Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems

Nils Kaminski[a] Evgeny Kusmenko[a] Bernhard Rumpe[a]

a. RWTH Aachen University, Department of Software Engineering (Germany)

Abstract The component and connector modeling paradigm has proven to be a powerful means for the design of complex systems in engineering domains like avionics, automotive, and cyber-physical systems. Many component and connector languages are limited to the description of static architectures. However, agents like autonomous vehicles living in a changing world need to be able to adapt themselves to unforeseen situations and communicate with a steadily changing network of peers. In this work we present a dynamic event-based reconfiguration modeling framework for the component and connector-based design of self-adaptive cooperative agents. Therefore, we introduce the concepts of data-triggered and service-based reconfiguration and enable modeling controlled dynamic component creation as well as adaptations of component interfaces without losing type-safety. The methodology is motivated and explained using a running example from the domain of cooperating vehicles.

Keywords Component and connector; dynamic reconfiguration; self-adaptive systems.

1 Introduction

In engineering domains like automotive, Cyber-Physical System (CPS), and robotics software architectures are composed of a multitude of parts fulfilling tasks from engine control and sensor fusion to trajectory planning and execution. Each of these parts is developed by a dedicated team of experts, e.g. control, communication, and mechanical engineers. Finally, a working solution needs to be assembled by systems engineers. To handle the complexity of such systems, tools and languages are required, enabling the decomposition of a large architecture into smaller modules which can be developed and tested individually until they are eventually composed by means of some clear interfaces. The Component & Connector (C&C) modeling paradigm has proven to be a powerful approach for the design of complex engineering systems in research and industry [BMR+17b]. Self-contained functionality is encapsulated into so called components, which in turn can receive and send data via typed ports. Communication

Nils Kaminski, Evgeny Kusmenko, Bernhard Rumpe. *Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems.* In Journal of Object Technology, vol. 18, no. 2, 2019, pages 2:1–20.
doi:10.5381/jot.2019.18.2.a2

[KKR19] N. Kaminski, E. Kusmenko, B. Rumpe:
Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems.
In: The Journal of Object Technology, 18(2), pp. 1-20. July 2019.
www.se-rwth.de/publications/

between the components needs to be declared explicitly by connecting two compatible ports with a so called connector. The possibility to organize component structures hierarchically makes the C&C paradigm applicable to all granularity levels of a project and allows the developer to zoom into the system until the desired details are exposed. Widely used solutions like Simulink [Mat16] offer large component libraries enabling rapid prototyping and experimentation while still providing type safety and code generation for production software.

However, C&C modeling languages focus on the design of static systems, i.e. systems where the architecture is fixed at compile-time. While this is perfectly sufficient for most domains, novel kinds of applications featuring Internet of Things (IoT) and CPS technologies may require the ability to restructure and reconfigure their architectures in accordance with new goals and requirements steadily emerging at runtime. Consider agent-based systems like cooperative vehicle networks where participants need to adapt themselves to the driving environment and communicate with a heterogeneous and steadily changing group of peers to achieve different, often competing goals. Traffic optimization at intersections or platooning require appropriate vehicle behavior relying on different communication protocols. In some situations decisions are made using decentralized schemes while other circumstances require instruction from a master agent negotiated in advance.

To tackle the design of such applications we propose a framework for the event-triggered and type-safe runtime reconfiguration of C&C-based systems. As a motivation, we introduce a running example from the cooperative vehicles domain in section 2. Next, we derive a set of requirements for a dynamic C&C modeling language in section 3. In section 4 we discuss state-of-the-art C&C languages and their means for dynamic architectural reconfiguration. The solution design is presented in section 5 as an extension for the textual C&C modeling language EmbeddedMontiArc (EMA) and explained using building blocks from a cooperative driving project. The paper is concluded in section 6.

2 Running Example

To motivate our work, we will look into the domain of cooperative vehicles. Such vehicles can be regarded as an advanced kind of fully autonomous vehicles using communication over a vehicle-to-vehicle (V2V) network to optimize the performance of the traffic system, e.g. with respect to throughput, safety, energy efficiency, and the like. Each participant of a cooperative vehicle network is an agent pursuing its own particular goals and possessing an individual knowledge of the environment. Consider the running example setup in fig. 1 featuring four vehicles labeled with the numbers 1-4. For simplicity of notation, we are going to assume that all vehicles in our example are identical in terms of both their physical properties as well as their behavior models. The four vehicles arrive at the intersection simultaneously and we assume there are no traffic lights. Since the priority to the right rule fails, as well, we obtain a deadlock. At this point, a new goal emerges for each vehicle $i \in \{1, 2, 3\}$: *resolve intersection deadlock with x and y* or *avoid collision with x and y* where $x, y \in \{1, 2, 3\} \setminus \{i\}$ are the two vehicles competing with i. Although, there is a whole lot of simple algorithms to resolve the problem easily without any communication, optimality is only achieved if the deadlock is detected and resolved way before the vehicles arrive at the intersection. The vehicle winning the priority can continue driving while the others slow down to individual velocities. To achieve this functionality, the vehicles need to communicate

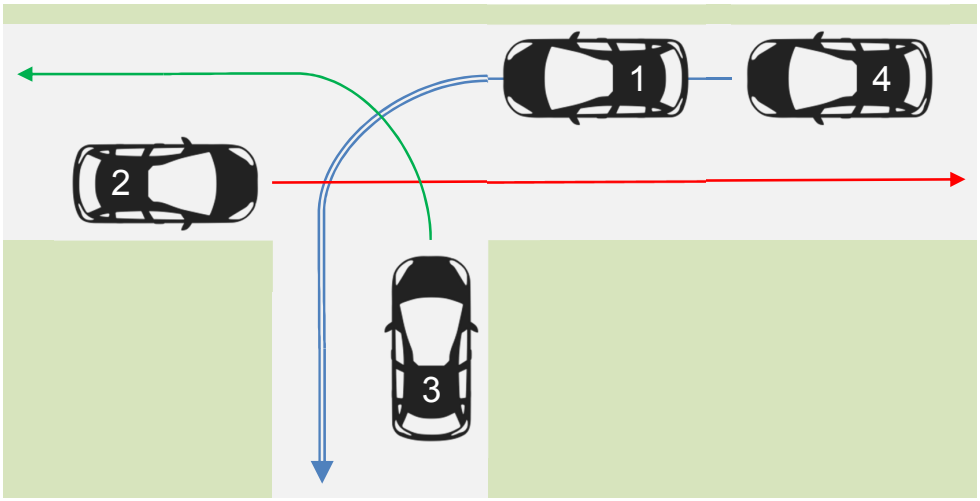


Figure 1 – Cooperative driving example: vehicles are steadily required to adapt their behavior to the current situation.

their states, trajectories, and goals to their peers. As soon as conflicting situations or opportunities for cooperation are detected, an appropriate communication scheme needs to be set up.

Furthermore, assume that vehicles 1 and 4 have formed a platoon to increase traffic efficiency, i.e. vehicle 1 is informing vehicle 4 about unforeseen maneuvers, particularly braking. To enhance throughput optimality, the priority decision should also depend on the length of the platoon. Obviously, vehicles 2 and 3 can only obtain information concerning the platoon configuration from vehicle 1.

In the automotive industry it is common to design vehicle software such as automated driver assistance systems (ADAS) using the C&C paradigm, for instance using a modeling tool like Simulink [BMR+17b]. Furthermore, to make the system cooperative each vehicle needs a communication system for message exchange with other cars. Thereby, messages received from other vehicles are fed into the input ports of the ADAS model. On the other hand, messages produced by the ADAS are transmitted through its output ports back to the communication unit which in turn sends them to the other vehicles through the vehicle network. Thereby, the number of the input and output ports depends on the communication partners of the respective vehicle. Moreover, depending on the situation to cope with, a vehicle may need to reconfigure its internal architecture. For instance, an intersection management component is only needed when there actually is an intersection, while a platoon manager is useless if there is actually no platoon. In both cases the structure depends on the number and kind of the participants.

3 Background & Requirements

Vehicle development processes aim at delivering safety critical software compliant with standards like ISO26262. An important part of such processes is the definition of a logical and a functional architecture abstracting away from technical details of the final implementation. In this phase, modeling tools and Architecture Description

Languages (ADLs) play a central role. In particular, the C&C modeling approach incorporated by tools like Simulink [Mat16], has become the predominant approach due to properties facilitating the design of complex systems, including the possibility of hierarchical decomposition, black-box testability, and clear data-flows free of side-effects [BMR+17b]. First-level citizens of a C&C model are components representing some self-contained functionality.

Using a C&C language, the design of an autonomous vehicle can be easily divided into smaller pieces, each developed by a dedicated team of engineers. The resulting modules, e.g. the *situation understanding*, the *trajectory planning*, and the *motion control*, can then be assembled to a working system based on the clear, strongly typed interfaces.

Unfortunately, C&C modeling languages are mostly focused on the description of static architectures. Architectural changes either cannot be realized at runtime at all or are limited to small sets of simple modifications. For the design of a cooperative vehicle, as introduced in section 2, the vehicle behavior steadily needs to be adapted to current circumstances. Depending on other cooperative traffic participants, a cooperative vehicle needs to add new data flows and data processing components to its logical architecture. Furthermore, it needs to exchange components based on environmental events. The modeling language should allow one to model such architectural changes accordingly, which leads to our main requirement:

(R0) Architectural reconfiguration The architecture modeling language must provide means to describe architectural changes of a system at runtime.

Based on our earlier considerations we have derived the following set of requirements on a dynamically reconfigurable C&C modeling language, which are all sub-requirements of (R0):

(R1) Dynamic components In cooperative systems the set of components needed to achieve a goal can change drastically over time. An ADL for dynamic systems must be able to model the instantiation of new and the removal or deactivation of unnecessary components.

(R2) Dynamic interfaces To enable data exchange with arbitrary communication partners the interface of a component should be alterable at runtime, as well. A dynamic ADL must therefore provide the means to request new and release old unnecessary ports. Nevertheless, type safety must be guaranteed at any point in time.

(R3) Internal event-triggerd reconfiguration A component should be able to initiate a reconfiguration based on events visible in the component's scope. Thereby, an event can be *data-* or *architecture-triggered*. A data-triggered event is defined in terms of values present at a component's ports. In contrast, architecture-triggered events depend on the creation or deletion of architectural elements, i.e. ports or components.

(R4) External service-based reconfiguration As opposed to (R3), a dynamic component should provide a reconfiguration interface, allowing other components as well as the runtime environment to request reconfigurations explicitly.

(R5) Blackbox reconfiguration A central idea of component-based software engineering is that components can be regarded and reused as blackbox units

only accessible through their interface. This principle must continue to hold for dynamic components. That is, a component cannot explicitly request changes in the internal structure of a subcomponent or a connected component. Consequently, (R4) only applies to interface reconfiguration requests and excludes invasive component reconfiguration. This also implies that a dynamic component has full sovereignty over its internal reconfiguration processes including their quality and duration. Furthermore, no details about the internal structure of a component are required to use its reconfiguration interface.

The dynamic C&C reconfiguration framework presented in this paper has been developed as an extension to the EmbeddedMontiArc language family [KRRvW17, KRSvW18, KRRvW18]. EmbeddedMontiArc is a textual C&C ADL focusing on modeling CPS by providing a strict matrix-based type system including matrix property assertions, SI-unit support, and a performance-optimized compiler toolchain. Lines 1-7 in fig. 2 show the basic syntax of EmbeddedMontiArc: first the main component is defined in line 1 using the `component` keyword followed by the component type name `BMux4` and optionally generic and/or configuration parameters. Next, the interface of `BMux4`, consisting of two input and one output ports, is declared in lines 2-4. Two of the ports are typed with the generic type parameter `T` while `ctrSig` is a scalar Boolean. Note that the two input ports are actually *port arrays* of a fixed size given in square brackets.

In EmbeddedMontiArc the behavior of a component can be modeled either using a so called implementation modeling languages or as an instantiation and reconnection of *subcomponents*. While the former is uninteresting from the architectural point of view, the latter is a central element for this work: in line 5 a named instance of the component type `BMux2` is created. In lines 6 and 7 the ports of the parent component are eventually connected to the ports of the subcomponent instance using the `connect` keyword.

4 Related Work

The intention of this section is to provide a short introduction to the field of dynamic C&C-based ADLs (with no claim to completeness). First we are going to introduce some important representatives one by one. Then, we proceed to a discussion of their dynamic reconfiguration capabilities according to the requirements derived in section 3. A tabular overview is given in table 1.

AADL [FG12]: Architecture Analysis & Design Language (AADL) is an ADL designed and used in the avionics and automotive domains. The language deals with both soft- and hardware architectures and provides means for analysis and verification of embedded systems. Reconfiguration can be modeled using *modes* and mode *transitions* defined in a mode finite state machine (FSM). A mode is a self-contained configuration state. The main purpose of reconfiguration is to switch between different operational states of an automobile or an aircraft, but also to compensate for hardware failures.

AutoFocus 3 [AVT+15]: AutoFOCUS 3 is a holistic methodology for the model-based design of embedded systems covering the development process from the requirement analysis to integration. It is based on the dynamic and static FOCUS theory [BS12] providing a foundation for formal verification and analysis of static and

dynamic architectures. Reconfiguration can be modeled using modes governed by FSMs.

Darwin [MDEK95]: Darwin is a π-calculus-based ADL for the specification of distributed systems. Dynamic aspects can be modeled in Darwin using lazy or dynamic instantiation. The former requires all the bindings of the architecture to be defined at design time. The latter allows the architecture to evolve in arbitrary ways. However, components cannot be removed or deactivated once they have been instantiated.

MontiArc [HRR12]: MontiArc is a modeling language for distributed architectures focusing on logical aspects of software architectures. Although the core language is constrained to static architectures, in later versions an extension supporting modes was retrofitted [HKR+16]. Similar to AutoFOCUS 3 and AADL, mode transitions are controlled by FSMs.

ROOM [RSRS99]: Originally developed for the design of telecommunication systems, ROOM is an ADL suited for the development of any event-driven real-time systems. It allows the replication of components and ports at runtime to represent dynamic architectures such as multiline telephone systems.

Simulink [Mat16]: Simulink is a graphical modeling tool widely used in engineering disciplines such as control and automotive, particularly for prototyping. In a 150% model, *Enabled Subsystems* and *Triggered Subsystems* allow the activation and deactivation of components at runtime based on external signals.

WRIGHT [ADG98]: WRIGHT is an ADL based on the communicating sequential processes (CSP) formalism. The formal nature allows for automated verification and consistence checks of architectures and architectural styles. Wright offers the possibility to reconfigure an architecture using a component setting pre-defined at runtime. Additionally, architectural elements can be instantiated or removed in an imperative manner.

In this comparison, we only considered ADLs supporting at least (R0), i.e. at least some kind of runtime reconfiguration. The actual realization varies from language to language. We can distinguish between imperative and declarative means of specifying a reconfiguration. For instance, WRIGHT implements an imperative approach providing actions such as `new` and `delete` to instantiate and remove architectural elements. On the other hand, declarative languages concentrate on *what* to do rather than *how* to do it. For instance, AutoFocus 3 and MontiArc provide modes governed by FSMs. Thereby, a mode declares the structure of a component in a specific state and the FSM describes possible state changes.

Dynamic components (R1) are at least partially supported by most of the presented languages. A restricted variant of component creation is provided by the mode concept, e.g. in MontiArc mode descriptions can contain individual component declarations. However, possible component instances have to be defined at design-time. Hence, we consider modes as a partial implementation of (R1) as long as mode transitions can initiate component creation. This is not the case for AADL and AutoFocus 3 as these languages only allow for a rewiring of the connectors or parameter changes at runtime. ROOM and WRIGHT are the only ADLs providing full support for component instantiation. Darwin, although supporting free component creation, lacks the possibility of component removal and thus fulfills the requirement

only partially. Simulink allows one to enable and disable components based on signal values using specialized ports. However, the components are always present in the model, hence, no component instantiation or removal is modeled.

Dynamic interfaces (R2) are a more invasive means for architectural modifications and are hence seen in a smaller number of ADLs. In ROOM, ports can be replicated at runtime, e.g. to handle arbitrary numbers of phone connections. In Darwin, services provided by dynamically created component instances can be offered as services of parent components (thereby changing the parent component's interface).

Internal reconfiguration (R3) Internal reconfiguration is supported by AutoFocus 3, MontiArc, and Simulink. In these languages, reconfiguration can be triggered by an internal trigger, such as an incoming port signal. For instance, in AutoFocus and MontiArc, a mode transition is activated if a corresponding condition of the mode FSM depending on some port values is met. In Simulink, signal values can be used to rewire the architecture using enabled and triggered subsystems. External reconfiguration can be simulated using internal reconfiguration since signals coming from other components can be reinterpreted as reconfiguration requests. Therefore, all languages supporting (R3) automatically support (R4). Darwin and WRIGHT are the only languages supporting **external reconfiguration (R4)** exclusively.

Self-directed blackbox reconfiguration (R5) is provided by all the discussed languages except AADL allowing for a blackbox reuse of dynamic components. In AADL a component's mode can be mapped to the mode of its parent component, i.e. the components' mode depends directly on the mode of its enclosing scope.

	AADL	Autofocus 3	Darwin	EMAD (this paper)	MontiArc	ROOM	Simulink	Wright
(R0) - Runtime reconfiguration	√	√	√	√	√	√	√	√
(R1) Dynamic component creation	–	–	p	√	p	√	–	√
(R2) Dynamic interface modification	–	–	p	√	–	√	–	–
(R3) Internal reconfiguration	–	√	–	√	√	–	√	–
(R4) External reconfiguration	?	√	√	√	√	√	√	√
(R5) Self-determined reconfiguration	–	√	√	√	√	?	√	√

Table 1 – Comparison of C&C modeling languages, √: yes, p: partially, –: no, ?: unknown

Further and more complete analysis of dynamic ADLs can be found, e.g. in [BHK+17, KJKD05]. Alternative approaches for the reconfiguration of robot system based on variability and feature modeling, meta-modeling, Domain Specific Languages (DSLs) exist.

In particular, the models@runtime paradigm is an important approach for the design of self-adaptive agents. Models@runtime-based systems usually consist of a managed core system as well as a reconfiguration framework which uses models to reason about the system's state and to adapt it if needed [BFCA14]. The reconfiguration space is highly dependent on the application.

For instance, a QoS-aware approach describing variability of UML MARTE models

[SG13] for robotics uses feature models [BCMT18] to describe the reconfiguration space. Feature modeling is a higher-level approach to reconfiguration focusing on *available* modules. Similar to modes, all possible feature configurations are provided in a static feature model.

In another self-adaptation framework a meta-controller evaluates the controller performance and reconfigures it when needed [HBAS18]. Thereby, the meta-controller uses a runtime model of the controller based on an ontology modeling the robot's missions and the controller architecture. Hence, there is no self-determined reconfiguration by the controller itself.

In contrast to the models@runtime approach, reconfiguration handled in this work is domain-independent and targets exclusively the system's C&C architecture. Thereby, possible reconfigurations are modeled (implicitly) at compile-time and neither require system models or model reasoning at runtime, nor a resource consuming reconfiguration framework. Reconfiguration is self-determined according to (R5).

5 EmbeddedMontiArc Dynamics

In this section we are going to present the main contribution of this work, namely, EmbeddedMontiArc Dynamics (EMAD), an extension of the MontiCore-based [HR17] C&C language family EmbeddedMontiArc by concepts allowing for the modeling of dynamic runtime reconfigurations. The aim of the extension is to fulfill the requirements elicited in section 3 and hence to enable modeling of interconnected systems as described in section 2.

EmbeddedMontiArc Execution Semantics EmbeddedMontiArc distinguishes between *distributed architectures* and *self-contained software architectures*. In a distributed architecture model components can be basically mapped to standalone processes. In our running example, a component might represent a cooperative vehicle. Connectors represent data-flows between the distributed actors of the overall system, e.g. status messages sent between the vehicles over a network. Communication and execution are handled by a middleware which can be provided explicitly in a separate middleware model [HKKR19]. The execution semantics of such a distributed model depends on the chosen middleware, but is in general asynchronous: each component is executed completely independently of other parts of the overall architecture with an individual resolution and frequency.

On the other hand, self-contained software architectures describe the structure of a single process, i.e. the components and data-flows *inside* the cooperative vehicle. Components present in a self-contained architecture model might range from high-level blocks such as trajectory planners, controllers, and platoon managers to low level elements including logical gates. This modus operandi most resembles the Simulink approach of modeling technical systems. A model is supposed to be run in a single process, hence, in this use case EmbeddedMontiArc has a synchronized, weakly causal execution semantics compatible to Simulink: at compile-time, an execution order based on the dataflow is computed. At runtime, the components of the architecture are executed sequentially in this order. Thereby, values at the output ports of a component computed in an execution cycle n are made available to successor components immediately (in contrast to asynchronous strongly causal architectures, where components are executed in parallel and thus the results of a component computed in the execution cycle n are only available to other components

in the succeeding execution cycle $n+1$). This semantics is more natural and efficient in the absence of inter-process communication.

```
component BMux4<T>{
  ports in T inSig[4],
        in B ctrSig[2],
        out T outSig;

  instance BMux2<T> mux2;
          default behavior of component
  connect ctrSig[1]    ->  mux2.ctrSig;
  connect mux2.outSig  ->  outSig;
          mode condition
  @ ctrSig[2]::value(true) {
    connect inSig[3]  ->  mux2.inSig[1];   Value-triggered
    connect inSig[4]  ->  mux2.inSig[2];   reconfiguration
  }

  @ ctrSig[2]::value(false) {
    connect inSig[1]  ->  mux2.inSig[1];   Alternative
    connect inSig[2]  ->  mux2.inSig[2];   reconfiguration
  }
}
```

Figure 2 – A four multiplexer component as an example for a reconfigurable component.

In a full system model defined in EmbeddedMontiArc it is natural to combine the two execution variants: while top level components represent nodes of a distributed architecture requiring a complex and expensive communication scheme, e.g. using a middleware such as ROS [QCG+09], their subcomponents can be generated to tightly coupled code. The EmbeddedMontiArc-to-C++ generator realizes connectors of synchronous components as simple C++ function calls [KRSvW18].

To enable reconfiguration and to support dynamically evolving architectures, we adapt the execution semantics of EmbeddedMontiArc by introducing a reconfiguration phase. In both types of architectures, we define the reconfiguration phase to take place directly before the execution of the component's code. Thereby, based on the requirements presented in section 3, we distinguish between three fundamental reconfiguration approaches in our work:

1. Data-triggered reconfiguration,
2. Service-based reconfiguration,
3. Modes.

Modes Since modes are a well-known means for architectural reconfigurations supported by many ADLs, we don't want to cover them in detail. Instead we want to use them as a reference for the discussion of the two other approaches. Modes are particularly well-suited to model systems choosing their behavior from a given static set of states pre-defined at compile-time. Thus, a natural way to describe states

and their transitions is by using FSMs or related modeling techniques such as state charts and to assign a concrete configuration to each of the states. Consequently, the activation of a particular mode is not triggered solely by the occurrence of an event, but also depends on the actual state of the mode FSM.

Data-triggered internal reconfiguration A straight-forward way to model reconfiguration is the data-triggered approach. The idea is to activate architectural elements *as long as* a condition is satisfied. This resembles best electronic and mechanic systems: a diode is active as long as the applied voltage is higher than the threshold voltage; a vehicle's charging electronics is active as long as a connector of the charging station is plugged in; highway driving assist is active as long as the vehicle's sensors perceive a highway ride.

To define data-triggered reconfiguration, we extend the body of an EmbeddedMontiArc component definition by an arbitrarily long sequence of reconfiguration definitions. A reconfiguration definition is initiated by the reconfiguration head, representing the event causing the reconfiguration, and followed by the reconfiguration body containing the architectural changes to be realized.

The example in fig. 2 contains two such reconfigurations each initiated with an @ symbol. Reconfiguration conditions are defined as boolean expressions. Thereby, we provide the possibility to query the port value using the `port::value()` function, where `port` is a placeholder for a port name visible in the component's scope. This means, a component can query its own ports as well as ports of its direct subcomponents, but no ports of its parent's components or its subsubcomponents thereby ensuring a self-directed blackbox reconfiguration.

To take into account past values at the port, i.e. to initiate a reconfiguration only if a certain value *sequence* has been observed at the port, it is possible to compare the port value with a sequence. For instance, the condition ctrSig[2]::value() == [true,false,true] evaluates to true at execution cycle n if the value at this port was `true` at $n-2$, `false` at $n-1$ and is `true` at n again.

The reconfiguration bodies in lines 9 and 10 for the first reconfiguration and 13 and 14 for the second one contain ordinary `connect` declarations as we use them in static EmbeddedMontiArc body syntax. The behavior behind this example model is interpreted as follows: the component instance `mux2` of type `BMux2` is a multiplexer with two data input ports `inSig[1]` and `inSig[2]` of a generic type `T`. Furthermore it has a Boolean control input `ctrSig`, cf. line 6, choosing which of the two data inputs to forward to its output port `signalOut`. The component instance `mux2` is wrapped by `BMux4` having *four* data input ports. Its second control port `ctrSig[2]` is used in the two reconfiguration conditions (lines 8 and 12) to choose which two of the four data ports to connect with the inner component `mux2`. A graphical representation of the model is depicted in fig. 3. The configuration on the left is used whenever the value at the `controlIn[2]` port is true. The architecture on the right is active otherwise.

This example shows the basics behind EmbeddedMontiArc Dynamics, but only has static reconfiguration elements: all possible architectural states of `BMux4` are explicitly given in the model. The same behavior can be modeled using modes. A corresponding mode FSM has two states corresponding to the given reconfigurations with transition guards equivalent to the respective reconfiguration conditions. In this case using modes has the advantage of having an explicit FSM facilitating model analysis, e.g. looking for unreachable states or underspecified state transitions, i.e. checking for determinism. On the other hand, the notation presented here is much more convenient if several subcomponents need to be activated and deactivated independently. A mode

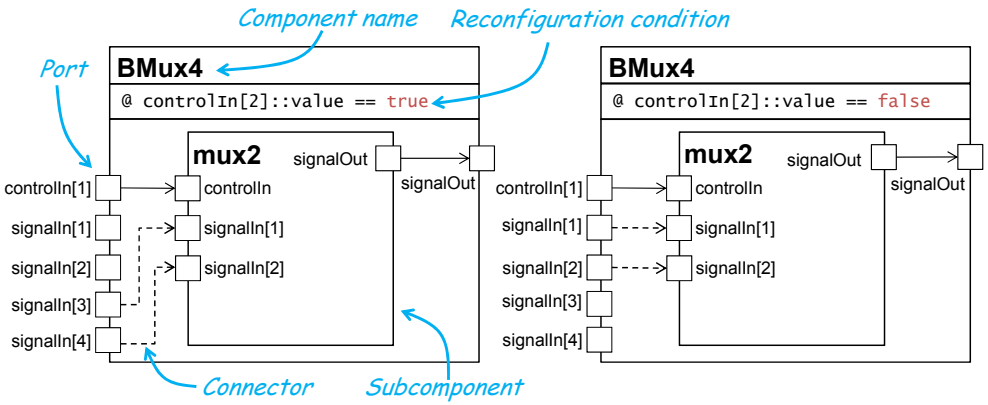


Figure 3 – The two architectural states of the BMux component.

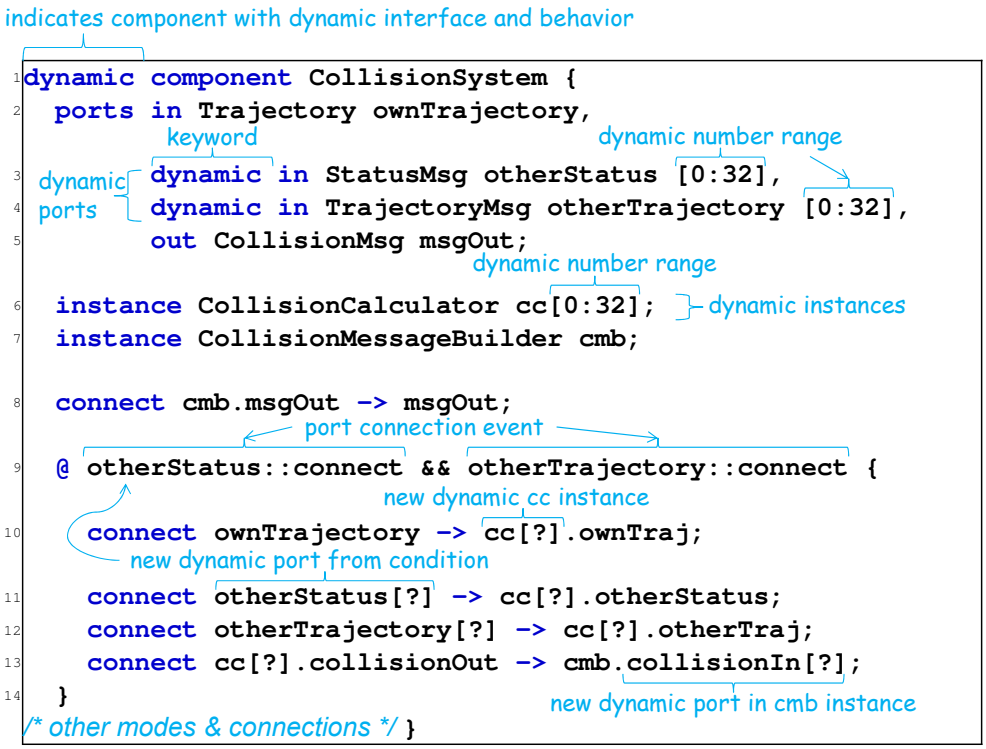
automaton would loose expressiveness and become difficult to maintain. We therefore recommend using the two approaches interchangeably depending on the concrete modeling task if only static reconfigurations are necessary.

Service-based reconfiguration Statically sized component and port arrays were introduced in EmbeddedMontiArc in order to facilitate modeling of large systems. For instance, if we wanted to model each vehicle of a vehicle network by a dedicated component, we could do so by using an array of vehicle components. If a single vehicle needs to receive messages from a set of peers, the interface can be defined using a port array where each port in the array is assigned to a particular peer.

A central element of our reconfiguration framework is the introduction of dynamic component and port arrays. In most real world situations we do not know at design-time, how many vehicles will appear in a certain region of interest at a particular point in time. Hence, we need to be able to create and delete components and ports on demand.

Consider `CollisionSystem` component depicted in fig. 4. In lines 3 and 4 we have a definition of dynamic port arrays marked with the keyword `dynamic`. Note that in contrast to fig. 2 where an integer number in square brackets denotes the number of ports in the array, in the dynamic case we specify a size *range*. The port arrays `otherStatus` and `otherTrajectory` receiving status and trajectory messages from surrounding cooperative vehicles can thus have up to 32 entries. If no other vehicles are around, the port array can be empty, as well. A similar syntax is used to denote component instance arrays. For each vehicle, an individual `CollisionCalculator` component is created in line 6. At runtime, when the collision system is instantiated, the minimum number of components and ports is instantiated, since only this minimum number of components is in the scope of the initial set up, i.e. available for connectors (cf. connect statement). Note that although an unlimited range can be easily modeled in EmbeddedMontiArc using the infinity symbol `oo`, we forbid such architectures at compile-time by context conditions. In embedded systems, we often need to deal with limited resources and therefore force the system designer to explicitly model the system's limits.

So how can the remaining *dynamic* ports and components be instantiated and used? As mentioned in (R4) we need a *reconfiguration service interface* allowing external components to request reconfigurations. Imagine a vehicle's communication




```
1  dynamic component CollisionSystem {
2    ports in Trajectory ownTrajectory,
3          dynamic in StatusMsg otherStatus [0:32],
4          dynamic in TrajectoryMsg otherTrajectory [0:32],
5          out CollisionMsg msgOut;
6    instance CollisionCalculator cc[0:32];
7    instance CollisionMessageBuilder cmb;
8    connect cmb.msgOut -> msgOut;
9    @ otherStatus::connect && otherTrajectory::connect {
10     connect ownTrajectory -> cc[?].ownTraj;
11     connect otherStatus[?] -> cc[?].otherStatus;
12     connect otherTrajectory[?] -> cc[?].otherTraj;
13     connect cc[?].collisionOut -> cmb.collisionIn[?];
14   }
/* other modes & connections */ }
```

Figure 4 – Collision system of autopilot which calculates all collisions with other vehicles

system reading beacon messages from a broadcast channel. At some point the decoder finds messages from a new traffic participant in this stream and needs to inform the `CollisionSystem` that from now on there is a new peer whose `StatusMsg` and `CollisionMsg` it would like to forward to a dedicated port of the `CollisionSystem` component.

To provide the means for such an interaction, the `CollisionSystem` has a reconfiguration definition given in lines 9-14. In its basic structure it is similar to the reconfiguration blocks of the `BMux4` component discussed above. However, instead of querying the *value* of a port, we employ the `port::connect` event in this condition. A `port::connect` event is activated if the designated port receives a connect request (outgoing or incoming) or if a connector is actually attached to this port, e.g. if a parent component connects to it in a reconfiguration block. For instance, lines 9 and 10 in fig. 3 cause a connect event for the `inSig` ports of `mux2`. Consequently, `mux2` could catch this event in its own reconfiguration blocks and react to it if it needed to.

The reconfiguration condition in line 9 of fig. 4 implicitly declares a *reconfiguration interface* for the `CollisionSystem` component. A reconfiguration can only be initiated if the whole condition is fulfilled. In our example this means that a connection request to an `otherStatus` *and* to an `otherTrajectory` port have to be present. Note that these port arrays are declared as `dynamic` and thus have a dynamic size. The semantics of the reconfiguration realized by the EmbeddedMontiArc Dynamics compiler boils down to the following steps:

1. Request: an external component sends a set of connect requests. These connect requests might arise from the requester's own events (i.e. from connect statements in an event body) or from the C++ API of the generated code (stay tuned).

2. Reservation: The receiving component checks if the requested ports are available, i.e. if the corresponding dynamic port arrays do not violate their respective upper limit constraint. If yes, the component returns IDs for the new ports, i.e. the newly allocated array indices, to the requester so that explicit access is possible in the future. Otherwise, the requester is informed that its request has been rejected.

3. Reconfiguration: at the reconfiguration phase of the component, the reconfiguration bodies of all valid reconfiguration requests, i.e. those complying with a reconnection conditions are realized (lines 10-13 in the `CollisionSystem` example). Consequently, the component reacts to the external reconfiguration event by internal self-modifications.

4. Follow-up request: possibly, the reconfiguration instructions of the previous step contain the creation of new ports and/or subcomponents, as well. In this case, the component becomes a requester itself initiating a follow-up reconfiguration in its sub-components or external components.

Modeling access to new dynamic ports and components is made easy using the `?`-operator. While `otherStatus[n]` denotes an access to the `n`-th port of the `otherStatus` port array, `otherStatus[?]`, cf. line 11 in fig. 4, indicates access to a new port in the array. Modeling the array access in this way abstracts away from the actual element position in the port (or component) array making the syntax independent from the technical realization. Indexing and inserting new elements is delegated to the reconfiguration code generated by the EmbeddedMontiArc compiler. It is important to note that the `?`-operator has a scope similar to a local variable: there

are four references to the dynamic component `cc[?]` in lines 10-13. All of them denote the same new `CollisionCalculator` component created in this reconfiguration block. Note that for the sake of simplicity, we omit an explicit mention of the `?`-operator in the reconfiguration condition.

In contrast to value-triggered reconfiguration, service-based reconfiguration is persistent in the sense that a requested connection remains present in the following execution cycles even if the port is not requested again. However, a dynamic port also provides a `port::free` interface allowing to remove it from the architecture. The `port::free` interface cannot be modeled explicitly, but is used implicitly to remove dynamic ports once the respective creation condition is invalidated. If, for instance, a communication partner has disappeared or canceled the connection, the vehicle's communication system will not find messages from this vehicle in its input signal any more. The corresponding data-driven reconfiguration condition will cease to hold. This will cause a *reverse reconfiguration*, i.e. the connectors, ports, and components created due to this data-driven event will be unrolled. This process propagates to all other components that were involved in the original reconfiguration sequence, e.g. subcomponents of `CollisionSystem`, due to some follow-up service-based reconfiguration events triggering their reconfigurations to be undone as well. An exception are outgoing ports: since an outgoing port can have multiple connectors attached to it, we only remove it when *all* of its leaving connectors have been removed. This mechanism ensures that an architecture can always return to any state it has been in before, no matter how many reconfigurations have taken place. This is in contrast to languages like Darwin, where components cannot been removed from an architecture once they have been created.

For the targeted domain, the combination of a service-based persistent creation and a data-driven activation of dynamic architectural elements turns out to be a powerful symbiosis. In particular, the combination boils down to a simple architectural pattern: usually a data-driven event is used as an origin for a complex reconfiguration while a chain of service-based follow-up events creates the required architectural infrastructure. Once the original trigger disappears, the changes are unrolled to the initial state. To illustrate this reconfiguration chaining mechanism, let's have a closer look at the parent component of the `CollisionSystem`, namely the `CoOpAutopilot` component responsible for driving decisions. To provide basic driving functionality, the `CoOpAutopilot` receives sensor signals through the corresponding sensor ports. To enable cooperative interaction with other traffic participants we introduce three other message types: `CollisionMsg` contains detected collision and priority information. This message type is used to resolve conflicting situations. The `TrajectoryMsg` type contains planned trajectory information of the vehicle. It can be used by other vehicles to compute potential collisions. The `StatusMsg` contains information about the sender, including current position, velocity, and other sensor information.

Each cooperative agent produces messages of these types and broadcasts them to all other traffic participants. Hence, each cooperative agent receives a status, trajectory, and collision messages from many participants. These messages cannot be exchanged by `CoOpAutopilot` components directly. Instead, we need a communication system, i.e. an antenna and further processing, to receive and send messages over a vehicle-to-vehicle network. We hence can assume that the input and output ports related to vehicle messages are connected to a communication component. The task of such a component is to receive and send messages, but also to forward received messages to the correct, indexed (or qualified) input ports of the `CoOpAutopilot`.

Once a signal received by the communication component contains the signature of a new vehicle, a data-driven event is triggered and as a consequence a connection to each of the three dynamic ports of the `CoOpAutopilot` is created. The induced reconfiguration is depicted graphically in fig. 5.

Graphical Notation The corresponding reconfiguration is depicted in fig. 5 as a graphical model, more precisely as a *reconfiguration view*. Views are generally used in C&C modeling to emphasize particular structural aspects of a system, i.e. the hierarchical relation of several components or information flows [BMR+17a, MMR+17, KKRvW18]. We introduce reconfiguration views as an extension to static C&C diagrams, e.g. SysML internal block diagrams, to cover the reconfiguration procedures of an architecture. The diagram syntax is straight-forward: a reconfiguration condition is given as Boolean expression below the component name at the top. The graphical part depicts newly created architectural elements with dashed lines, namely the message ports from the condition, their respective connectors, as well as the target ports of the `CollisionSystem` and the `PlatoonManager` components (to be precise, the latter are not created, but requested during the reconfiguration).

As is inherent for views, only the parts relevant for the depicted reconfiguration are shown in the diagram. In particular, no architectural elements are shown that are not affected by the reconfiguration in some way. For instance, components of the `CoOpAutopilot` responsible for trajectory planning and control are omitted in fig. 5. This ensures a clean encapsulation and separation of concerns in graphical reconfiguration modeling.

Since the `CollisionSystem` itself performs a kind of an *aggregation* operation, the reconfiguration chain does not affect any output ports which is why no output ports or outgoing connectors are depicted in fig. 5. The reconfiguration chain is completed after this reconfiguration.

A chain of similar reconfigurations involving inner component reconfiguration and the creation of new output ports is depicted in fig. 6. The idea is that once a vehicle spots another vehicle it wants to follow in a platoon, it starts generating platoon messages. Similarly to the collision avoidance case described above, this causes the communication system to request a new port for platoon messages, but also forces the followed vehicle to produce platoon messages on its own, as well. This leads to the reconfiguration depicted in the left sub-diagram of fig. 6: a connector is created and connected to a dynamic port of the subcomponent `PlatoonManager`.

The reaction of the `PlatoonManager` is hidden from the `CoOpAutopilot` due to our blackbox assumption. However, the reconfiguration in the right subdiagram is obviously a reaction to changes *inside* the `PlatoonManager`.

Until now, we have discussed event-based data-driven reconfiguration, and service-based reconfiguration, cf. fig. 3 and fig. 5, respectively. The reconfiguration in the right part of fig. 6, however, is event-based and *architecture-driven*: in contrast to the left subdiagram and fig. 5 it is triggered by the creation of a subcomponent port by this very subcomponent. The reaction is once again the creation of a connector as well as a new output port. The latter might trigger further reconfigurations in an observing parent component.

Generative Aspects of EmbeddedMontiArc Dynamics We have not modeled the communication system of the cooperative vehicle as an EmbeddedMontiArc component to illustrate another concept: integration of EmbeddedMontiArc-based architectures into arbitrary environments. EmbeddedMontiArc models are generated

as C++ code and middleware adapters if necessary. Consequently, the code generated from the models can be integrated into any software project using the native C++ interfaces or the chosen middleware, e.g. to set port inputs or read port outputs.

Integration with non-model-driven legacy software is an important feature, particularly for large-scale projects and must continue to hold for EmbeddedMontiArc Dynamics, as well. Therefore, we extend the generated interfaces to support service-based reconfiguration, i.e. an external piece of software is enabled to request ports by invoking the `connect` interface of a dynamic port. In contrast to EmbeddedMontiArc Dynamics syntax, the generated C++ interface allows for an explicit removal of a port or connection by calling the port's `free(id:int)`-interface. This is necessary, since a service-based reconfiguration initiated by an external caller is the first reconfiguration in the event chain and has no means to be undone otherwise. Note that requesting reconfigurations via the C++ interfaces triggers the same events as if the reconfiguration was defined in the model. If our external communication system requests new ports of the types `StatusMsg`, `CollisionMsg`, and `TrajectoryMsg`, the `CoOpAutopilot` would react as if the request came from an other EmbeddedMontiArc component.

Remarks on Architectural Consistency Dynamic reconfiguration of software architectures imposes issues concerning model consistency and correctness not present in static architectures. Avoiding such inconsistencies requires constraining the available reconfiguration space. We ensure this by several architectural checks at compile-time. First, the creation of a subcomponent is only allowed if the same reconfiguration block contains a minimum set of connectors required by this component. This ensures that an architecture never creates orphan components not having any inputs or outputs. Second, explicit removal of any architectural elements is forbidden in the reconfiguration model. Instead, reconfigurations are seen as atomic procedures and are also unrolled as atomic procedures if a required condition is invalidated.

Third, reconfigurations involving port request to subcomponents are checked against the reconfiguration interface of the respective subcomponents. For instance, in fig. 5 the `CoOpAutopilot` provides a reconfiguration condition requiring three ports to be connected at one go. If its parent component defined a reconfiguration only involving one or two of these ports, the reconfiguration would be invalidated by the compiler. This is trickier to ensure if the generated C++ code of the dynamic architecture is interfaced by external software as discussed in the previous paragraph. The generated C++ level interfaces must forbid reconfiguration requests unsupported by the model. While the most intuitive solution is to generate a `connectPortName()` function for each dynamic port of the model, this solution would require our communication system to call the functions `connectCollisionMsg()`, `connectTrajectoryMsg()`, `connectStatusMsg()` in a sequence in order to trigger fig. 5. A type-safe solution guaranteeing correct port requests at compile-time is to generate a single method for the reconfiguration condition. This leads us to the constraint forbidding the `or` operator in reconfiguration conditions. Otherwise the combinatorial complexity would explode for large models with many dynamic ports. This however is not a problem, since alternatives are difficult to handle in a single reconfiguration block. Instead, we split conditions of the form $a \wedge (b \vee c)$ in three distinct reconfiguration events $a \wedge b \wedge \neg c$, $a \wedge \neg b \wedge c$, and $a \wedge b \wedge c$ with clear individual reconfiguration semantics.

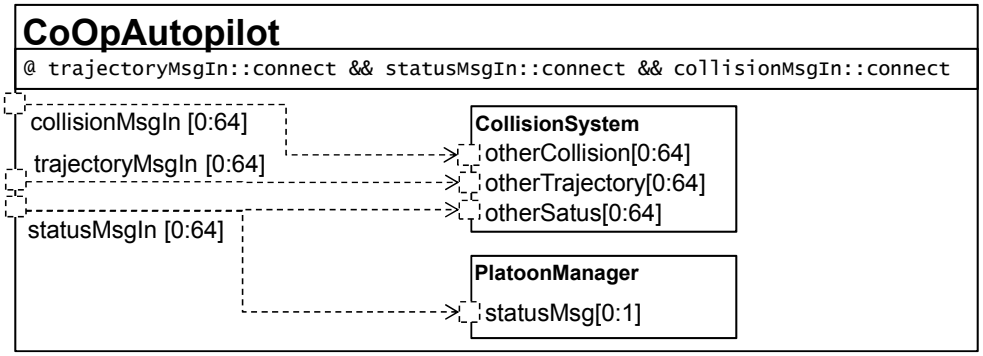


Figure 5 – Reconfiguration view of the CoOpAutopilot component depicting the dynamic ports and connectors created when a new cooperative vehicle is spotted. Static architectural elements are omitted in reconfiguration views to maintain readability.

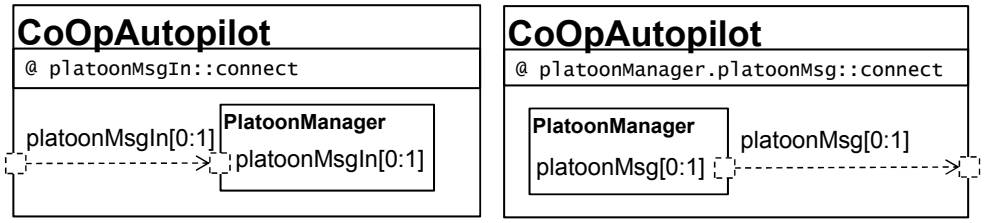


Figure 6 – A reconfiguration chain involving in and output ports of the `PlatoonManager` component. An arriving platoon message causes the creation of new input ports in the diagram on the left. Follow-up reconfigurations inside the `PlatoonManager` result in a new output port and a new outgoing connector as depicted in the diagram on the right.

6 Conclusion

In this work we designed a dynamic reconfiguration framework for component-and-connector architecture description languages. The methodology combines event-triggered events with service-oriented reconfiguration interfaces to enable a declarative modeling of dynamic self-adaptation of cyber-physical systems. The concepts are implemented as an add-on to EmbeddedMontiArc, a language family for model-based architecture-centric system design and demonstrated using model excerpts from a cooperative driving project. Thereby, we introduced dynamic component arrays and interfaces enabling the creation of new component instances and ports at runtime. These concepts can be easily used in the same way as standard EmbeddedMontiArc architectural elements in a declarative manner. Despite a high degree of flexibility, model consistency can be guaranteed by a variety of compile-time checks. A seamless integration with external software is made possible by generating reconfiguration interfaces for the target platform.

We believe that, enabling a natural modeling of self-adaptive cooperating systems, a reconfiguration system as the one presented in this work can be a valuable extension to component-and-connector modeling tools widely used in engineering disciplines like automotive. The introduced graphical notation can be employed for graphical solutions like Simulink to keep the design expressive and maintain a high degree of separation of concerns.

References

[ADG98] Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *International Conference on Fundamental Approaches to Software Engineering*, pages 21–37. Springer, 1998.

[AVT+15] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. Autofocus 3: Tooling concepts for seamless, model-based development of embedded systems. In *ACES-MB&WUCOR@ MoDELS*, pages 19–26, 2015.

[BCMT18] Davide Brugali, Rafael Capilla, Raffaela Mirandola, and Catia Trubiani. Model-based development of qos-aware reconfigurable autonomous robotic systems. In *2018 Second IEEE International Conference on Robotic Computing (IRC)*, pages 129–136. IEEE, 2018.

[BFCA14] Nelly Bencomo, Robert B France, Betty HC Cheng, and Uwe Aßmann. *Models@ run. time: foundations, applications, and roadmaps*, volume 8378. Springer, 2014.

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A classification of dynamic reconfiguration in component and connector architecture description languages. In *4th International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp'17)*, volume 1, 2017.

[BMR+17a] Vincent Bertram, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Case Study on Structural Views for

Component and Connector Models. International Conference on Model-Driven Engineering and Software Development, 2017.

[BMR+17b] Vincent Bertram, Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Component and Connector Views in Practice: An Experience Report. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 167–177. IEEE, September 2017.

[BS12] Manfred Broy and Ketil Stolen. *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer Science & Business Media, 2012.

[FG12] Peter H Feiler and David P Gluch. *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley, 2012.

[HBAS18] Carlos Hernández, Julita Bermejo-Alonso, and Ricardo Sanz. A self-adaptation framework based on functional knowledge for augmented autonomy in robots. *Integrated Computer-Aided Engineering*, (Preprint):1–16, 2018.

[HKKR19] Alexander Hellwig, Stefan Kriebel, Evgeny Kusmenko, and Bernhard Rumpe. Component-Based Integration of Interconnected Vehicle Architectures. In *Intelligent Vehicle Symposium (IV'19). Workshop on Cooperative Interacting Vehicles*. IEEE, 2019.

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, volume 9839 of *LNCS*, pages 175–182. Springer, December 2016.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[KJKD05] Mohamed Hadj Kacem, Mohamed Jmaiel, Ahmed Hadj Kacem, and Khalil Drira. Evaluation and comparison of adl based approaches for the description of dynamic of software architectures. In *ICEIS (3)*, pages 189–195, 2005.

[KKRvW18] Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Michael von Wenckstern. Finding Inconsistencies in Design Models and Requirements by Applying the SMARDT Process. Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme XIV (MBEES'18), Univ. Hamburg, April 2018.

[KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[KRRvW18] Evgeny Kusmenko, Jean-Marc Ronck, Bernhard Rumpe, and Michael von Wenckstern. EmbeddedMontiArc: Textual modeling alternative to Simulink. In *EXE at MODELS*, 2018.

[KRSvW18] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018.

[Mat16] Mathworks Inc. Simulink User's Guide R2016b. *Technical Report*, 2016.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *European Software Engineering Conference*, pages 137–153. Springer, 1995.

[MMR+17] Shahar Maoz, Ferdinand Mehlan, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Ocl framework to verify extra-functional properties in component and connector models. In *ModComp at MODELS*, 2017.

[QCG+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. ROS: an Open-Source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[RSRS99] Bernhard Rumpe, Maurice Schoenmakers, Ansgar Radermacher, and A Schurr. Uml+ room as a standard adl? In *Proceedings Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)(Cat. No. PR00434)*, pages 43–53. IEEE, 1999.

[SG13] Bran Selic and Sébastien Gérard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. Elsevier, 2013.

About the authors

Nils Kaminski holds an M.Sc. in computer science. He graduated from the RWTH Aachen University with a master thesis on the dynamic reconfiguration in C&C languages. Contact him at `nils.kaminski@rwth-aachen.de`

Evgeny Kusmenko is a Ph.D. student in the Software Engineering group of the RWTH Aachen University working on model-based development of cyber-physical systems. Contact him at `kusmenko@se-rwth.de`, or visit `http://www.se-rwth.de/staff/~kusmenko`.

Bernhard Rumpe is full professor and head of the Software Engineering group of the RWTH Aachen University. Contact him at `rumpe@se-rwth.de`, or visit `http://www.se-rwth.de/staff/rumpe/`.

Acknowledgments This work was supported by the Grant SPP1835 from DFG, the German Research Foundation.