Modeling and Verifying Dynamic Communication Structures based on Graph Transformations*

Stefan Henkler, Martin Hirsch, Claudia Priesterjahn, Wilhelm Schäfer

Abstract: Current and especially future software systems increasingly exhibit so-called self* properties (e. g. , self healing or self optimization). In essence, this means that software in such systems needs to be reconfigurable at runtime to remedy a detected failure or to adjust to a changing environment. Reconfiguration includes adding or deleting software components as well as adding or deleting component interaction. As a consequence, the state space of self* systems becomes so complex, that current verification approaches like model checking or theorem proving usually do not scale. Our approach addresses this problem by first defining a so-called "regular" system architecture with clearly defined interfaces and predefined patterns of communication such that dependencies between concurrently running component interactions are minimized with respect to the system under construction. The construction of such architectures and especially its reconfiguration is controlled by using graph transformation rules which define all possible reconfigurations. It is formally proven that such a rule set cannot produce any "non-regular" architecture. Then, the verification of safety and liveness properties has to be carried out for only an initially and precisely defined set of so-called coordination patterns rather than on the whole system.

1 Introduction

Current and especially future software systems increasingly exhibit so-called self* properties (e. g. , self healing or self optimization). In essence, this means that software in such systems needs to be reconfigurable at runtime to remedy a detected failure or to adjust to a changing environment. Reconfiguration includes adding or deleting software components as well as adding or deleting component interaction. This increases the complexity of the software significantly because the reconfiguration process also has to be controlled by software.

As many of these systems are used in a safety critical environment, high quality software is absolutely necessary; in particular, it must satisfy safety and liveness constraints. A common approach (at least in research and advanced industrial projects) to address these challenges is to verify as many constraints as possible at the model-level, i. e. , the software model is formally analyzed and code is automatically generated from the model.

*This work was developed in the course of the Special Research Initiative 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

However, the state space of self* systems becomes so complex, that current verification approaches like model checking or theorem proving usually do not scale [Alu08]. Even when using a compositional approach to verification and applying symbolic techniques, it is hardly possible to track all possible reachable system states because the possible dependencies between different, usually highly concurrent component interactions in a complex network of components are by far too many.

On a general level, our approach addresses this problem by first defining a regular system architecture with clearly defined interfaces and predefined patterns of communication such that dependencies between concurrently running component interactions are minimized with respect to the system under construction. The construction of such architectures and especially its reconfiguration is controlled by exploiting a formal definition of the construction process and by proving based on that formal definition that an architecture includes in fact only dependencies which are absolutely necessary. In most cases, this means that reconfiguration can only add components and corresponding component interactions, if that does not interfere with already existing interactions which have been proven to be correct already. Then, the verification of safety and liveness properties has to be carried out for only an initially and precisely defined set of so-called coordination patterns rather than on the whole system.

We clearly distinguish between the internal behavior of a component and the interaction behavior between components that is based on message passing. In this paper, we focus on modeling and analyzing the communication structure among different components, i. e. , the specification of communication protocols.

Self*-systems are usually built as a network of components, possibly even mobile, and exhibit a high degree of component interaction. Often, this interaction must satisfy strict real-time constraints while of course still guaranteeing the mentioned safety properties (e. g. , in the case of network failures). Because of this characteristic, the protocols built are often complex. An approach to model and verify such protocols is described in [GTB$^+$03]. However, it is restricted to the specification of bilateral communication based on (extended) timed automata and does not consider reconfiguration at runtime.

In this paper, we extend our so-called MECHATRONIC UML approach. (The name stems from the fact that the approach has originally been developed in the domain of mechatronic systems but it is, of course, not restricted to this class of applications. In fact, a significant part of the software of advanced mechatronic systems deals with component interaction and adjusting their behavior concerning e.g. self-healing or self-optimization. In that sense, mechatronic systems are just a special case of self*-systems [SW07].) We add graph transformation rules to the original approach, i.e. a generating system, to formally define the construction of component architectures, as mentioned above. This approach also supports the formal analysis of certain properties of the resulting (reconfigurable) architectures which in turn minimizes the verification effort to check safety and liveness properties of the communication protocols.

The key contribution of this paper is thus a particular modeling and verification approach based on a combination of graph transformation rules and timed automata. It addresses the challenge of scalability of verification even in cases where a system does not only exhibit an arbitraryly large finite number of states but an infinite number of states.

A concrete example for a complex self*-system with the need to coordinate a varying number of components is the RailCab project[1]. The vision of the RailCab project is a mechatronic rail system where autonomous vehicles called shuttles apply the linear drive technology, as used by the Transrapid system, but travel on the existing passive track system of a standard railway system. This system is currently under construction and a first version of the controlling software of the physically existing system has been built using the approach of [HTBS08].

One particular problem (previously presented in [GTB$^+$03]) is the convoy coordination of certain system components, e.g. the shuttles. Shuttles drive in a convoy in order to reduce energy consumption caused by air resistance and to achieve a higher system throughput. Such convoys are established on-demand and require small distances between the shuttles. These required small distances cause the real-time coordination between the speed control units of the shuttles to be safety critical which results in a number of constraints, that have to be addressed when building the shuttles' control software. In addition a complex coordination is required when the convoy consists of more than two shuttles. Since shuttles can join or leave a convoy during runtime a flexible structure for the specification of the coordination is needed.

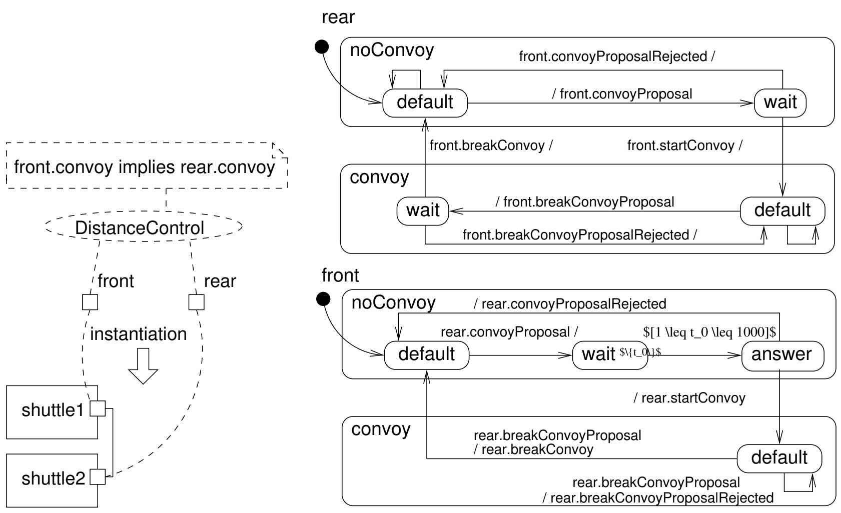
The structure of the paper is as follows: We first review the original MECHATRONIC UML approach in Section 2. Section 3 covers the concepts for modeling parameterized coordination patterns with a flexible number of participants and multi-ports. Then, we describe how compositional verification of our approach is achieved in Section 4. Finally, we will discuss related work in Section 5. The last section summarizes the paper and gives an outlook on future work.

2 Foundations

In our approach, the architecture is given by components, their ports and the connections between ports (which are just identified by links). This model is formally described by an adaptation of the UML 2.0 component model. Among other things, the adaptation especially covers the definition of restrictions on how ports have to be connected such that communication via different ports of the same component is guaranteed to be side-effect-free.

Communication between autonomous components has been defined by so-called *coordination patterns* [GTB$^+$03]. A coordination pattern, as depicted in Figure 1(a), describes the communication between two components and consists of multiple communication partners, called *roles*. Roles are linked by a connector. The communication behavior

[1]http://www-nbp.upb.de



(a) Structure (b) Role behavior

Figure 1: Real-Time Coordination Pattern for a Shuttle Convoy

of a role is specified by a real-time statechart.

Real-time statecharts are an extension of UML state machines which support more powerful concepts for the specification of real-time behavior. They are semantically based on the timed automata formalism such that a formal analysis is possible using the model checker UPPAAL[2].

The behavior of the connector is described by another real-time statechart that, in addition to the transport of the messages, models the possible delay and the reliability of the channel, which are of crucial importance for many systems.

Safety constraints which have to hold (and are model checked) for these patterns are either a so-called pattern constraint or a role invariant which concerns a property of a single role only. A role invariant specifies a property that has to be satisfied by the communication partner. A pattern constraint specifies a property that has to be satisfied by all communication partners and connectors. Both types are defined in TCTL[3].

The role behavior is refined by ports that build the interfaces of our components, i.e. the ports implement the external behavior as specified by the role behavior. The refinement has to respect the role behavior (do not add possible behavior or block guaranteed behavior) and additionally has to respect the guaranteed behavior of the roles given by its invariants [GTB$^+$03].

An additional statechart for synchronization is used to describe required dependencies between role behavior (as the component behavior is not necessarily only a parallel composition of the different role behavior). This allows for the strict separation of communication

[2]www.uppaal.com
[3]TCTL: Timed Computation Tree Logic

behavior and internal component behavior. We have described the specification and analysis of the synchronization behavior for bilateral communication in [GTB+03]. We have presented an automatic synthesis of the synchronization behavior in [HGH+09].

In our application example, the coordination between two shuttles is modeled by the DistanceControl pattern. It consists of two roles, the front role and the rear role and one connector that models the link between the two shuttles. The pattern specifies the protocols to coordinate two successive shuttles.

Initially, both roles are in state noConvoy::default, which means that they are not in a convoy. The rear role decides to propose building a convoy or not. After the decision to propose a convoy, a message is sent to the other shuttle resp. its front role. The front role decides to reject or to accept the proposal after max. 1000 msec. In the first case, both statecharts revert to the noConvoy::default state. In the second case, both roles switch to the convoy::default state.

Eventually, the rear shuttle decides to propose a break of the convoy and sends this proposal to the front shuttle. The front shuttle decides to reject or accept that proposal. In the first case, both shuttles remain in convoy-mode. In the second case, the front shuttle replies by an approval message, and both roles switch into their respective noConvoy::default states.

A safety requirement of the pattern is that no collision happens. The pattern constraint enforces the shuttle role to be in state Convoy while the coordinator role is also in state Convoy (shuttle.convoy implies coordinator.convoy).

3 Modeling Parameterized Coordination Patterns

So far, our approach allows constructing and verifying system structures with fixed bilateral communication only, i.e. reconfiguration is not supported. As self*-systems can be composed of thousands of components which might be replaced at runtime, it is not possible to specify and verify the system as a whole.

We use graph transformation systems to define the construction and reconfiguration of architectures consisting of components and corresponding component interactions. A so-called start graph defines an initial configuration and a set of rules defines all allowed configurations of components and component interactions (without specifying any behavioral part). Based on the start graph, one can formally prove that such a generating system does not produce any so-called forbidden graph structure. Such a graph structure would include a sub graph (corresponding to an architecture as a result of a reconfiguration operation) which incorporates component interactions where no predefined corresponding coordination pattern exists.

This means that the behavior of component interactions is again specified by real-time statecharts as explained in the previous section. However, in order to cover possible multilateral component interactions, they have to be extended by introducing parameters in the definition of statecharts as it will be explained in section 3.2.

3.1 System Structure Specification

The reconfiguration of an embedded or mechatronic system architecture defined by graph transformation rules, is often time critical. Especially the creation or deletion of a port object or component might involve some complex operations. Verification of time constraints of reconfiguration operations should consequently be verifiable on the model level as well. We enhance our specification approach by adding clock instances and invariants over these clocks to a graph transformation rule to specify the time which is needed for the corresponding reconfiguration operation.

Our formal definition of Timed Graph Transformation Systems [Hir08] enables the formal verification, i. e. reachability analysis to prove the correctness of our system structure concerning time constraints across several reconfiguration operations as well.

Figure 2(a) shows a graph transformation rule for joining an existing convoy. A graph transformation rule consists of a left hand side and a right hand side. The left hand side gives the system configuration in which the rule can be applied, the right hand side shows the result of the application. Here, left and right hand side are depicted in one single graph using the notation $++$ or $--$ for edges (and nodes) which are created or deleted during rule application. Since the reconfiguration happens within a certain time interval, we add clocks to our graph transformation rules and invariants over these clocks.

In our example rule in Figure 2(a) we add the clock t. The invariant $t < 5$ specifies that the creation of a link between two shuttles takes maximum 5 time units. By the attribute $<< last >>$ we specify that a new shuttle will join the convoy at the last position only.

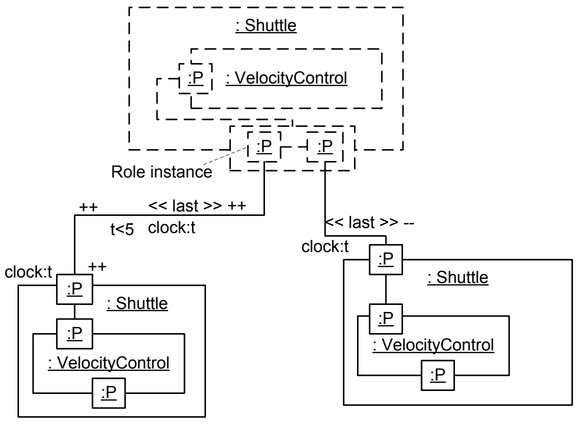
As graph transformation systems have a formal basis and tool support for model checking, a formal proof of properties is possible. In our case, properties which must not hold, would mean the generation of communication structures with no corresponding coordination pattern. We define such ”non-regular” structures by so-called forbidden graphs.

Figure 2(b) depicts such a forbidden graph structure. It basically means that shuttles in a convoy or rather their member roles are not allowed to communicate with anything else than the next role instance in an ordering of role instances ($_i$, $_{i-1}$, ... reflects the ordering of the role instances).

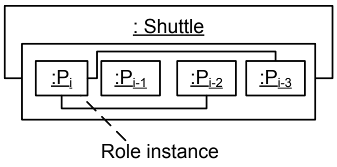
3.2 Specification of Component Communication

To model the communication of more than two components we extend our coordination patterns to parameterized coordination patterns by introducing multi-roles. A multi-role consists of one or more sub-roles. The behavior of a multi-role is specified by a parameterized real-time statechart which defines the behavior of all sub-roles. Parameterized real-time statecharts are semantically based on a parameterized timed automaton, a timed automaton extended by parameterized signals.

Each sub-role of a multi-role is initialized by a coordination statechart (e. g. cf. Figure 5). A sub-role is created and initialized by a specific call, e. g. createPort, of a timed graph



(a) Extension Rule



(b) Forbidden Graph Structure

Figure 2: Extension Rule and Forbidden Graph Structure

transformation rule. The formal specification of the sub-roles of a multi-role can be found in [Hir08].

If an additional synchronization statechart is required to describe dependencies between role behavior (see Section 2), we can apply our approach for bilateral communication [HHG08]. Here, we exploit the fact that the coordination statechart of a multi role encapsulates the set of role instances, while the internal synchronization uses the uniform interface.

In the following, we extend our example by the ConvoyCoordination pattern (see Figure 3), which enables the coordinator (role) to control the convoy (member role).

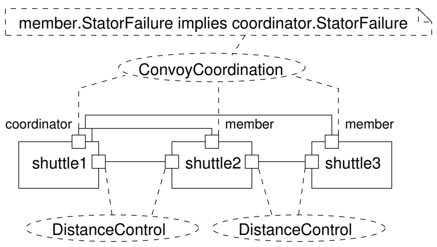


Figure 3: Extended convoy example

The parameterized role coordinator is depicted in Figure 4. Initially, the coordinator role is in state WaitForTrigger. If it receives a next$_k$ trigger, the role switches to state Idle. Initially, next$_k$ is sent by the adaption statechart as depicted in Figure 5 while in all following operations next$_k$ is sent by the ancestor role with index $k-1$. Now, the communication with the shuttle role is started. First, an update message including current profile data, the reference velocity as well as the reference position of the shuttle, is sent. Thereafter, the role waits in state SentAcknowledge for the confirmation acknowledge by the shuttle role. If this message is received the role switches to state WaitForTrigger and sends the next_{k+1} signal to activate the next sub-role. If the acknowledge message is not received, the role switches to state NextFailed and sends a signal to the internal component behavior in order to initiate appropriate routines. Further, the role switches to state StatorFailure, if a publishStatorFailure is received when being in state Idle.

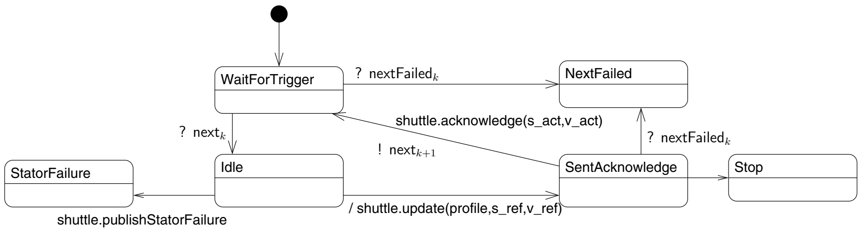


Figure 4: The behavior of a coordination role role_k

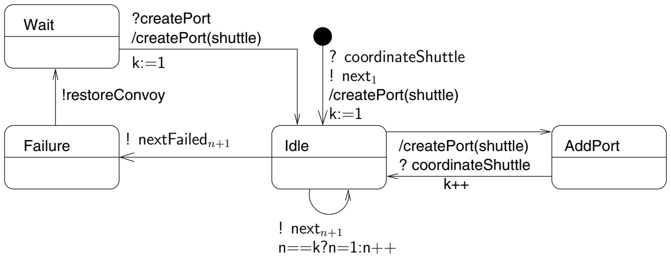


Figure 5: The coordination statechart for all coordinator roles

The real-time statechart of the member role consists of three states (cf. Figure 6). Initially, the role is in state Normal. Every 150 time units the role has to receive an update message from the coordinator role. This message includes the current profile, reference position and reference velocity. The role confirms the receipt of the message with an acknowledge message including the current position and current velocity. In case no update message is received (e.g. network failure) the role switches to the network failure state after 150 time units. The state StatorFailure will be reached if another shuttle propagates a StatorFailure.

The pattern constraint enforces the member role to be in state StatorFailure while the coordinator role is also in state StatorFailure (shuttle.StatorFailure implies coordinator.StatorFailure).

4 Verification

Figure 3 shows the communication structure of one coordinator and two shuttles. If we had created an arbitrary communication structure (by reconfiguration operations), we would have to verify the constraints for the whole state space which is computed by taking the concrete maximal instance number into account (which could be a very large number;

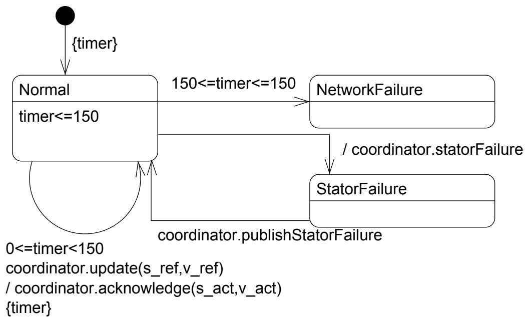

{timer}
Normal
timer<=150
150<=timer<=150
NetworkFailure
/ coordinator.statorFailure
StatorFailure
coordinator.publishStatorFailure
0<=timer<150
coordinator.update(s_ref,v_ref)
/ coordinator.acknowledge(s_act,v_act)
{timer}

Figure 6: The behavior of a shuttle role member

maybe even infinite). By taking advantage of our regular system structure we only need to verify the constraints for an arbitrarily chosen coordination pattern and an arbitrarily selected pair of sub-roles.

The key idea to tackle these problems and to define a scalable verification approach is to exploit the underlying modeling formalism (cf. Section 3) and still use the compositional reasoning approach for bilateral communication [GTB$^+$03].

The system achitecture is built such that component interactions do not interfere. This is achieved by the definition of forbidden graph structures as shown in Figure 2(b). Consequently, it is enough to verify an arbitrarily chosen pair of sub-roles concerning safety and liveness constraints[4]. Each sub-role corresponds to one port of a communication channel. If such a regular structure is correctly defined by a graph transformation system, it can be formally proven by induction over the number of sub-roles that the behavior of other sub-roles does not interfere with arbitrarily chosen pairs of sub-roles.

The induction is based on reachability analysis by the tool GROOVE [Ren04]. We have extended GROOVE by time to examine our timed graph transformation rules. In detail, all time constraints specified in the rules are considered and it is automatically checked if a specified invalid communication structure can be created (verification step 2).

To check the safety constraint shuttle.convoy implies coordinator.convoy as introduced in Section 2 for a pair of parametrized sub-roles we use the model checker UPPAAL[5] which we have successfully used in all our previous modeling and verification approaches, e. g. [BGHS04] (verification step 3). For completeness reasons we also have to check local invariant properties to ensure correct behavior of a single sub-role (verification step 1).

In the following we give a sketch of the necessary verification steps based on the definitions introduced in Section 3 to check if a parameterized coordination pattern is correct w. r. t. to the specified constraints, e. g. shuttle.convoy implies coordinator.convoy.

[4]This is correct for local constraints like safety constraints. For some global constraints the complete behavior of the system still has to be checked. For these constraints we refer to simulation. However, we did not find constraints of this type in our applications.

[5]www.uppaal.com

Verification steps:

1. For the parameterized coordination pattern verify that an arbitrary sub-role fulfills the local constraints of the multi-role, i.e. the roles are compatible, and there exists no deadlock.
2. Verify that the structural constraints of the timed graph transformation rules are satisfied, i.e. apply reachability analysis for timed graph transformation systems to verify that
   a) the timing constraints of the timed graph transformation rules are satisfied (see Figure 2(a)) and
   b) the situations specified by the forbidden graph structure do not occur (see Figure 2(b)).
3. Verify that pairs of subsequent sub-roles do not contain a deadlock.

5 Related Work

The state explosion problem limits the applicability of model checking for complex software systems. A number of modular and compositional verification approaches have therefore been proposed. One particular compositional approach is the assume/guarantee paradigm [MC81]. Model checking techniques that permit compositional verification following the assume/guarantee paradigm have been developed [CGP00, p. 185ff]. Our approach also employs the assume/guarantee paradigm, but supports dynamic structures and takes advantage of information available in form of communication structures and pattern role protocols to derive the required additional assumed and guaranteed properties automatically rather than manually as in [CGP00]. None of the current approaches for the specification and verification of dynamic systems considers a well-defined combination of the specification of a dynamic structure and its behavior to facilitate a compositional verification approach [BCDW04].

6 Conclusion and Future Work

Model-driven development of self*, mobile systems should support a formal verification approach, as many of these systems are used in a safety-critical environment. To a large extent, the behavior of such systems is given by the communication between the system components. We extend the existing model-driven MECHATRONIC UML approach to deal especially with the possibly infinite state space of self*, mobile systems. The infinite state space makes it basically impossible to apply existing "‘classical"’ approaches to system verification. The extensions of MECHATRONIC UML include the definition and verification of regular communication structures based on timed graph transformation systems. As a consequence, only predefined communication protocols (coordination patterns) specifying bilateral as well as multilateral communication, need to be individually verified which is much less time consuming (and in fact possible for real applications) than checking a complete system specification. Component internal behavior in turn can be verified independently from the communication behavior but this is not the subject of this paper. Basically, the approach suggests a particular system decomposition enabling compositional verification.

Admittedly there is still one significant limit of the approach. Broadcasting based communication architectures where communication between components basically does not follow any rules (and does not exhibit corresponding hierarchies), cannot be covered by our approach. However, we found many examples of systems where this type of generality does not exist and thus our approach is applicable to a wide range of system like e. g. public transport, production or telecommunication systems. In fact, one could even argue that building "really" safe systems requires to avoid broadcasting like communication structures. Even the automobile industrie (where production costs are a major issue) works on approaches which refrain from broadcasting based communication architectures [NS08]. A first step to apply a structured modeling and analysis approach is realized by integrating the FlexRay[6] communication system. FlexRay requires a structured specification of the communication properties to enable deterministic communication.

A first evaluation of our approach was given in [HTBS08]. It describes the software development of the RailCab project (cf. section 1). As said, RailCab is not just a lab experiment. A real running prototype has been physically built on campus in the the scale of 1:2.5. The complete communication software of the shuttles has been implemented using the approach as presented in this paper.

We also work on a project which integrates the approach into an existing commercial tool called CAMeL-View. This tool is used in control engineering to specify single controllers. State-based communication between controllers is supposed to be modelled and verified by the MECHATRONIC UML approach [GBSO04].

Acknowledgements We thank Tobias Eckardt and Markus von Detten for their comments and proof-reading of this paper.

References

[Alu08] Rajeev Alur. Model Checking: From Tools to Theory. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 89–106. Springer Berlin / Heidelberg, 2008.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM.

[6]http://www.flexray.com/

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In *Proc. of the International Workshop on Specification and Validation of UML Models for Real-Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, pages 1–20, October 2004.

[CGP00] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, January 2000.

[GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*, pages 179–188. ACM Press, November 2004.

[GTB$^+$03] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[HGH$^+$09] Stefan Henkler, Joel Greenyer, Martin Hirsch, Wilhelm Schäfer, Kathan Alhawash, Tobias Eckardt, Christian Heinzemann, Renate Löffler, Andreas Seibel, and Holger Giese. Synthesis of Timed Behavior From Scenarios in the Fujaba Real-Time Tool Suite. In *Proc. of the 31th International Conference on Software Engineering (ICSE), Vancouver, Canada*, May 2009.

[HHG08] Martin Hirsch, Stefan Henkler, and Holger Giese. Modeling Collaborations with Dynamic Structural Adaptation in Mechatronic UML. In *Proc. of the ICSE 2008 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08),Leipzig, Germany*, pages 33–40. ACM Press, May 2008.

[Hir08] Martin Hirsch. *Modell-basierte Verifikation von vernetzten mechatronischen Systemen*. PhD thesis, University of Paderborn, Paderborn, Germany, September 2008.

[HTBS08] Christian Henke, Matthias Tichy, Joachim Böcker, and Wilhelm Schäfer. Organization and Control of Autonomous Railway Convoys. In *Proceedings of the 9th International Symposium on Advanced Vehicle Control, Kobe, Japan*, October 2008.

[MC81] J. Misra and M. Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[NS08] Oliver Niggemann and Joachim Stroop. Models for model's sake: why explicit system models are also an end to themselves. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 561–570. ACM, 2008.

[Ren04] Arend Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer Verlag, 2004.

[SW07] Wilhelm Schäfer and Heike Wehrheim. The Challenges of Building Advanced Mechatronic Systems. In *FOSE '07: 2007 Future of Software Engineering*, pages 72–84, Washington, DC, USA, 2007. IEEE Computer Society.