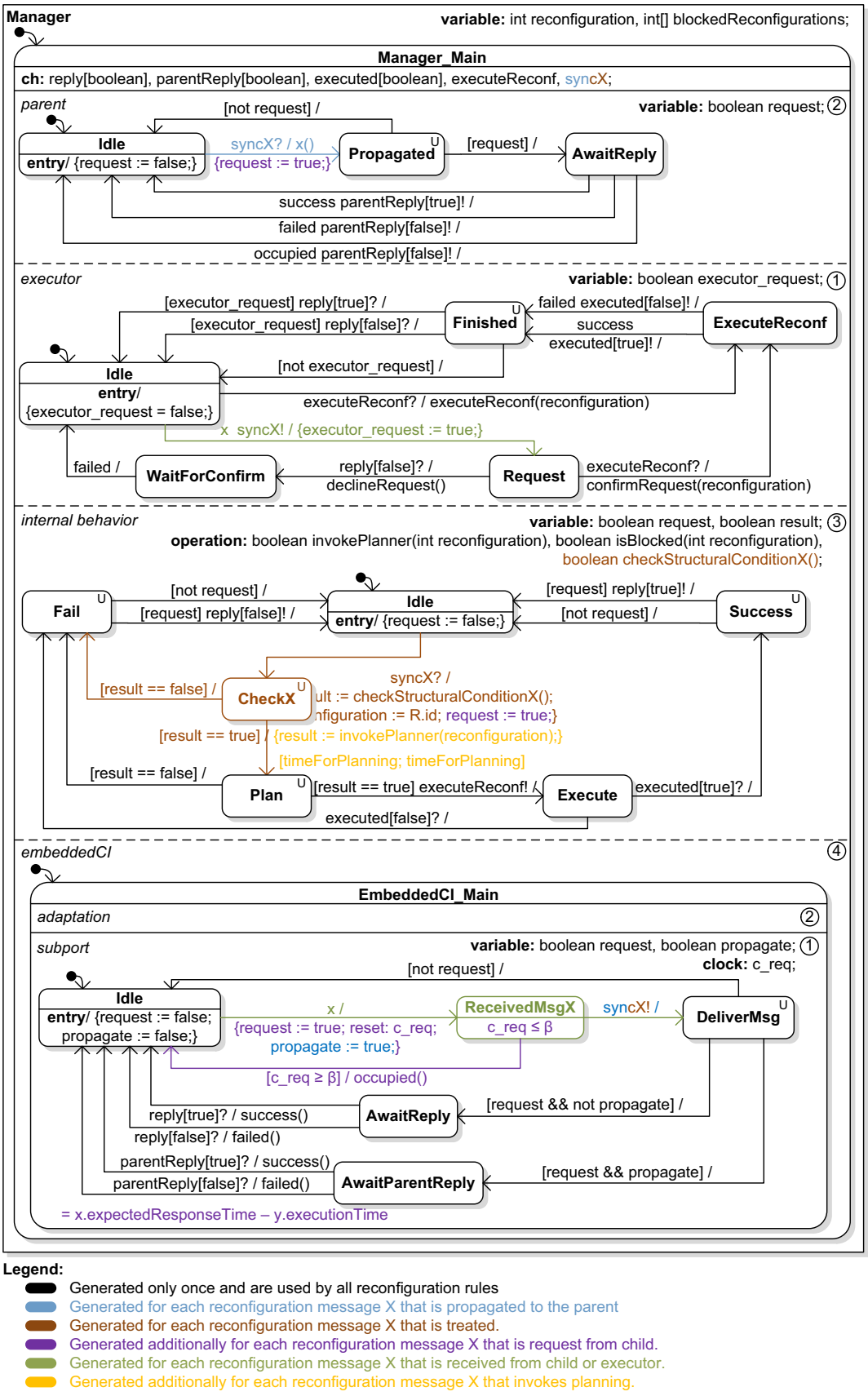

Fig. 17 Generation template for the manager RTSC (cf. [7])

variation points that depend on the manager specification. By using our generation template, however, we hide the complexity of the RTSC from the developer who can reuse the template for all of his reconfigurable structured components without understanding its internals.

In the RTSC, all black states and transitions form the general frame of the RTSC. They are always present and will only be generated once for every manager RTSC. The colored parts are variable and are generated based on the entries of the manager specification. The green parts are generated of each entry of the manager specification. They are used to handle an incoming message. If the message is a request, we additionally generate the purple parts for the corresponding entry. The brown parts are generated for each message that is treated. They specify the behavior for checking whether to execute the reconfiguration. If the message is propagated to the parent, we generate the blue parts. The yellow and pink parts provide the functionality for invoking a planner and checking whether a reconfiguration is blocked if this is specified by the corresponding manager specification entry.

The resulting manager RTSC consists of one state Manager_Main with four or five regions. The region parent implements the RM port parent of the manager that is used for communicating with the parent. The region executor implements the executor port for communicating with the executor. The region internal behavior specifies the behavior of deciding whether to execute which reconfiguration. The region embeddedCI implements the behavior of the multi-port embeddedCI that is used for communicating with the children. The RTSC follows the standard structure of a multi-port RTSC [5], i.e., it contains one region adaptation and one region subport. At run time, behavior defined by the subport region is executed in separate threads for each instance of the multi-port. The adaptation region is only executed once and coordinates the behavior of the different instances.

The information flow through the manager RTSC depends on the use cases. In the following, we only give a coarse description of the information flow and refer to [10] for a detailed technical description. In Use Case 1 as shown in Fig. 12, messages sent by the children are received by the embeddedCI port of the manager and are processed by the subport region (transition from Idle to ReceivedMsgX). If the message is treated, the subport triggers the internal behavior via syncX which decides whether to execute a reconfiguration by checking the structural condition and optionally invoking the planning. Then, the internal behavior triggers the executor region via executeReconf, which in turn sends a message to the executor to trigger the reconfiguration (transition from Idle to ExecuteReconf). Then, the executor region waits until the executor reports the result of the reconfiguration (either success or failed). The result is first propagated to the internal behavior via executed and finally to the subport via reply. Finally, the subport reports the result to the child that requested the reconfiguration (transitions from AwaitReply to Idle).

If the manager specification defines that the received message is to be propagated to the parent, the subport triggers the parent region directly. In case of a request, the region parent waits for the answer of the parent component and reports this answer back to the subport.

In Use Case 2 as shown in Fig. 13, the executor propagates the messages that it received via its RE port to the manager. The manager RTSC receives this message in the executor region (transition from Idle to Request). Then, the executor RTSC triggers the internal behavior and the execution proceeds as for Use Case 1.

If several requests reach the manager at the same time, for example, from different children, we need to serialize these messages to ensure isolation of the reconfiguration operations. This is achieved by the internal behavior that ensures that only one message is treated at a time.

7.2 Executor specification

Figure 18 shows an excerpt of the generation template for the executor RTSC. The RTSC template is also complex and provides many variation points that depend on the executor specification. By using our generation template, however, we hide the complexity of the RTSC from the developer who can reuse the template for all of his reconfigurable structured components without understanding its internals.

As for the manager RTSC, all black states and transitions form the general frame of the RTSC, while the colored parts are variable and are generated based on the entries of the executor specification. The purple parts are generated for any reconfiguration that can be executed by the executor. They compute the children that are affected by the reconfiguration and initialize the data structures for keeping track of the reconfiguration progress. The green parts are generated additionally for reconfigurations that are executed based on single-phase execution. In particular, they invoke the CSDs that perform the reconfiguration. The blue parts are generated additionally for reconfigurations that are executed based on three-phase execution. They invoke the partial reconfigurations for each of the three phases and wait for the start of the next phase.

As part of this paper, we only show a small excerpt of the executor RTSC. A full version with a detailed technical description can be found in [10]. The excerpt shown in Fig. 18 shows the parts of the executor RTSC that are responsible for controlling the execution of the 2-phase-commit protocol. Thus, the information flow is the same for Use Case 1 and Use Case 2. At first, the internal behavior is triggered via startExecution to initialize the 2-phase-commit protocol in response to the message received from the manager (not shown in the excerpt). Then, we store whether the recon-

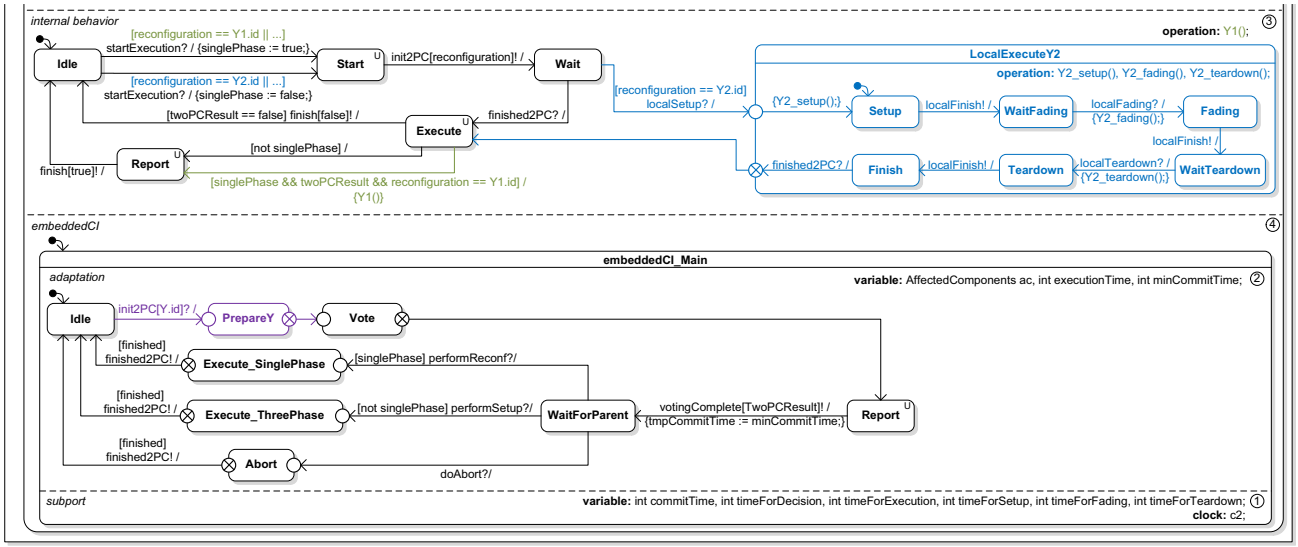

Fig. 18 Excerpt of the generation template for the executor RTSC

figuration is executed based on single-phase execution or three-phase execution and start the 2-phase-commit protocol. The actual 2-phase-commit protocol is handled by the adaptation region of the embeddedCI multi-port RTSC. It computes the affected children (state PrepareY), coordinates the voting phase (state Vote) by querying and receiving the votes from the affected children, and performs the execution (states Execute_SinglePhase, Execute_ThreePhase, Abort). The necessary communication with the children is handled by the subport region.

The internal behavior executes the reconfiguration of the structured component. For single-phase execution, the reconfiguration Y1 is invoked as an effect of the transition from Execute to Report after the children have finished their reconfigurations. For three-phase execution, the partial reconfigurations in each phase are executed inside the LocalExecuteY2 state based on the order shown in Fig. 14.

8 Verification

In order to guarantee that the reconfiguration behavior of the structured component is correct and, thus, safe, we need to verify the specified reconfiguration behavior of a structured component. We need to ensure that it fulfills all of the ACI-T properties of the 2-phase-commit protocol. In our approach, formal verification is enabled by the operational behavior specifications for manager and executor in terms of RTSCs.

For formally verifying the 2-phase-commit protocol with respect to the ACI-T properties, we need to verify the following properties. We present the properties here in an informal fashion to improve read and understandability. When we verified them, we had to formalize them first as we will explain below:

1. If the executor decides to execute (abort), then all affected children execute (abort) (**Atomicity**).
2. The reconfiguration rules cannot produce an inconsistent CIC (**Consistency**).
3. The executor will execute no other reconfiguration than the one requested by the manager (**Consistency**).
4. At any time, at most one reconfiguration is executed (**Isolation**).
5. The RTSCs of manager and executor are free from deadlocks (**Timing**).
6. Each reconfiguration is executable (**Timing**).

Properties 1, 3, and 4 can already be guaranteed by the correctness of the generation templates given in Sect. 7. Therefore, they do not need to be verified again for a particular structured component. The correctness of the generation templates with respect to formalizations of these three properties has been verified using UPPAAL [7].

Property 2 specifies that reconfigurations may not produce an inconsistent CIC. A CIC may be either syntactically inconsistent or semantically inconsistent. A CIC is syntactically inconsistent if it violates the conditions for syntactical correctness that we introduced in Sect. 2.2. In our approach, the CSDs provable guarantee that CICs remain syntactically consistent after a reconfiguration due to syntactic restrictions. Thus, no further check is necessary. A CIC is semantically inconsistent if the instantiated component instances, port instances, and connector instances do not constitute a desired

functional behavior. In the worst case, the component may even be unsafe. In our case study, we considered a RailCab that drives as part of a convoy as a member but which does not have an instance of MemberControl (cf. Fig. 15) although it switched the controller. Such situations cannot be prevented by syntactic rules but need to be verified for each structured component as we describe in Sect. 8.1.

Finally, Properties 5 and 6 specify the conditions for a correct timing specification. In a platform-independent model, we can formally verify whether the timing requirements provided in our declarative, table-based specification are satisfiable in principle. If they are satisfiable, there may exist a hardware platform that enables to execute the reconfiguration behavior without violating the timing requirements. After deriving a platform-specific model that includes a platform model and a deployment of components to hardware nodes [38], we may already check at design time whether the execution of the reconfigurations on the hardware platform fulfills the imposed requirements. We describe the verification of the timing specification including a formalization of Properties 5 and 6 in Sect. 8.2.

In combination, both verification steps and the verified generation templates enable to formally verify the reconfiguration behavior of our components completely with respect to the ACI-T properties. The only part of the reconfiguration behavior that cannot be formally verified using model checking is given by the implementations of the fading functions. Their correctness needs to be assessed via testing, e.g., using simulation-based testing in MatLab [16,28].

8.1 Consistency

We ensure consistency by verifying that the reconfiguration behavior cannot produce a semantically inconsistent CIC. However, it is not possible to automatically derive from the component model which CICs are semantically inconsistent and which are not. Therefore, a developer needs to provide this information explicitly. In the following, we introduce three possibilities for specifying semantically inconsistent CICs. These are forbidden CICs (Sect. 8.1.1), architectural invariants (Sect. 8.1.2) and properties in temporal logic (Sect. 8.1.3). For each of these, we describe an approach for formal verification. However, we did not evaluate this aspect as part of our case study. This is subject to future work.

8.1.1 Forbidden CICs

A forbidden CIC is a particular CIC or part of a CIC that may never occur for a given component. As an example, consider the CIC in Fig. 19 that defines an excerpt of an instance inconsistent of the component RailCabDriveControl. It specifies the situation where both, MemberControl and ConvoyCoordination,

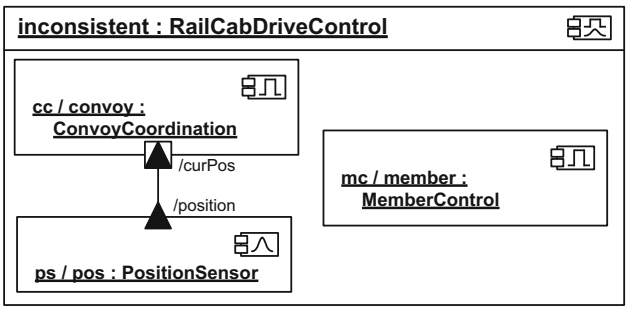

Fig. 19 Example of a forbidden CIC

are instantiated. In our example, we only allow RailCabs to be either the coordinator or a member but not both at the same time. Therefore, we consider this CIC as semantically inconsistent and, thus, as forbidden.

For verifying that a forbidden CIC may not occur, we perform a reachability analysis [39] on the CSDs. In this approach, we compute all possible configurations of a component starting from the initial configurations. Then, we may check whether the forbidden CIC occurs in any of these configurations.

8.1.2 Architectural invariants

Our component model enables to specify architectural invariants based on component SDDs as described in Sect. 2.4. Any CIC that violates an architectural invariant is considered as semantically inconsistent. As an example, an architectural invariant may define that having an instance of the MemberControl component inside RailCabDriveControl implies that VelocityController embeds an instance of the ConvoyDrive component (cf. Figs 2 and 3). Component SDDs are more expressive than forbidden CICs because they enable specifying conditional constraints and support quantification based on first-order logic.

Component SDDs may be verified by a reachability analysis [39] on the CSDs. This is facilitated by translating the component SDDs to equivalent CSDs [40]. Then, the SDD is fulfilled if and only if the resulting story diagram can be executed successfully on each configuration of the component. In the reachability analysis, we check whether there exists a configuration to which the CSD resulting from the component SDD cannot be matched.

8.1.3 Properties in temporal logic

Temporal logic constraints based on CTL and LTL [41] enable to specify constraints on the evolution of a CIC. In our example, we want to specify that a RailCab may not directly switch from being coordinator to being member. Such properties may be expressed by graph-based variants of CTL and

LTL such as quantified CTL (QCTL, [42]), graph-based LTL (GLTL, [43]), or first-order TCTL (FO-TCTL, [44]). As an example, we might want to verify that a coordinator RailCab always needs to return to the configuration for driving alone (cf. Fig. 4) before it may become member of a convoy. This could be formalized in GLTL as:

```
G ((coordinator ⇒ (¬ member U
   standalone)) U (member ∨
         G ¬ member))
```

where `coordinator`, `standalone`, and `member` are graph patterns representing the (partial) CICs of RailCabDriveControl operating as a coordinator, standalone RailCab, or member.

Verifying such properties requires a graph-based model checking, for example, based on GROOVE [43]. Applying GROOVE on CSDs requires translating the component model and CSDs into a typed graph transformation system [20]. Then, the component model defines the type graph and the initial configuration of the structured component defines the initial graph. In addition, the CSDs are translated to typed graph transformation rules based on the type graph.

8.2 Timing

We ensure a correct timing specification by applying timed model checking [13] on the RTSCs of manager and executor that are contained in the platform-independent component model. If the model checking encounters a deadlock or a reconfiguration rule that cannot be executed, the timing requirements in our declarative, table-based specification are not satisfiable. If the timing requirements are satisfiable, then there may exist a platform on which the reconfiguration controllers can be executed. Thus, we only check the necessary conditions for correct execution on the model level. Ensuring that the reconfiguration controllers are indeed executable on a given platform needs to be checked in a later refinement step toward a platform-specific model or implementation. This is, however, beyond the scope of this paper and we refer to [10] for more information.

In order to achieve an efficient and scalable verification approach, we verify the timing requirements separately for each structured component. This is enabled by using stubs for the parent component as well as the children. These stubs formalize the behavior that is allowed by the interface specifications of the RM and RE ports of the children and that is necessary for a correct vertical integration of the component with respect to timing. This enables a scalable assume/guarantee style [41, Chap. 12] verification where the verification of the structured component assumes that the children adhere to their interface specification. That a component cannot violate its interface specification is guaranteed by the generation templates and the timing verification of the children.

For applying timed model checking, we need to translate the RTSCs of manager and executor into a network of timed automata as illustrated in Fig. 20. Its their properties we want to guarantee and, therefore, verify. In particular, we obtain one timed automaton for each region of the manager RTSC and of the executor RTSC. In addition, we need one automaton that defines the behavior of the connector between manager and executor. Finally, we add two parent stubs for the parent component and two child stubs for each embedded component part which we assume to operate according to their specifications. The number of child stubs is equal to the number of timed automata that are generated for the subport RTSCs of the embeddedCl ports of manager and executor (cf. Sect. 7). The arrows illustrate the information flow between the timed automata that results from the synchronizations used in the RTSCs and the messages being sent.

Figure 21 shows an executor child stub that was generated based on the RE port specification of ConvoyCoordination for verifying the correct timing of RailCabDriveControl in our case study.

The behavior of the executor child stub is as follows. It waits in Idle for being triggered by RailCabDriveControl for executing a reconfiguration. In this case, only AddConvoyMemberAtPos may be triggered by RailCabDriveControl. This corresponds to the channel childAddConvoyMemberAtPos and the executor child stub switches to ReceivedAddConvoyMemberAtPos. As part of the transition, the executor child stub non-deterministically chooses whether it will commit

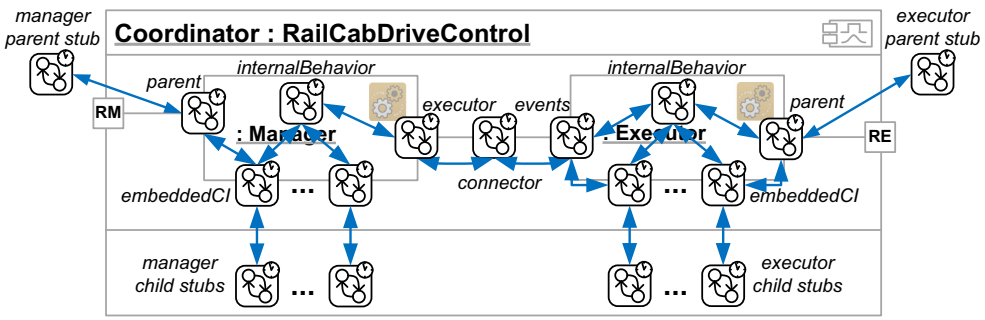

Fig. 20 Sketch of the generated Uppaal model

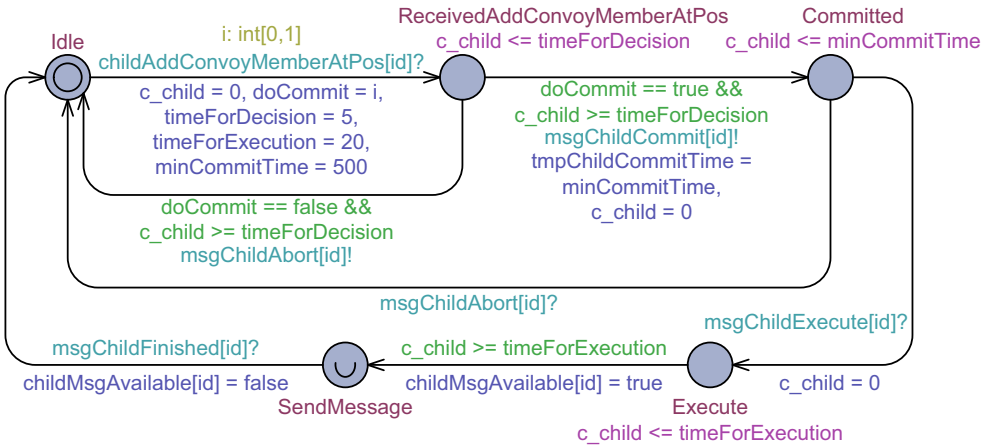

Fig. 21 Child stub representing ConvoyCoordination for the verification of RailCabDriveControl

or abort the request. The result is stored in doCommit. In addition, we assign the time values for the timeForDecision, the timeForExecution, and the minCommitTime that are contained in the RE port interface specification to the eponymous variables. The child stub now waits in ReceivedAddConvoyMemberAtPos until the timeForDecision has expired. Then, it either synchronizes via msgChildAbort and returns to Idle or synchronizes via msgChildCommit and switches to Committed. In Committed, the invariant ensures that the executor child stub will only rest in the Committed state until the minCommitTime expires. As a result, a deadlock occurs if RailCabDriveControl does not send a decision whether to execute or abort the reconfiguration in time. Based on the decision by RailCabDriveControl, the executor child stub switches either to Idle (abort) or to Execute (execute). In Execute, the executor child stub rests as long as the timeForExecution has not expired. Then, it switches to SendMessage to check whether RailCabDriveControl may accept the result message, which is then sent at the transition from SendMessage to Idle.

We provide a specification of parent stubs and manager child stubs on our Web site.[3] The resulting timed automata may then be verified using UPPAAL [45]. Therefore, we need to formalize the timing-related properties mentioned above. First, we need to check that the model contains no deadlock, which is formalized in Uppaal as:

$$\varphi \; = \; \texttt{A[]}\ \texttt{not deadlock}$$

Second, we need to check that each reconfiguration is executable, which is formalized as:

$$\forall_{i \in reconfIDs} : \varphi_i \; = \; \texttt{E<>}$$
$$\texttt{(Executor.internal\_behavior.Report \&\&}$$
$$\texttt{executor\_reconf == i)}$$

That means we obtain one property for each reconfiguration in the executor specification that we identify by an ID. For each reconfiguration, we check whether we can reach the report state in the internal behavior region of the executor RTSC (cf. Fig. 18). Reaching this state indicates a successful execution of the corresponding reconfiguration.

9 Implementation

Our proof-of-concept implementation covers the modeling extensions to MechatronicUML (Sect. 4), the table-based declarative specification of the reconfiguration controller (Sect. 6), and the generation of the operational behavior specification (Sect. 7). We integrated our implementation into the MechatronicUML Tool Suite[4] [46].

We extended the meta-model of the MechatronicUML Tool Suite for being able to add reconfiguration controllers to structured components and for creating the declarative specifications. The meta-model has been developed using the Eclipse Modeling Framework (EMF, [47]). The static semantics has been completely encoded in the meta-model using the Object Constraint Language (OCL, [48]). Based on the meta-models, we extended the diagram editors such that they support the specification of reconfigurable components. In addition, we implemented a generator that enables to generate RTSCs for manager and executor based on the generation templates given in Sect. 7. The generator has been implemented in QVT Operational [49]. In addition, we support to convert a non-reconfigurable component into a reconfigurable component for user's convenience.

Finally, we modeled the full version of the RailCab example that we use in this paper in the MechatronicUML Tool Suite. A detailed description of the example can be found in [10]. Instructions for downloading the MechatronicUML Tool Suite including the example can be found on our Web site.[5]

[3] https://trac.cs.upb.de/mechatronicuml/wiki/PaperCBSE2013.

[4] https://trac.cs.upb.de/mechatronicuml.

[5] https://trac.cs.upb.de/mechatronicuml/wiki/PaperSEAMS2015.

At present, our proof-of-concept implementation is limited in the following ways. The generation templates do not support input and output parameters of CSDs. The concept for verifying consistency introduced in Sect. 8.1 has not yet been implemented. The generation of stubs for the parent and the children as described in Sect. 8.2 has not yet been automatized, and the translation of the generated RTSCs for manager and executor to UPPAAL is not yet fully covered by the current implementation [14]. However, the manually derived UPPAAL model for the RailCab case study can be found on our Web site [6] to allow for reproducibility.

10 Assumptions and limitations

Our approach for the transactional execution of reconfiguration is based on the assumption that any reconfiguration that has been started can be finished successfully. In particular, we assume that no hardware failures occur while executing a reconfiguration.

Our method has two limitations. First, all monitoring has to be performed by atomic components that accumulate the monitoring data and provide accumulated data to the manager of the parent component. Second, we may only trigger at most one reconfiguration on each child of a structured component instance when executing a CSD for the structured component instance.

Finally, our evaluation is limited in the sense that we only conducted a case study on an excerpt of a research prototype system up to the platform-independent modeling level. The RailCab model used as a running example has not yet been executed on an embedded hardware platform.

11 Related work

Our concept for hierarchical execution of reconfigurations in component-based systems relates to three research areas. First, we discuss related works regarding architecture description languages (ADLs) that support the specification of self-adaptive systems in Sect. 11.1. Thereafter, we discuss related works based on the autonomic computing architecture MAPE-K [50] in Sect. 11.2. Finally, we relate our approach to software component models that support either runtime reconfiguration or real-time systems or both in Sect. 11.3. For all of the approaches, we discuss whether they support reconfigurations in a hierarchical component model while preserving component encapsulation and respecting ACI-T properties.

11.1 ADLs for self-adaptive systems

ADLs specify software architectures based on components and connectors, although the term *component* is less strictly defined as for component models. Bradbury et al. [51] survey ADLs that enable runtime reconfiguration of the software architecture. They classify these ADLs into three categories: graph-based (e.g., CHAM, Hirsch et al.), process algebra-based (e.g., Dynamic Wright, Darwin), and formal-logic-based (GeReL). Graph-based approaches define an initial configuration that is modified by graph rewriting rules as in MechatronicUML. Process algebra-based approaches specify processes for each configuration using a process algebra. At run time, components switch between processes to execute reconfigurations. Formal-logic-based approaches declaratively specify component behavior based on first-order logic. We refer to [10] for references and a detailed discussion of the mentioned approaches. Besides the approaches surveyed by Bradbury et al., two more recent approaches in this direction exist, namely by Kacem et al. [52] (graph-based) and Bartels and Kleine [53] (process algebra-based). All of the mentioned approaches do not support real-time constraints for functional or reconfiguration behavior. Most approaches discussed above (except Darwin and GeReL) do not support structured components and GeReL is the only one that can guarantee atomicity and isolation for the execution of reconfigurations. None of the approaches supports real-time constraints or the exchange of feedback controllers.

11.2 Reconfiguration approaches based on MAPE-K

The MAPE-K reference architecture [50] defines autonomic managers that execute a self-adaptation control loop for executing runtime reconfigurations. The control loop **m**onitors the system, **a**nalyzes whether a reconfiguration is necessary, **p**lans when to execute the reconfiguration, and finally **e**xecutes it. It may be extended by a **k**nowledge source for improving adaptation decisions. To this end, our approach currently supports the analysis and execution parts. The monitoring is performed inside the atomic components and the planning is not necessary because reconfigurations will always be triggered instantaneously.

There exist several approaches that implement or refine the MAPE-K reference architecture. The Rainbow framework [54] provides an implementation of the reference architecture MAPE-K that targets business information systems. Their concept defines an adaptation manager and an adaptation executor that closely correspond to the manager and executor in our approach. The framework by de Oliveira et al. [55] uses several autonomic managers for adapting cloud applications in a coordinated fashion. Their autonomic managers have a similar purpose as our reconfiguration controller

[6] https://trac.cs.upb.de/mechatronicuml/wiki/PaperCBSE2013.

but are horizontally composed. The approach by Edwards et al. [56] uses meta-level components for implementing a self-adaptation control loop similar to MAPE-K based on a hierarchical component model. The meta-level components fulfill a similar purpose as our reconfiguration controller. Similarly, Vromant et al. [57] connect several MAPE control loops that are located on the same hierarchy level following a master-slave pattern. All of these approaches do not explicitly connect meta-level components or MAPE control loops, respectively, on different hierarchy levels such that hierarchical execution is not supported and ACI-T properties cannot be guaranteed. EUREMA [58] supports the specification of multiple self-adaptation feedback loops based on MAPE-K in a single system. Feedback loops on different architectural levels may be connected and coordinated. Weyns et al. [59] discuss different design patterns for connecting multiple MAPE feedback loops in a system. With respect to their pattern, our approach is based on the hierarchical control pattern. In contrast to our approach, the approaches do not support ACI-T properties except that EUREMA satisfies isolation of adaptations.

11.3 Software component models

The surveys by Lau [60] and Crnković et al. [61] review various component models. They distinguish between general purpose component models as, for example, EJB [62], and specialized component models for particular domains. The latter usually address business information systems or embedded real-time systems. In this section, we focus primarily on component models for embedded real-time systems (Sect. 11.3.1) and on component models that support runtime reconfiguration (Sect. 11.3.2).

11.3.1 Component models for embedded real-time systems

Hošek et al. [63] surveyed component models for embedded real-time systems. Only few of which support runtime reconfiguration including SOFA-HI [64], ProCom [65], MyCCM-HI [66], BlueArX [67], and AUTOSAR[7] [68]. All of these component models are restricted to mode changes [69] where a component instance moves from one configuration to another one. In contrast, CSDs of MechatronicUML provide a more powerful and flexible specification of reconfigurations including control flow. Of the mentioned approaches, only SOFA-HI and ProCom support hierarchical reconfigurations and support some of the ACI-T properties.

The approach by Hang et al. [70] implements a composable mode change operator based on the ProCom [65]. Similar to our approach, they use dedicated reconfiguration components that are hierarchically connected. Reconfiguration requests may traverse the hierarchy bottom-up or top-down. They adopted our approach for executing reconfiguration based on single-phase execution and use our verification approach introduced in Sect. 8.2. Therefore, they support ACI properties but they do not provide explicit real-time properties regarding the execution of reconfigurations in their specification.

Pop et al. [4] introduce a mode change operation of embedded real-time systems based on SOFA-HI [64]. In their approach, each mode of a component instance corresponds to a configuration. They also separate functional and reconfiguration behavior and enable mode changes across different levels of hierarchy. Their approach ensures consistency and isolation but they cannot ensure atomicity if a child is currently not able to reconfigure and they do not provide a formal verification support for checking for a correct timing of reconfigurations.

CompoSE [71] defines a hierarchical component model for modeling embedded systems. Components specify configurations based on combinations of ports and embedded component instances similar to modes. At run time, a component may switch between configurations. Their approach supports isolation as well as the verification of consistency and timing properties. Atomicity is not explicitly supported.

The DEECo component model [72] provides non-hierarchical components for soft real-time systems based on ensembles. Components declaratively specify conditions for being part of an ensemble and a shared runtime framework automatically constructs and dissolves ensembles based on these conditions. De Nicola et al. [73] present a textual language named SCEL that enables to express these conditions as policies including the necessary modifications of the software architecture for establishing the ensemble. In contrast to our approach, they neither support hierarchy nor ACI-T properties.

The fault-tolerant component model by de Lemos et al. [74] partitions component behavior into normal and abnormal (exception) behavior. We follow the same idea by separating normal behavior and reconfiguration behavior. Their approach provides horizontal propagation of exceptions, but not propagation to parent components. With a similar objective, Strunk and Knight [75] provide a dependable reconfiguration approach for hard real-time systems where a system moves from one configuration to another one with degraded functionality in case of a failure. The approach, however, does not support hierarchy or timing properties, but it ensures by formal proofs that any reconfiguration can be executed successfully.

All of the approaches mentioned have in common that they do not consider the particularities of exchanging feedback controllers. Consequently, none of them supports a mecha-

[7] http://www.autosar.org.

nism for three-phase execution of reconfigurations enabling safe reconfigurations involving feedback controllers.

Other component models for embedded real-time systems like Koala [76], Robocop [77], SaveCCM [78], Rubus [79], COMDES-II [80], PECOS [81], Flex-eWare [82], and CHESS [83] provide the ability to specify real-time behavior on a low level of abstraction including formal analysis. The EAST-ADL2 [84] architecture description language for the development of automotive systems provides the ability to specify a hierarchical component architecture with a focus on the integration of feedback controllers and it supports the specification and formal analysis of component behaviors [85]. However, all of these approaches have in common that they do not enable the specification and analysis of run time reconfiguration.

11.3.2 Further component models supporting reconfiguration

A main inspiration for our reconfiguration approach was the Fractal component model [23,25]. Fractal supports the definition of hierarchical components including runtime reconfiguration of structured components. Each component consists of a membrane and a content area. The content area embeds other components, while the membrane contains so-called controllers that enable introspection and reconfiguration. Their concept extends each reconfigurable component with a reconfiguration interface and a reconfiguration executor for executing reconfiguration scripts. We have adopted the concept of a reconfiguration executor and extended the remote reconfiguration invocation [26]. In contrast to our approach, Fractal starts reconfigurations optimistically and performs a roll-back in case that the reconfiguration is not possible. Therefore, Fractal achieves ACI properties as well, but does consider neither timing of reconfigurations nor the verification of functional behavior and reconfigurations.

Zhang et al. [86] provide an approach for safe adaptation of component-based systems. Their approach uses one central adaptation manager that orchestrates the adaptation process, and several agents that are attached to the components and perform their modification. If an adaptation cannot be finished successfully, they perform a roll-back to the previous configuration. Therefore, their approach guarantees ACI properties for the execution of reconfigurations but considers neither hierarchical components nor real-time properties. The approach by Boyer et al. [87] also follows a roll-back approach for achieving reliable reconfiguration, but their approach neither treats hierarchy nor achieves any of the ACI-T properties. The SOFA 2.0 component model [88] uses component controllers similar to Fractal and to our approach, called microcomponents, that enables hierarchical reconfigurations. However, they do not address transactional execution of reconfigurations.

12 Conclusions

In this paper, we introduced our comprehensive solution for safe reconfigurations of encapsulated hierarchical component-based CPS. Our solution is able to guarantee ACI and real-time properties. In addition, it also addresses the continuous nature of the physical system environment by incorporating feedback controller components. Our reconfiguration protocol is based on an adaption of the 2-phase-commit protocol [2, ch. 7]. In our approach, we syntactically extend the components of the MechatronicUML component model by a dedicated reconfiguration controller that executes the 2-phase-commit protocol. The reconfiguration controller enables to execute reconfigurations across different levels of hierarchy without violating component encapsulation. Our approach significantly reduces the complexity of specifying such hierarchical reconfigurations by providing a rather simple declarative specification based on tables that enables to automatically generate an implementation of the 2-phase-commit protocol. We extended the existing 2-phase-commit protocol such that it can execute reconfigurations in a CPS including the exchange of feedback controllers according to ACI-T properties. While our 2-phase-commit protocol specification guarantees atomicity and isolation offhand, we define a verification approach for guaranteeing consistency and a correct timing of reconfigurations. Therefore, we can ensure the correctness and, thus, the safety of the reconfigurations. We demonstrated the effectiveness of our approach by specifying the full hierarchical reconfiguration behavior for the RailCab system, which we used as case study. This smart railway system that employs runtime reconfigurations allowed us to show that our approach actually allows the effective and efficient modeling and verification of a selected part of a complex, real-world CPS prototype system. In particular, we generated the 2-phase-commit protocol implementation for the reconfiguration behavior and verified the resulting models. We found that this saved the developer from creating more than 50 states and 90 transitions. In addition, we can guarantee that the reconfiguration behavior which the developer specified in a high-level declarative model is correct and safe. Hence, we conclude from our case study that our modeling and verification approach is suited for CPS.

The contribution of this paper enables software engineers of CPS to cope with the additional complexity that is introduced by adding self-adaptive behavior in the form of runtime reconfigurations to their systems. In particular, self-adaptive behavior adds more sources for errors that may occur at run time and hardens to predict the behavior of the system. How-

ever, self-adaptive behavior is the basis for self-healing [89] and self-optimization [90] that enable to improve safety, availability, and (resource) efficiency of the system. With our contribution, we support software engineers in coping with this additional complexity such that they may safely unleash the full potential of self-adaptive behavior when developing the next generation of CPS.

In future works, we plan to extend our approach in several ways. First, our approach currently only covers reactive reconfigurations based on fixed rules, i.e., we always execute the same reconfiguration rule under the same condition. The approach should be extended by an interface to a knowledge component as advised by MAPE-K [50] for improving reconfiguration decisions and enabling proactive reconfigurations rather than only reactive reconfigurations supported so far. This also enables to provide several reconfiguration rules for a situation where the system can adjust the decision which rule to execute based on experience from past decisions. As a second extension, an explicit consideration of functional safety concerns is necessary, e.g., based on considering self-healing operations in reaction to hardware failures. In addition, a runtime risk manager [91] may reevaluate safety concerns during run time in order to forbid reconfigurations that are unsafe under particular environmental circumstances. As a third extension, we plan to add explicit monitoring support to MechatronicUML, e.g., using a framework like Kieker.[8] In particular, this should also enable to specify additional monitoring in the reconfiguration controller of a structured component, e.g., for monitoring information entering the structured component. In any case, the decision about executing a reconfiguration is made based on monitoring data that may be incomplete or inconsistent, e.g., due to false assumptions, unpredictable phenomena in the environment, or even imprecise and inaccurate sensors [92]. Therefore, we want to investigate whether the reconfiguration behavior in our approach may be improved by explicitly addressing uncertainty during the development. Fourth, our reconfiguration approach needs to be extended by an approach for quiescence [18,29] of discrete atomic component instances. In particular, this approach should support the developers at design time in specifying when a discrete atomic component instance is quiescent and it should provide automatic checks at run time. Finally, we plan to further evaluate our approach in an industrial context.

Acknowledgements The work presented in this paper has been conducted at the time that the authors spent at the University of Paderborn and the Fraunhofer Institute for Mechatronic Systems Design (IEM) in Paderborn. At the time of conducting the research, they have been funded by these institutions.

[8] http://kieker-monitoring.net.

References

1. Szyperski, C., Gruntz, D., Murer, S.: Component Software-Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, Boston (2002)
2. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley, Boston (1987)
3. Hang, Y., Carlson, J., Hansson, H.: Towards mode switch handling in component-based multi-mode systems, In: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE'12, pp. 183–188. ACM, New York, NY (2012). doi:10.1145/2304736.2304766
4. Pop, T., Plášil, F., Outly, M., Malohlava, M., Bureš, T.: Property networks allowing oracle-based mode-change propagation in hierarchical components, In: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE'12, pp. 93–102. ACM, New York, NY (2012). doi:10.1145/2304736.2304753
5. Eckardt, T., Heinzemann, C., Henkler, S., Hirsch, M., Priesterjahn, C., Schäfer, W.: Modeling and verifying dynamic communication structures based on graph transformations. Comput. Sci. Res. Dev. **28**(1), 3–22 (2013). doi:10.1007/s00450-011-0184-y
6. Becker, S., Dziwok, S., Gerking, C., Heinzemann, C., Schäfer, W., Meyer, M., Pohlmann, U.: The MechatronicUML method: Model-driven software engineering of self-adaptive mechatronic systems, In: Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, pp. 614–615. ACM, New York, NY (2014). doi:10.1145/2591062.2591142
7. Heinzemann, C., Becker, S.: Executing reconfigurations in hierarchical component architectures, In: Proceedings of the 16th international ACM Sigsoft symposium on Component based software engineering, CBSE '13, pp. 3–12. ACM, New York, NY (2013). doi:10.1145/2465449.2465452
8. Becker, S., Dziwok, S., Gerking, C., Heinzemann, C., Thiele, S., Schäfer, W., Meyer, M., Pohlmann, U., Priesterjahn, C., Tichy, M.: The MechatronicUML design method –process and language for platform-independent modeling, Tech. Rep. tr-ri-14-337, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, version 0.4 (2014)
9. Group, O.M.: Model Driven Architecture (MDA) – MDA Guide rev. 2.0, document – ormsc/14-06-01 (2014). http://www.omg.org/cgi-bin/doc?ormsc/14-06-01
10. Heinzemann, C.: Verification and simulation of self-adaptive mechatronic systems, Ph.D. thesis, University of Paderborn (2015)
11. Heineman, G.T., Councill, W.T. (eds.): Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing Co. Inc, Boston (2001)
12. Group, O.M.: Unified Modeling Language (UML) 2.4.1 Superstructure Specification, document formal/2011-08-06 (2011)
13. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools, In: Desel, J., Reisig, W. Rozenberg, G. (Eds.) Lectures on Concurrency and Petri Nets, Vol. 3098 of Lecture Notes in Computer Science, pp. 87–124. Springer, Heidelberg (2004). doi:10.1007/978-3-540-27755-2_3
14. Gerking, C., Dziwok, S., Heinzemann, C., Schäfer, W.: Domain-specific model checking for cyber-physical systems, In: 12th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa 2015), Ottawa (2015)
15. Burmester, S., Giese, H., Oberschelp, O.: Hybrid UML components for the design of complex self-optimizing mechatronic systems, In: Braz, J. Araújo, H. Vieira, A. Encarnação, B. (Eds.) Informatics in Control, Automation and Robotics I, pp. 281–288. Springer, Netherlands (2006). doi:10.1007/1-4020-4543-3_34

16. Osmic, S., Münch, E., Trächtler, A., Henkler, S., Schäfer, W., Giese, H., Hirsch, M.: Safe online-reconfiguration of self-optimizing mechatronic systems, In: Gausemeier, J., Rammig, F.J., Schäfer, W. (Eds.) Selbstoptimierende mechatronische Systeme: Die Zukunft gestalten. 7. Internationales Heinz Nixdorf Symposium für industrielle Informationstechnik, pp. 411–426. (2008)
17. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. Computer **37**(7), 56–64 (2004). doi:10.1109/mc.2004.48
18. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software, In: Proceedings of the 28th international Conference on Software Engineering, ICSE '06, pp. 371–380. ACM, New York, NY (2006). doi:10.1145/1134285.1134337
19. Tichy, M., Henkler, S., Holtmann, J., Oberthür, S.: Component story diagrams: A transformation language for component structures in mechatronic systems, In: Postproceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), pp. 27–39 (2008)
20. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation, Monographs in Theoretical Computer Science. Springer, Berlin (2006). doi:10.1007/3-540-31188-2
21. Garlan, D., Monroe, R.T., Wile, D.: Acme: architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–67. Cambridge University Press, New York, NY (2000)
22. Heinzemann, C.: Component story decision diagrams, Tech. Rep. tr-ri-14-335, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn (2014)
23. Léger, M., Ledoux, T., Coupaye, T.: Reliable dynamic reconfigurations in a reflective component model, In: Grunske, L., Reussner, R., Plášil, F. (Eds.) Component-Based Software Engineering, Vol. 6092 of Lecture Notes in Computer Science, pp. 74–92. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13238-4_5
24. Heinzemann, C., Sudmann, O., Schäfer, W., Tichy, M.: A discipline-spanning development process for self-adaptive mechatronic systems, in: Proceedings of the 2013 International Conference on Software and System Process, ICSSP 2013, pp. 36–45. ACM, New York, NY (2013). doi:10.1145/2486046.2486055
25. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: The FRACTAL component model and its support in Java. Softw. Pract. Exp. **36**(11—-12), 1257–1284 (2006). doi:10.1002/spe.767
26. Bennour, B., Henrio, L., Rivera, M.: A reconfiguration framework for distributed components, In: Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime, SINTER '09, pp. 49–56. ACM, New York, NY (2009). doi:10.1145/1596495.1596509
27. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer **42**(10), 22–27 (2009). doi:10.1109/mc.2009.326
28. Heinzemann, C., Rieke, J., Schäfer, W.: Simulating self-adaptive component-based systems using MATLAB/Simulink, In: IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems, SASO '13, IEEE Computer Society, pp. 71–80. (2013). doi:10.1109/SASO.2013.17
29. Kramer, J., Magee, J.: Analysing dynamic change in software architectures: A case study, in: Proceedings of the Fourth International Conference on Configurable Distributed Systems, CDS '98, IEEE Computer Society, pp. 91–100. (1998). doi:10.1109/CDS.1998.675762
30. Schubert, D., Gerking, C., Heinzemann, C.: Towards safe execution of reconfigurations in cyber-physical systems, In: Proceedings of the 19th International ACM Sigsoft Symposium on Component Based Software Engineering, CBSE '16 (2016)
31. Priesterjahn, C., Steenken, D., Tichy, M.: Timed hazard analysis of self-healing systems, In: Cámara, J., de Lemos, R., Ghezzi, C., Lopes, A. (Eds.) Assurances for Self-Adaptive Systems, Lecture Notes in Computer Science, vol. 7740, pp. 112–151. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36249-1_5
32. Ziegert, S., Wehrheim, H.: Temporal plans for software architecture reconfiguration. Comput. Sci. Res. Dev. **30**, 1–18 (2014). doi:10.1007/s00450-014-0259-7
33. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem–overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. (TECS) **7**(3), 36:1–36:53 (2008). doi:10.1145/1347375.1347389
34. Burmester, S., Giese, H., Seibel, A., Tichy, M.: Worst-case execution time optimization of story patterns for hard real-time systems, In: Proceedings of the 3rd International Fujaba Days 2005, pp. 71–78 (2005)
35. Heinzemann, C., Brenner, C., Dziwok, S., Schäfer, W.: Automata-based refinement checking for real-time systems. Comput. Sci. Res. Dev. **30**(3–4), 255–283 (2015). doi:10.1007/s00450-014-0257-9
36. Pohlmann, U., Holtmann, J., Meyer, M., Gerking, C.: Generating Modelica models from software specifications for the simulation of cyber-physical systems, In: Proceedings of the 40th Euromicro Conference on Software Engineering and Advanced Applications, SEAA '14, IEEE Computer Society, pp. 191–198 (2014). doi:10.1109/SEAA.2014.18
37. Burmester, S., Giese, H., Schäfer, W.: Model-driven architecture for hard real-time systems: From platform independent models to code, In: Hartman, A., Kreische, D. (Eds.) Proceedings of the European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA '05). Lecture Notes in Computer Science, vol. 3748, pp. 25–40. Springer, Heidelberg (2005). doi:10.1007/11581741_4
38. Pohlmann, U., Meyer, M., Dann, A., Brink, C.: Viewpoints and views in hardware platform modeling for safe deployment, In: Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, VAO '14, pp. 23:23–23:30. ACM, New York, NY (2014). doi:10.1145/2631675.2631682
39. Heinzemann, C., Suck, J., Eckardt, T.: Reachability analysis on timed graph transformation systems, Electron. Commun. EASST **32**
40. Ahmadian, A.S., Aydogan, C., Braun, D., Bustamante, L.G., Gerking, C., Issiz, S., Kopecki, L., Prescher, P.: Developer Documentation of the Project Group SafeBots I. Project group. University of Paderborn, Paderborn (2011)
41. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
42. Rensink, A.: Model checking quantified computation tree logic, In: Baier, C., Hermanns, H. (Eds.) CONCUR 2006 – Concurrency Theory, Lecture Notes in Computer Science, vol. 4137, pp. 110–125. Springer, Heidelberg (2006). doi:10.1007/11817949_8
43. Rensink, A.: Explicit state model checking for graph grammars, In: Degano, P., Nicola, R., Meseguer, J., (Eds.) Concurrency, Graphs and Models, Lecture Notes in Computer Science, vol. 5065, pp. 114–132. Springer, Heidelberg (2008). doi:10.1007/978-3-540-68679-8_8
44. Suck, J., Heinzemann, C., Schäfer, W.: Formalizing model checking on timed graph transformation systems, Tech. Rep. tr-ri-11-316, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Paderborn (2011)
45. Behrmann, G., David, A., Larsen, K. G., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0, In: Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems, QEST 2006, IEEE Computer Society, pp. 125–126. Los Alamitos, CA (2006). doi:10.1109/QEST.2006.59
46. Dziwok, S., Gerking, C., Becker, S., Thiele, S., Heinzemann, C., Pohlmann, U.: A tool suite for the model-driven software engineering of cyber-physical systems, In: Proceedings of the 22nd ACM

SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 715–718. ACM, New York, NY (2014). doi:10.1145/2635868.2661665
47. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. The Eclipse Series, 2nd edn. Addison-Wesley, Boston (2008)
48. Group, O.M.: Object Constraint Language (OCL) 2.3.1, document formal/2012-01-01 (2012). http://www.omg.org/spec/OCL/2.3.1/
49. Group, O.M.: Query/View/Transformation (QVT) 1.1, document formal/2011-01-01 (2011). http://www.omg.org/spec/QVT/1.1/
50. IBM. An architectural blueprint for autonomic computing, Autonomic Computing White Paper, IBM (2006)
51. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications, In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems, WOSS '04, pp. 28–33. ACM, New York, NY (2004). doi:10.1145/1075405.1075411
52. Kallel, S., Kacem, M.H., Jmaiel, M.: Modeling and enforcing invariants of dynamic software architectures. Softw. Syst. Model. **11**(1), 127–149 (2012). doi:10.1007/s10270-010-0162-z
53. Bartels, B., Kleine, M.: A CSP-based framework for the specification, verification, and implementation of adaptive systems, In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '11, pp. 158–167. ACM, New York, NY (2011). doi:10.1145/1988008.1988030
54. Cheng, S.-W., Garlan, D., Schmerl, B.: Evaluating the effectiveness of the Rainbow self-adaptive system, In: ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '09, IEEE Computer Society, pp. 132 –141 (2009). doi:10.1109/seams.2009.5069082
55. De Oliveira, F. A., Ledoux, T., Sharrock, R.: A framework for the coordination of multiple autonomic managers in cloud environments, In: IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems, SASO'13, IEEE Computer Society, pp. 179–188 (2013). doi:10.1109/saso.2013.27
56. Edwards, G., Garcia, J., Tajalli, H., Popescu, D., Medvidović, N., Sukhatme, G., Petrus, B.: Architecture-driven self-adaptation and self-management in robotics systems, In: ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '09, IEEE Computer Society, pp. 142 –151 (2009). doi:10.1109/seams.2009.5069083
57. Vromant, P., Weyns, D., Malek, S., Andersson, J.: 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 '11, pp. 202–207. ACM, New York, NY (2011). doi:10.1145/1988008.1988037
58. Vogel, T., Giese, H.: Model-driven engineering of self-adaptive software with EUREMA. ACM Trans. Auton. Adapt. Syst. (TAAS) **8**(4), 18:1–18:33 (2014). doi:10.1145/2555612
59. Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K. M.: On patterns for decentralized control in self-adaptive systems, In: de Lemos, R., Giese, H., Müller, H. A., Shaw, M. (Eds.) Software Engineering for Self-Adaptive Systems II, Lecture Notes in Computer Science, vol. 7475, pp. 76–107. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35813-5_4
60. Lau, K.-K., Wang, Z.: Software component models. IEEE Trans. Softw. Eng. **33**(10), 709–724 (2007). doi:10.1109/tse.2007.70726
61. Crnković, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.V.: A classification framework for software component models. IEEE Trans. Softw. Eng. **37**(5), 593–615 (2011). doi:10.1109/tse.2010.83
62. Oracle, JSR 345: Enterprise JavaBeansTM, Version 3.2, EJB Core Contracts and Requirements (Apr. 2013) (2015). http://download.oracle.com/otn-pub/jcp/ejb-3_2-fr-eval-spec/ejb-3_2-core-fr-spec.pdf
63. Hošek, P., Pop, T., Bureš, T., Hnětynka, P., Malohlava, M.: Comparison of component frameworks for real-time embedded systems, In: Grunske, L., Reussner, R., Plášil, F. (Eds.) Component Based Software Engineering, Lecture Notes in Compute Science, vol. 6092, pp. 21–36. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13238-4_2
64. Prochazka, M., Ward, R., Tuma, P., Hnětynka, P., Adamek, J.: A component-oriented framework for spacecraft on-board software, In: Proceedings of DASIA 2008, DAta Systems In Aerospace, Palma de Mallorca, European Space Agency Report Nr. SP-665, (2008)
65. Vulgarakis, A., Suryadevara, J., Carlson, J., Seceleanu, C., Pettersson, P.: Formal semantics of the ProCom real-time component model, In: Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications, SEEA '09, IEEE Computer Society, pp. 478–485. Los Alamitos, CA (2009). doi:10.1109/seaa.2009.53
66. Borde, E., Feiler, P.H., Haïk, G., Pautet, L.: Model driven code generation for critical and adaptative embedded systems. SIGBED Rev. **6**, 10:1–10:5 (2009). doi:10.1145/1851340.1851352
67. Kim, J.E., Rogalla, O., Kramer, S., Hamann, A.: Extracting, specifying and predicting software system properties in component based real-time embedded software development, In: 31st International Conference on Software Engineering–Companion Volume, IEEE Computer Society, pp. 28–38 (2009). doi:10.1109/icse-companion.2009.5070961
68. AUTOSAR, AUTOSAR 4.1 - Guide to Modemanagement, document Identification No. 440, Version 2.2.0 (2014). http://www.autosar.org/fileadmin/files/releases/4-1/software-architecture/system-services/auxiliary/AUTOSAR_EXP_ModemanagementGuide.pdf
69. Hirsch, D., Kramer, J., Magee, J., Uchitel, S.: Modes for software architectures, In: Gruhn, V., Oquendo, F. (Eds.) Software Architecture, Lecture Notes in Computer Science, vol. 4344, pp. 113–126. Springer, Heidelberg (2006). doi:10.1007/11966104_9
70. Hang, Y., Hansson, H.: Handling multiple mode switch scenarios in component-based multi-mode systems, In: Proceedings of the 20th Asia-Pacific Software Engineering Conference, APSEC'13, IEEE Computer Society, vol. 1, pp. 404–413 (2013). doi:10.1109/apsec.2013.61
71. Adler, R., Schaefer, I., Trapp, M., Poetzsch-Heffter, A.: Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. ACM Trans. Embed. Comput. Syst. **10**(2), 201–2039 (2010). doi:10.1145/1880050.1880056
72. Bureš, T., Gerostathopoulos, I., Hnětynka, P., Keznikl, J., Kit, M., Plášil, F.: DEECo: an ensemble-based component system, In: Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering, CBSE '13, pp. 81–90. ACM, New York, NY (2013). doi:10.1145/2465449.2465462
73. De Nicola, R., Ferrari, G., Loreti, M., Pugliese, R.: A language-based approach to autonomic computing, In: Beckert, B., Damiani, F., de Boer, F.S., Bonsangue, M.M.: (Eds.) Formal Methods for Components and Objects, Lecture Notes in Computer Science, vol. 7542, pp. 25–48. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35887-6_2
74. de Lemos, R., de Castro Guerra, P.A., Rubira, C.M.Fischer: A fault-tolerant architectural approach for dependable systems. IEEE Softw. **23**(2), 80–87 (2006). doi:10.1109/ms.2006.35
75. Strunk, E.A., Knight, J.C.: Dependability through assured reconfiguration in embedded system software. IEEE Trans. Dependable Secure Comput. **3**(3), 172–187 (2006). doi:10.1109/tdsc.2006.33

76. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. Computer **33**(3), 78–85 (2000). doi:10.1109/2.825699
77. Maaskant, H.: A robust component model for consumer electronic products, In: Stok, P. (Ed.) Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, Philips Research Book Series, vol. 3, pp. 167–192. Springer, Netherlands (2005). doi:10.1007/1-4020-3454-7_7
78. Åkerholm, M., Carlson, J., Fredriksson, J., Hansson, H., Håkansson, J., Möller, A., Pettersson, P., Tivoli, M.: The SAVE approach to component-based development of vehicular systems. J. Syst.Softw. **80**(5), 655–667 (2007). doi:10.1016/j.jss.2006.08.016
79. Hänninen, K., Mäki-Turja, J., Nolin, M., Lindberg, M., Lundbäck, J., Lundbäck, K.-L.: The Rubus component model for resource constrained real-time systems, In: 3rd IEEE International Symposium on Industrial Embedded Systems, SIES 2008, IEEE Computer Society, pp. 177–183 (2008). doi:10.1109/SIES.2008.4577697
80. Ke, X., Sierszecki, K., Angelov, C.: COMDES-II: A component-based framework for generative development of distributed real-time control systems, In: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA '07, IEEE Computer Society, pp. 199–208 (2007). doi:10.1109/rtcsa.2007.29
81. Genssler, T., Christoph, A., Winter, M., Nierstrasz, O., Ducasse, S., Wuyts, R., Arévalo, G., Schönhage, B., Müller, P., Stich, C.: Components for embedded software: The PECOS approach, In: Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES '02, pp. 19–26. ACM, New York, NY (2002). doi:10.1145/581630.581634
82. Jan, M., Jouvray, C., Kordon, F., Kung, A., Lalande, J., Loiret, F., Navas, J., Pautet, L., Pulou, J., Radermacher, A., Flex-eware, L.S.: A flexible model driven solution for designing and implementing embedded distributed systems. Softw. Pract. Exp. **42**(12), 1467–1494 (2012). doi:10.1002/spe.1143
83. Panunzio, M., Vardanega, T.: A component-based process with separation of concerns for the development of embedded real-time software systems. J. Syst. Softw. **96**, 105–121 (2014). doi:10.1016/j.jss.2014.05.076
84. Cuenot, P., Frey, P., Johansson, R., Lönn, H., Papadopoulos, Y., Reiser, M.-O., Sandberg, A., Servat, D., Tavakoli Kolagari, R., Törngren, M., Weber, M.: The EAST-ADL architecture description language for automotive embedded software, In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (Eds.) Model-Based Engineering of Embedded Real-Time Systems, Lecture Notes in Computer Science, vol. 6100, pp. 297–307. Springer, Heidelberg (2010). doi:10.1007/978-3-642-16277-0_11
85. Chen, D., Feng, L., Qureshi, T.N., Lönn, H., Hagl, F.: An architectural approach to the analysis, verification and validation of software intensive embedded systems. Computing **95**(8), 649–688 (2013). doi:10.1007/s00607-013-0314-4
86. Zhang, J., Cheng, B.H.C., Yang, Z., McKinley, P.K.: Enabling safe dynamic component-based software adaptation, In: de Lemos, R., Gacek, C., Romanovsky, A. (Eds.) Architecting Dependable Systems III, Lecture Notes in Computer Science, vol. 3549, pp. 194–211. Springer, Heidelberg (2005). doi:10.1007/11556169_9
87. Boyer, F., Gruber, O., Pous, D.: Robust reconfigurations of component assemblies, In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Computer Society, pp. 13–22. Piscataway, NJ (2013). doi:10.1109/ICSE.2013.6606547
88. Hnětynka, P., Bureš, T.: Advanced features of hierarchical component models, In: Zendulka, J. (Ed.) Proceedings of the 10th International Conference on Information System Implementation and Modeling, ISIM'07, CEUR-WS.org. vol. 252, pp. 1–8 (2007)
89. Shaw, M.: "self-healing": softening precision to avoid brittleness: position paper for WOSS '02: workshop on self-healing systems, In: Proceedings of the first workshop on Self-healing systems, WOSS '02, pp. 111–114. ACM, New York, NY (2002). doi:10.1145/582128.582152
90. Gausemeier, J., Rammig, F.-J., Schäfer, W. (Eds.) Design Methodology for Intelligent Technical Systems, Lecture Notes in Mechanical Engineering, Springer, Berlin (2014)
91. Priesterjahn, C., Heinzemann, C., Schäfer, W., Tichy, M.: Runtime safety analysis for safe reconfiguration, In: Proceedings of the 3. Workshop „Self-X and Autonomous Control in Engineering Applications", 10. IEEE International Conference on Industrial Informatics, INDIN'12, IEEE Computer Society, pp. 1092 – 1097 (2012). doi:10.1109/INDIN.2012.6300900
92. Ramirez, A.J., Jensen, A.C., Cheng, B.H.C.: A taxonomy of uncertainty for dynamically adaptive systems, In: Proceedings of the 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS'12, IEEE Computer Society, pp. 99 –108 (2012). doi:10.1109/seams.2012.6224396

Christian Heinzemann is a researcher in software engineering in the Corporate Research Department of the Robert Bosch GmbH in Renningen, Germany. Before, he worked as a research assistant at the Fraunhofer Institute for Mechatronic Systems Design (IEM) in Paderborn and at the University of Paderborn. He received his PhD in the year 2015 in software engineering from the University of Paderborn, Germany, for his work on the verification and simulation of self-adaptive mechatronic systems.

Steffen Becker holds the chair for software engineering at the University of Technology Chemnitz. Before, he was assistant professor at the University of Paderborn, department head at the Research Centre for Informatics (FZI) in Karlsruhe and research assistant in Karlsruhe and Oldenburg. He received his PhD in the year 2008 in Software Engineering from the university of Oldenburg, Germany, for his work on the quality prediction of model-driven developed software systems in the context of the five-year DFG-funded young researchers group Palladio. He is a frequent reviewer for journals in software engineering and a member of various program committees of conferences in the area. In particular, he is a permanent member of the steering committee of the International Conference on Quality of Software Architectures (QoSA) part of the CompArch federated events since 2005.

Andreas Volk is a software engineer at Bosch SoftTec GmbH in Hildesheim, Germany. He received his master's degree in the year 2014 in software engineering from the University of Paderborn, Germany. During his studies he worked on the simulation of self-adaptive mechatronic systems.