Simulating Self-Adaptive Component-Based Systems using MATLAB/Simulink

Christian Heinzemann, Jan Rieke, Wilhelm Schäfer
Software Engineering Group
Heinz Nixdorf Institute, University of Paderborn
Zukunftsmeile 1, 33102 Paderborn, Germany
Email: [c.heinzemann | jrieke | wilhelm]@uni-paderborn.de

Abstract—The automotive industry uses MATLAB/Simulink models for specifying the behavior of software components and for early validation of that behavior using model-in-the-loop simulations. During a simulation run, these models may not structurally change. Thus, MATLAB/Simulink is not amenable to realizing self-adaptation behavior, where the software architecture of the system evolves during runtime. In this paper, we show how to model self-adaptive software using our language MechatronicUML and how we transform a model specified in MechatronicUML into a MATLAB/Simulink model automatically. In particular, we generate several helper functions that emulate self-adaptive behavior in MATLAB/Simulink, relying only on standard Simulink blocks. We illustrate our approach using an example of car platoons.

I. INTRODUCTION

Today, the automotive industry increasingly applies model-driven techniques for developing the software for their cars. Current development approaches [1] employ the AUTOSAR standard [2] for defining a software architecture of the system and MATLAB with its toolbox Simulink for defining the behavior of the software components. Central to this approach is the early validation of the software functions using model-in-the-loop (MIL) simulation in Simulink. In a MIL simulation, a model of the software is validated against a model of the environment.

A Simulink simulation model is static, i.e., once specified, the components and their connections in the simulation model may not change while running a simulation. That prevents the specification and early validation of functions that require structural changes in the software architecture. Examples of such functions are self-healing to recover from hardware failures [3] or car-to-car communication where communication partners change frequently.

In this paper, we use car platoons as an of a self-adaptive system that uses such car-to-car communication. In a car platoon, cars drive at low distances for reducing the energy consumption and increasing the throughput on a highway [4]. In current approaches, the leading car is usually operated manually and all remaining cars, called followers, operate autonomously and are controlled by software only. In a platoon, the leading car must notify all followers about a braking maneuver to prevent accidents. Consequently, all vehicles that want to join the platoon need to register with the leader.

Such functionality requires a self-adaptation [5] of the software architecture at runtime by using structural reconfiguration [6] i.e., instantiating, removing, and reconnecting software components during runtime. In our example, joining a platoon requires a follower to instantiate the software for driving autonomously and to replace the current velocity controller by a distance controller that considers the distance to the preceding car. The leading car needs to instantiate a software component for managing the platoon members. That, in turn, requires to manage a varying number of communication links with varying communication partners, because cars may join or leave the platoon at any time. Specifying the necessary runtime reconfiguration and message-based communication functionality is not natively supported by Simulink. Thus, supporting such functions in a MIL simulation requires the manual implementation of workarounds, which is time-consuming and error-prone.

In literature, there exist only few approaches for realizing self-adaptation in Simulink. They support the simulation of reconfigurable FPGA hardware [7], [8], but they do not consider adding/removing components to/from a simulation model and message-based communication behavior. Approaches targeting reconfiguration of AUTOSAR-based automotive systems [9], [10], [11] only support later phases of the development process. All of these approaches have in common that they do not provide concepts for early validation of MIL simulation in Simulink. Corresponding efforts for MIL simulation in Modelica [12] require language extensions and, as a consequence, special simulators [13], [14]. That, in turn, prevents the use of standard production code generators for producing the target code for the system.

In this paper, we extend our approach of [15] towards supporting MIL simulation of a self-adaptive system in Simulink without extending Simulink. In particular, we use our modeling language MechatronicUML [16], [17], [18] for specifying a reconfigurable software architecture. We use a rule-based specification based on graph transformations [19] for specifying reconfigurations that is called component story diagrams [20]. These rules encode the knowledge of which reconfiguration is necessary in which particular situation. Since MechatronicUML natively supports reconfigurable software architectures, the resulting rule-based specification is compact and easily maintainable. In this paper, we restrict ourselves to a fixed set of rules and leave the integration of further rules (e.g., encoding machine learning algorithms) for future works.
As our core contribution, we illustrate how we translate a MechatronicUML model of a self-adaptive system to MATLAB/Simulink. Therefore, we automatically derive all possible configurations of the software including the transitions between configurations from the MechatronicUML specification. This information serves as input for an automatic model transformation that generates a Simulink model emulating the reconfiguration behavior. In particular, we automatically generate a set of Embedded MATLAB Functions that encode the reconfiguration behavior of the system, which relieves the developer from implementing such functionality manually. By using only native features of Simulink for emulating reconfiguration, we enable the use of off-the-shelf production code generators for producing the target code. We illustrate our approach using the aforementioned example of a car platoon.

The paper is structured as follows. In Section II, we provide a short overview of MATLAB/Simulink. Section III introduces MechatronicUML, which we use for specifying the reconfigurable software architecture. We provide an overview of the transformation of non-adaptive MechatronicUML models to Simulink in Section IV before giving an overview of our core contribution, the transformation of a reconfigurable software architecture, in Section V. Sections VI to VIII provide details on the different steps of the transformation. We discuss related work in Section IX before concluding the paper in Section X.

II. MATLAB/SIMULINK

MATLAB is a commercial tool for numerical computation and visualization, which is developed by The MathWorks. The Simulink toolbox is an extension for MATLAB for modeling and simulating technical systems. In particular, it supports the specification and simulation of systems with continuous dynamics such as cars.

A Simulink model consists of blocks connected by signals. Simulink provides a wide range of basic blocks, e.g., for mathematical operations, signal generation and routing. Blocks can be grouped hierarchically using subsystems. A special kind of subsystem is the enabled subsystem shown in Figure 1. The enabled subsystem NormalDriveController is only active if the control signal at the top has a value greater than 0. If it is not active, it is not executed.

“An enabling signal is a time-varying quantity that has values at all points in time” [21]. It has a name, a data type, and a dimension, where the dimension is used for specifying arrays. Signals can be grouped into buses. A bus may contain signals of different types that are accessed by their name. We use buses for encoding messages; as we explain in Section IV.

Simulink models may also be extended by Embedded MATLAB Functions. Embedded MATLAB is a subset of the MATLAB scripting language that can be generated into efficient real-time C code using the Simulink Coder.

III. MechatronicUML

MechatronicUML [16], [17] is a model-driven design method that adapts concepts of the UML to model software for self-adaptive mechatronic systems. Mechatronic systems like cars consist of mechanical, electrical, and software parts that, in combination, realize the behavior of the system. MechatronicUML follows the component-based approach [22] where the software architecture of a system is specified by a set of interacting components.

In our component model [18], we distinguish between atomic components and structured components. Atomic components contain a behavior specification while structured components are assembled by embedding other (atomic or structured) components. A structured component may optionally specify a reconfiguration behavior [23]. In case of atomic components, MechatronicUML distinguishes between software components and continuous components. A software component contains a stateful discrete behavior specification while a continuous component defines a quasi-continuous behavior specification based on differential equations. We use continuous components to represent the feedback controllers of the system that control the physical system. Their behavior is specified by control engineers in Simulink directly.

![Fig. 1. Enabled Subsystem](http://www.mathworks.com/products/simulink)

Figure 2 shows an example of a structured component which is a simplified excerpt of the software architecture of a Car. It embeds four other components. The PlatoonSetup component implements the communication protocol for forming a platoon or for joining an existing platoon. The PlatoonLeader component implements the behavior of a platoon leader, e.g., for notifying followers about a braking maneuver. The PlatoonFollower component implements the functionality of being a follower, e.g., the automatic drive functionality and the communication with the platoon leader. Finally, the DriveController controls the speed of the car based on its current speed measured by a sensor and the reference speed at which the car should drive, it computes the torque which is to be applied by the engine. The DriveController, shown in Figure 3, is a structured

![Fig. 2. MechatronicUML Component for a Car](http://www.mathworks.com/products/simulink)
component as well. It embeds two continuous components NormalDriveController and DistanceDriveController. The NormalDriveController is used by the leader and by cars that do not drive in a platoon. The DistanceDriveController is used by followers in a platoon, because it considers the distance (measured by a distance sensor) to the preceding car while computing the torque. Since the controllers operate in a feedback loop, they automatically adjust the torque such that speed and distance remain constant.

Fig. 3. MechatronicUML Component for the DriveController

A component encapsulates its inner structure and behavior and allows interaction with other components only via its ports. The MechatronicUML component model distinguishes between discrete ports, continuous ports, and hybrid ports. Discrete ports may only be used for software components and execute a stateful, message-based communication protocol. In Figure 2, the port leader of PlatoonLeader is a discrete port that executes the communication protocol for interacting with followers. Continuous ports define the interface of the feedback controllers of the system. They emit or receive a signal value of a specified data type which has the same semantics as a signal in Simulink (cf. Section II). For instance, the DriveController receives the current speed from a sensor by its continuous curSpeed port. Hybrid ports are used at software components for interacting with the continuous components. In particular, they are used for receiving and setting values from/to controllers. refSpeed of PlatoonLeader is a hybrid port that is used for setting a new reference speed to the DriveController. We connect ports of components of the same level of hierarchy by assembly connectors and those of adjacent levels by delegation connectors.

In our component model, we distinguish between component types and component instances. The component types, denoted as components, are instantiated to component instances in a component instance configuration. A component instance configuration defines a concrete software architecture of a system. Components may be instantiated multiple times within a system. All components of the same type share the same behavior specification, but may be in different states at runtime.

Figure 4 shows a component instance configuration of a platoon consisting of two cars where car2 is the leader and car1 is a follower. As described before, the leader executes an instance of PlatoonLeader and uses the NormalDriveController as it can be inferred from the ports of the DriveController instance. The follower executes an instance of PlatoonFollow and uses the DistanceDriveController.

Cardinalities define the minimum and maximum number of instances of a port or an embedded component. For embedded components, the cardinality is specified in square brackets. For ports, we use a cascaded border line if the port may be instantiated more than once, as it is the case for the port leader of a car. Since a platoon leader needs to communicate with many followers, we instantiate one additional instance of the port for each new follower [17]. The resulting port instances represent single-end-to-end connections between the cars using a wireless network. We assume that platoons only use a single lane of traffic without any non-platoon cars mixed in. In our model, the instances of the multi-port leader in a particular configuration are ordered in the same way as the followers in the convoy. In our example, we do not specify a fix upper bound for the number of instances of the leader port enabling platoons of arbitrary length in our MechatronicUML model.

Each component specifies an initial configuration that defines the port instances and, in case of a structured component, the embedded component instances that will be created when the component is instantiated. The initial configuration of all components define the initial component instance configuration for the system. The car component in Figure 2 initially only contains an instance of the PlatoonSetup and an instance of the DriveController. The DriveController, in turn, initially contains one instance of the NormalDriveController.

The component instance configuration of a system can be modified at runtime by performing a reconfiguration. MechatronicUML supports adding and removing component instances, port instances, and connector instances. We specify reconfiguration operations in a rule-based fashion using so-called component story diagrams [20]. Component story diagrams are UML activity diagrams whose activity nodes are formally specified by domain-specific graph transformations [19]. In particular, we embed partial component instance configurations into the activity nodes and annotate the modification as shown in Figure 5.

The component story diagram in Figure 5 specifies the reconfiguration of a car to become a follower in a platoon. The component instance on which the component story diagram is executed is denoted by this. Upon execution, the component story diagram searches for the DriveController in the component instance configuration of the car. Then, it creates an instance of the PlatoonFollow component.
which is indicated by the «create» annotation. In addition, it creates an instance of the follower port for the car including the delegation to the follower port instance of the new PlatoonFollow instance. Further, the component story diagram calls another component story diagram named switchToDistanceCtrl on the DriveController for switching to the DistanceDriveController and connects the DriveController to PlatoonFollow by two assembly connectors.

![Component Story Diagram for Becoming a Follower](image.png)

Fig. 5. Component Story Diagram for Becoming a Follower

For realizing this car platoon example, we need, of course, further rules covering other aspects of self-adaptation. For enabling longer convoys, we need another rule for adding further followers to an existing platoon (cf. [17]). We may add further rules that perform reconfigurations for handling, for instance, changing environmental conditions or hardware failures in a system. We refer to [23] and [24] for more information on these topics.

IV. Transforming Non-Adaptive MechatronicUML Models to Simulink

In our previously published approach for transforming non-adaptive MechatronicUML Models to Simulink [15], we transform one component instance configuration into a Simulink model. In particular, we transform each component instance in the component instance configuration into a subsystem with the same name. In case of a structured component instance, we repeat that transformation recursively for the embedded component instance configuration.

Figure 6 shows the result of transforming the component instance configuration in Figure 4 to Simulink. We obtain two subsystems for the two cars car1 and car2. These subsystems, in turn, contain subsystems for the embedded component instances, which are left out in the figure for better readability.

![Non-Adaptive Subsystem in Simulink](image.png)

Fig. 6. Non-Adaptive Subsystem in Simulink

switch is used for MIL simulation only and is intended to be replaced by the AUTOSAR Virtual Function Bus (VFB) later on. For realizing the message dispatching, each port of the MechatronicUML model receives a uniquely identifying network address in Simulink.

We encode messages by a six-tuple consisting of a package id, the sender network address, the receiver network address, the message, an optional parameter, and a timestamp. If a message enters the communication switch, it is dispatched based on the receiver network address that is contained in the message.

For realizing message-based communication, each port of MechatronicUML is transformed into a port structure consisting of three input ports and one output port in Simulink as shown in Figure 6. Considering the bidirectional port follower of car1 in Figure 4, we obtain the inputs follower_recv, follower_net_addr, and follower_recv_net_addr as well as the output follower_send. follower_recv receives messages from the communication switch, follower_send sends messages to the communication switch. The input follower_net_addr defines the uniquely identifying network address of the port instance follower of car1, while follower_recv_net_addr is the network address of the port instance to which the port instance is connected.

We realize the assembly between follower of car1 and leader1 of car2 by setting the receiver network addresses. The port follower of car1 receives the receiver network address 3 which is the network address of leader1 of car1. By the same rule, leader1 receives the receiver network address 2. The peer ports receive a receiver network address 0 which indicates that they are not connected.

V. Approach for Transforming Reconfigurable MechatronicUML Models to Simulink

MechatronicUML supports the creation and deletion of component instances, port instances, and connector instances during runtime. Since Simulink does not permit that a simulation model changes during a simulation run, the Simulink model needs to contain all possible configurations a priori. Then, we need to switch between those
configurations during a simulation run to emulate the self-adaptive behavior.

Figure 7 provides an overview of the necessary steps for computing and encoding the configurations in Simulink. The input for the procedure are an initial top-level configuration of the system, the component model containing all components, and the component story diagrams defining reconfigurations of the components. The initial top-level configuration specifies which top-level components exist and how many instances of them should be executed. A top-level component is a component that is not embedded in a structured component. In our example, Car is the only top-level component and we define that two instances of Car should be executed in Simulink.

In the first step, we compute all possible configurations of the MechatronicUML model based on the initial top-level configuration (cf. Section VI). Based on the configurations, we determine the required number of component instances and port instances that need to be created in Simulink (cf. Section VII). Then, we encode the configurations as well as the transitions between different configurations in Simulink (cf. Section VIII). In particular, we specify several helper blocks for realizing the reconfiguration. In Step 4, we generate a Simulink model file using the results of Steps 1 to 3. The generation of reconfigurable models extends the generation of non-adaptive models (cf. Section IV). We use a bidirectional model transformation language for creating an intermediate representation of the Simulink model file before generating the actual file using XPand [25]. For a detailed technical description of the generation process, we refer to our technical report [26].

The result of the procedure is a Simulink model that resembles the MechatronicUML model including the reconfigurations. That model can then be integrated with the controller models and the plant model for performing a MIL simulation. The plant model captures the physical aspects of the controlled system and the environment of the system.

VI. Computing Possible Configurations

In the first step, we apply a reachability analysis for computing all possible configurations of the system using our tool presented in [27]. The result of this step is a reachability graph where each node is a component instance configuration and where transitions correspond to the application of a component story diagram. Thereby, we follow the general structure for self-adaptive systems illustrated in [28].

The inputs for the reachability analysis are the initial top-level configuration providing a number of cars, the component model, and the component story diagrams. Initially, each car is instantiating with its initial configuration. This component instance configuration is contained in the initial state config1 of the reachability graph as shown in Figure 8.

Then, Car1 and Car2 may agree on executing the component story diagram startPlatoon() which invokes the component story diagrams for becoming a leader and becoming a follower (cf. Figure 5) on the two cars. The resulting component instance configuration, which has been shown in Figure 4, forms the second state of the reachability graph.

We continue computing the reachability graph by repeatedly applying component story diagrams to the states.
of the reachability graph. The computation terminates if no new configurations can be found.\(^2\) If a configuration that is already contained in the reachability graph is created a second time, the corresponding states are merged. In our example in Figure 8, if car1 and car2 split the platoon, the initial configuration is reached again. In addition, we respect the cardinalities of embedded components (cf. Section III), i.e., we may not call becomeFollower on car1 a second time because a Car component instance may contain at most one instance of PlatoonFollow.

In our illustrative example, we obtain three configurations for each of the two cars: one for being the platoon leader, one for being a follower, and one for driving alone.

VII. DETERMINE REQUIRED NUMBER OF INSTANCES

In our component model, we specify lower and upper bounds for all ports of a component and for all embedded components of a structured component. Naively, we could just generate the maximum number of subsystems for each embedded component and the maximum number of ports. However, after computing the reachability graph in Step 1, we may check whether we actually need that number of embedded component instances and port instances by counting how many have actually been used in the computed configurations.

In our example, we did not specify an upper bound for the number of instances of the leader port of a Car (cf. Section III). However, since we restrict ourselves to two cars in the initial top-level configuration, we can automatically set the upper bound for the multi-port to 1 for this particular export.

VIII. ENCODING CONFIGURATIONS IN SIMULINK

We realize reconfiguration in Simulink by implementing a set of helper blocks that encode the configurations that we computed in Step 1 (cf. Section VI). The helper blocks can change the currently instantiated component instances, port instances, and connector instances. In addition, they define an interface for setting a new configuration for a subsystem. In the following subsections, we illustrate how we create and destroy component instances (Section VIII-A), port instances including assemblies (Section VIII-B), and delegation instances (Section VIII-C) in Simulink.

A. Component Instances

We transform component instances of MechatronicUML to enabled subsystems with the same name in Simulink. An enabled subsystem has a control input which controls whether the subsystem will be executed or not. If a component instance is not instantiated in the current component instance configuration, we set a 0 to the control input of the corresponding subsystem. Otherwise, we set a 1 to the control input to enable the corresponding subsystem.

\(^2\)Please note that our reachability analysis is depth-limited to achieve termination even if the configuration space is unbounded. Then, the developer needs to revisit the MechatronicUML model.

Figure 9 shows a simplified subsystem that was generated for the Car component of Figure 2. We obtain one enabled subsystem for each embedded component and one additional BlockControl block. The BlockControl block is responsible for setting the control inputs of the enabled subsystems and, thus, for controlling which subsystem becomes enabled based on the current component instance configuration. If a an embedded component instance is instantiated in all configurations, we transform it into a normal subsystem as, e.g., the subsystem ctrl. Listing 1 shows the contents of the BlockControl of Car1.

Listing 1. Contents of BlockControl of Car1

```matlab
switch config
    case 1
        platoonSetup = 1
        leader = 0
        follower = 0
    case 2
        platoonSetup = 1
        leader = 0
        follower = 1
    case ...
end
```

The BlockControl is implemented as an Embedded MATLAB Function using switch/case-statements to distinguish the configurations. If Car1 is in configuration 1 (cf. Figure 8), only the embedded component instances platoonSetup and ctrl are instantiated. Consequently, the BlockControl outputs 1 for platoonSetup, which enables the enabled subsystem, and 0 for follower and leader. ctrl is always enabled because it is a normal subsystem. If Car1 switches to configuration 1, the follower subsystem is enabled for instantiating the follower component instance.

B. Port Instances and Assemblies

For the transformation of port instances and assemblies, we need to distinguish between discrete message-based ports and continuous signal-based ports.

In Simulink, we emulate instantiating and deleting assemblies and their adjacent port instances by setting new receiver network addresses to the ports. As an example, consider the assembly between the leader port instance of car2 and the follower port instance of car1 in Figure 4. Initially, the two cars will not be in platoon mode and, therefore, the assembly is not present. We realize this situation by setting 0 as the receiver network address for both ports in Simulink. We create the assembly by giving the leader1 port instance of car2 the receiver network address 2 (which is the network address of follower of car1).
In addition, we set the receiver network address of follower of car1 to 3. Then, messages sent by follower of car1 will be dispatched to leader1 of car2 by the communication switch. For destroying the assembly, we set 0 as the receiver network address, again, which causes both ports to be silent.

We use the PortControl block shown in Figure 10 for setting the receiver network addresses of all discrete port instances based on the current configuration. Listing 2 shows the contents of the PortControl of the top-level subsystem.

![PortControl Block for Creating/Destroying Assemblies between Discrete Ports](image)

Fig. 10. PortControl Block for Creating/Destroying Assemblies between Discrete Ports

Listing 2. Contents of PortControl of the Top-Level Configuration

```
switch config
  case 1
    car1_follower = 0
    car2_leader1 = 0
  case 2
    car1_follower = 3
    car2_leader1 = 2
  case ...
end
```

In configuration 1, the two cars are not in platoon mode and, therefore, they are not connected. As a consequence, PortControl sets the receiver network address for both port instances to 0 which disables them. In configuration 2, the port instance follower of Car1 is connected to the port instance leader1 of Car2. Therefore, PortControl sets the receiver network address of follower of Car1 to 3 and the receiver network address of leader1 of Car2 to 2.

For continuous and hybrid ports, we realize reconfiguration differently, because these ports directly specify Simulink signal values. In our example in Figure 4, the refSpeed for the DriveController is either provided by the component instance ld1 if the car is the platoon leader or by the component instance fo1 if the car is a follower. Thus, on the type level (cf. Figure 2), the port refSpeed of DriveController has two incoming assemblies where at most one of them may be instantiated during runtime.

![Multiport Switch](image)

Fig. 11. PortControl Block for Creating/Destroying Assemblies between Continuous and Hybrid Ports

For transforming delegations, we need to distinguish between discrete ports and continuous ports, again. At runtime, we can reconfigure delegations such that a port instance of a structured component instance is delegated to a different port instance of an embedded component instance.

For discrete ports, we require that they receive uniquely identifying network addresses (cf. Section IV). When using the address of port leader1 of the embedded subsystem leader in Car2 for the communication between the two cars, changing a delegation becomes visible to Car1. That violates the desired encapsulation of components.

Therefore, we use a hierarchical addressing scheme based on network address translation (NAT, [29]) where network addresses are uniquely identifying only within one subsystem. Then, port instances of structured component instances serve as proxies and translate between the network address visible to the outside world and the network address of the delegated port.

In our example in Figure 10, the port leader1 of car1 is delegated to the embedded component leader (cf. Figure 2). If car2 sends a message to the port leader1 of car1, it is opaque for car2 whether this port is delegated or not and to which embedded component it is delegated. Consequently, each port instance of a structured component receives a network address which is used by the communication partner. That address is then translated by so-called address translation blocks to the internal address of the delegated port, as shown in Figure 12.

![Address Translation](image)

Fig. 12 shows how the port leader1 of car1 is delegated to the port leader1 of the embedded subsystem leader. If a message is sent to leader1 of car1, it first enters the addressTranslationIN subsystem which knows the network address of the port leader1 of car1. The address translation then replaces the receiver network address in the message by the address of the receiving port and
For realizing reconfiguration of such delegation, we implemented the DelegationSwitch subsystem as shown in Figure 13.

Fig. 13. DelegationSwitch Block for Creating/Destroying Delegations for Continuous and Hybrid In-Ports.

The subsystem has two inputs and one output for each delegation that is specified in the MECHATRONICUML component. In this case, we have two delegations, one for each controller. Based on the control input, the value received via refSpeed of DriveController is either propagated to normal_ctrl (control == 1) or to distance_ctrl (control == 2) using an Embedded MATLAB function. By applying a 0 to the control input, the delegation is destroyed.

The out-port torque of DriveController is always provided by one of the two controllers depending on the current configuration. In this case, changing the delegation always results from exchanging the controller. If a controller is exchanged instantaneously by another controller, a jump may occur at the output torque of DriveController which may cause damage to the physical system. In previous publications, fading functions [30] and flat switching functions [31] have been introduced for solving that problem. Then, the output of the destroyed controller is faded to the output of the created controller in a controlled fashion that guarantees stability of the system.

Fig. 14. Fading Block for Creating/Destroying Delegations for Continuous and Hybrid Out-Ports.

Figure 14 shows the resulting Simulink block structure. The torque output ports of normal_ctrl and distance_ctrl are both connected to the Fading Function. The control input of the Fading Function determines which value has to be used. If the value of control changes, a fading is performed. The specification of a fading function is subject to control engineers and depends on the concrete controllers to be exchanged. Due to space limitations, we will not provide details of the fading functions in this paper and refer to the given literature [30], [31].
IX. Related Work

Related work stems from three areas of research: specifying reconfiguration in Simulink models, adding reconfiguration capabilities to AUTOSAR-based systems, and specifying reconfiguration in other tools for MIL simulation, namely Dymola/Modelica. We discuss related approaches in this order.

Cancare [8] and Paiz et al. [7] describe approaches for simulating reconfigurable FPGA-boards in Simulink. They only switch between implementation variants of the same block using switches, but provide no means for message-based communication and adding/removing components from the simulation. Schulze et al. [32] provide a concept for product line support in Simulink where a concrete variant is configured via control signals. This is comparable to our block control blocks, but their approach does not permit changing the configuration at run-time. The Quanser Real-Time Control Software (QUARC, [33]) provides special blocks for switching between two Simulink models during run-time. They stop the simulation, transfer variables, and restart the simulation on the target model. In contrast to our approach, this approach does not permit to simulate the transient phase where the reconfiguration is executed. In addition, using self-defined blocks hinders the use of standard code generators.

Becker et al. [9] define an extension for AUTOSAR to support architectural reconfiguration which makes it probably the closest to our approach. In their approach, a developer needs to specify all configurations of system manually including an automaton defining switches between the configurations. Then, they generate an AUTOSAR system containing all variants including code for a so-called StateManager and a RoutingComponent. The StateManager controls the current configuration, while the RoutingComponent redirects signals. In contrast to their approach, we use a rule-based approach for specifying reconfiguration, which is more efficient if the system configurations are large showing only little differences. In addition, their approach does not allow for early validation using MIL simulations. Berger and Tichy [10] extend the AUTOSAR watchdogs towards transactional reconfigurations with rollback support, but they do not consider simulation of the system. Zeller et al. [34], [11] and Klobedanz et al. [35] provide reconfiguration of networked embedded systems by reallocating of software components to new ECUs at run-time. Their approaches can be used for technically realizing reconfiguration, but not for MIL-simulations in Simulink. Trumler et al. [36] and Feng et al. [37] propose middlewares for automotive systems supporting runtime reconfiguration by migrating tasks (Trumler et al.) or switching between different component implementations (Feng et al.). Their middlewares are supposed to replace the AUTOSAR RunTime Environment (RTE), but do not support MIL simulation.

Modelica [12] with the commercial simulator Dymola\(^3\) is a competitor of MATLAB/Simulink for the specification and simulation of technical systems. While Modelica does not support runtime reconfiguration, there exist two extensions of Modelica, Mosilab [13] and Sol [14], that support reconfiguration. However, both approaches rely on own simulators because their extensions are not supported by Dymola. That, in turn, hinders using standard code generator for generating production code for the system.

X. Conclusions

In this paper, we presented an approach for realizing MIL simulations of self-adaptive systems in MATLAB/Simulink extending our approach of [15]. Based on a model of the system in our language MECHATRONIC UML, we automatically derive all possible system configurations based on the specification of the reconfiguration behavior. We generate an encoding of these configurations by implementing several helper blocks in Simulink. In particular, we use enabled subsystems for creating/deleting component instances and a communication switch implementation for creating/deleting ports and assembly connectors. Finally, we support creating/deleting delegation connectors by address translation blocks. By using only native blocks of Simulink and Embedded MATLAB Functions, we enable using standard off-the-shelf code generators for generating production code for the target system.

In contrast to MATLAB/Simulink, MECHATRONIC UML enables developers of self-adaptive systems to natively specify the communication and reconfiguration behavior. That significantly reduces the complexity of the reconfiguration model and, therefore, helps improving the quality of the software. We used this approach successfully for creating models for the RailCab system\(^4\), a modern railway transportation system with autonomous trains having complex self-adaptive behavior.

In future work, we plan to integrate our approach with the transactional reconfiguration approach for MECHATRONIC UML presented in [23]. This approach guarantees that all reconfigurations are executed safely and only produce a consistent component instance configuration. In addition, it ensures that conflicting adaptation requests are properly serialized. Further, we plan to investigate whether our approach can be combined with the approach by Becker et al. [9] such that the generated Simulink model may be used as an implementation of their reconfigurable AUTOSAR system. Finally, we want to further evaluate our approach in an industrial case study.

Acknowledgment

This work was partially developed in the project “Simulationgestützter Entwurf für Elektrofahrzeuge” (Simulation-based Design of Electrical Vehicles) which is funded by the European Union and the state of Northrhine-Westphalia (grant No. 64.65.69-EM-1008A). We thank Andrey Pines for laying ground to this paper as part of his Bachelor’s Thesis. We thank Matthias Becker and Matthias Meyer for useful comments.

---

\(^3\)http://www.3ds.com/de/products/catia/portfolio/dymola

\(^4\)http://www.railcab.de
REFERENCES


