A Tool Suite for the Model-Driven Software Engineering of Cyber-Physical Systems

Stefan Dziwok, Christopher Gerking, Steffen Becker, Sebastian Thiele
Software Engineering Group,
Heinz Nixdorf Institute, University of Paderborn,
Zukunftsmeile 1, 33102 Paderborn, Germany
stefan.dziwok@upb.de

Christian Heinzemann, Uwe Pohlmann
Fraunhofer IPT, Project Group Mechatronic Systems Design, Software Engineering,
Zukunftsmeile 1, 33102 Paderborn, Germany
uwe.pohlmann@ipt.fraunhofer.de

ABSTRACT

Cyber-physical systems, e.g., autonomous cars or trains, interact with their physical environment. As a consequence, they commonly have to coordinate with other systems via complex message communication while realizing safety-critical and real-time tasks. As a result, those systems should be correct by construction. Software architects can achieve this by using the MECHATRONICUML process and language. This paper presents the MECHATRONICUML TOOL SUITE that offers unique features to support the MECHATRONICUML modeling and analyses tasks.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*

General Terms

Design, Languages, Verification

Keywords

Cyber-physical systems, mechatronic systems, DSL, timed model checking, real-time coordination, system simulation

1. INTRODUCTION

Mechatronic systems or Cyber-physical systems (CPS) [7] are systems that interact with their physical environment. Examples are autonomous cars, trains, rescue robots, etc. Due to their nature, different engineering disciplines are involved in their development like mechanical, control, and software engineering. The development of such systems is especially difficult as they have to obey real-time properties, are safety-critical, and need to take their physical environment and laws into account. They coordinate via software by sending and receiving asynchronous messages.

As a consequence of this list of properties, the software development for such systems needs new languages and supporting tools. In our previous work, we have developed the MECHATRONICUML method [1, 9], which defines a modeling language and a corresponding development process [13] focusing on CPS's software development. A tool that supports MECHATRONICUML has to face several challenges. First, it needs to seamlessly integrate itself into the MECHATRONICUML process. As MECHATRONICUML is a visual language, dedicated high-quality graphical editors supporting different views to reduce the visual complexity are needed. As CPSs operate in real-time and safety-critical environments, testing such systems is often not an option because rare failures might remain undetected. Hence, the tool has to ensure model correctness by construction, e.g., by supporting scalable formal analyses like model checking or virtual prototyping in simulations. For such analyses, the tool needs to interoperate with established tools, e.g., tools from other engineering domains like MATLAB/Simulink [16] or Dymola/Modelica [2].

In this paper, we present the MECHATRONICUML TOOL SUITE that is customized for the MECHATRONICUML method. It addresses all the requirements outlined in the previous paragraph. Hence, its particular set of features and broad range of integrated established tools is unique compared to related tools. We illustrate our tool on a running example from the automotive domain. Our example scenario is a coordinated overtaking situation in which the system ensures that the car being overtaken does not accelerate while the overtaking is in progress. By ensuring this behavior in our design, we improve the safety of such an overtaking action.

The paper is structured as follows. Section 2 introduces the MECHATRONICUML TOOL SUITE by walking through different steps in the tool's usage. Section 3 compares our tool to other tools in the same domain to highlight our unique features. Finally, Section 4 concludes the paper and outlines future work.

2. MECHATRONICUML TOOL SUITE

The MECHATRONICUML TOOL SUITE is an Eclipse-based tool that aims at providing support for the complete MECHATRONICUML process [13]. In this paper, we focus on the design of platform-independent software models (PIM). As depicted in Fig. 1, the PIM process is based on (formal) requirements and consists of four major modeling steps and three integrated analysis steps that shall ensure the correctness of the models. The result of this process is the platform-independent software model which will be the input for the subsequent platform-specific development.

In the following, we explain how our tool supports the software engineer to carry out the PIM process for developing the previously introduced coordinated overtaking of two cars. For doing this, each subsection will explain one process step. Our tool suite and a screencast are freely available via `https://trac.cs.upb.de/mechatronicuml/wiki/FSETools2014`.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2661665

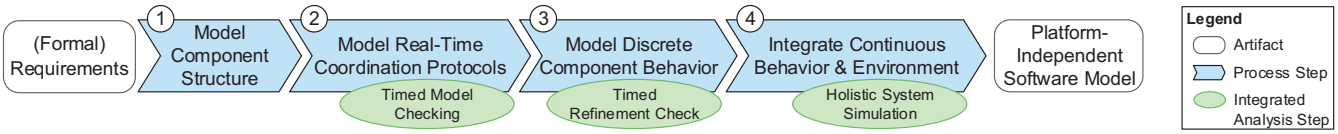

Figure 1: The MechatronicUML Process for the Platform-Independent Software Design

2.1 Model Component Structure

MechatronicUML follows a component-based approach for the holistic modeling of CPSs in collaboration between software and control engineers. Thus, the MechatronicUML Tool Suite comprises a visual component editor that provides engineers with components as building blocks for the construction of system architectures. Whereas each component entails a functional aspect of the system behavior, engineers may interconnect components via ports in order to enable inter-component coordination. Fig. 2 illustrates a component-based system architecture created using our tool. The architecture represents the mentioned overtaking scenario between the two cars. To guarantee safety, the cars autonomously coordinate their behavior such that the yellow car does not accelerate while it is overtaken by the red car.

Our tool provides modeling support for hybrid CPSs consisting of both discrete and continuous parts. On the one hand, software engineers may provide discrete software components with a state-based behavior, including interaction via asynchronous message exchange. In Fig. 2, the discrete components redSw and yellowSw exchange messages using the discrete ports Overtaker and Overtakee to coordinate their state-based behavior.

On the other hand, our tool enables the modeling of continuous components, representing system parts such as time-continuous feedback controllers or sensors. From the software engineering viewpoint, we regard such components as blackboxes, i.e., their behavior is either constituted by the environment, or to be developed as part of other disciplines such as control engineering. Thus, our tool only supports modeling of the continuous component interfaces in terms of data signals exchanged via continuous ports. For example, both continuous components of type Car in Fig. 2 comprise continuous ports (named velocityL and velocityR) to control the target velocity for the left/right engines during the overtaking.

Focusing on the integration of software and control engineering, MechatronicUML allows to adjust the continuous controlling strategy depending on the current discrete state. For this purpose, our tool provides engineers with hybrid ports as an interface between discrete and continuous system components. As an example, whenever component redSw changes to the discrete state of overtaking, it uses the hybrid port Velocity to increase the target velocity of the continuous component redCar.

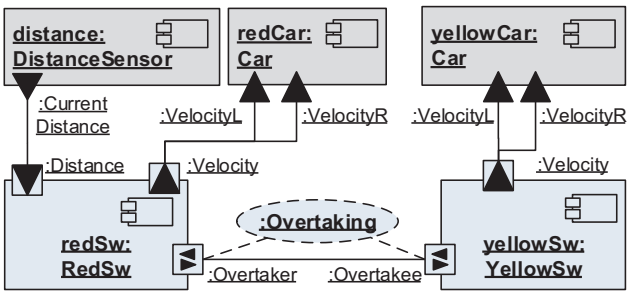

Figure 2: Component Instance Configuration for the Coordinated Overtaking Scenario in MechatronicUML

2.2 Model Coordination Protocols

The next process step is to ensure correct message-based coordinations between discrete components. Thus, each discrete port shall adhere to a contract called Real-Time Coordination Protocol (RTCP). Again, our tool provides a visual editor for this modeling task. A RTCP consists of two communicating partners, which we call roles, whereby each role represents a discrete port. Thus, for the coordination of the overtaking procedure, the software engineer specifies the RTCP Overtaking (cf. Fig. 3), which has the following two roles: Overtaker which represents the overtaking car and Overtakee which represents the car that is overtaken.

Within the same editor, due to the asynchronous communication, the software engineer also has to specify the exchanged messages, the properties of the incoming message buffers, and RTCP-specific quality-of-service assumptions like the message delay. In our example, the role Overtaker may send the messages request, laneChanged, and finish to Overtakee and may receive the messages accept and decline from it. Both roles have an incoming message buffer of size 5 and the message delay is assumed with 0 to 1 s.

Concerning the behavior of an RTCP, a software engineer has to model which role has to send and receive which message at which point in time. For doing this, our tool provides a visual editor for specifying extended hierarchical statemachines which we call Real-Time Statecharts (RTSCs). Fig. 3 shows in its lower part the two role RTSCs of RTCP Overtaking. In the following, we give a short informal description. Initially, both roles are in state noOvertaking.init. At first, role Overtaker can send message request to role Overtakee and waits in state requested at most 7 s for an answer. Due to the message delay, after 0 to 1 s, Overtakee receives the request and has 3 s to accept or decline it. If Overtakee accepts, it changes to state noAcceleration.noBraking which forbids him to accelerate or brake. If Overtaker receives a decline message or no message within 6 s, it changes back to state noOvertaking.init; if it receives an accept message, it changes to overtaking.init and starts the overtaking maneuver. After 5 s, Overtaker has changed the driving lane. Thus, it informs the Overtakee that braking is allowed again. Moreover, after at most 10 s, the overtaking is finished. When this is the case, Overtaker sends message finish to Overtakee and switches back to its initial state. After 0 to 1 s of message delay, Overtakee also switches back to its initial state.

Timed Model Checking

For ensuring correctness of the protocol design, our tool enables timed model checking of RTCPs. Thus, by checking all possible execution paths, we can prove that a RTCP will always fulfill specific properties. In particular, as the first step, the software engineer has to define the properties that shall hold by using a domain-specific variant of TCTL. For example, he can specify that the message buffers never overflow. Afterwards, the software engineer can start the automatic model checking procedure. In particular, the design model as well as the properties are exported by a chain of model-transformations [8] to input models of the model checker Uppaal [18]. Then, Uppaal verifies the given properties and returns the results. In our case, all properties are satisfied.

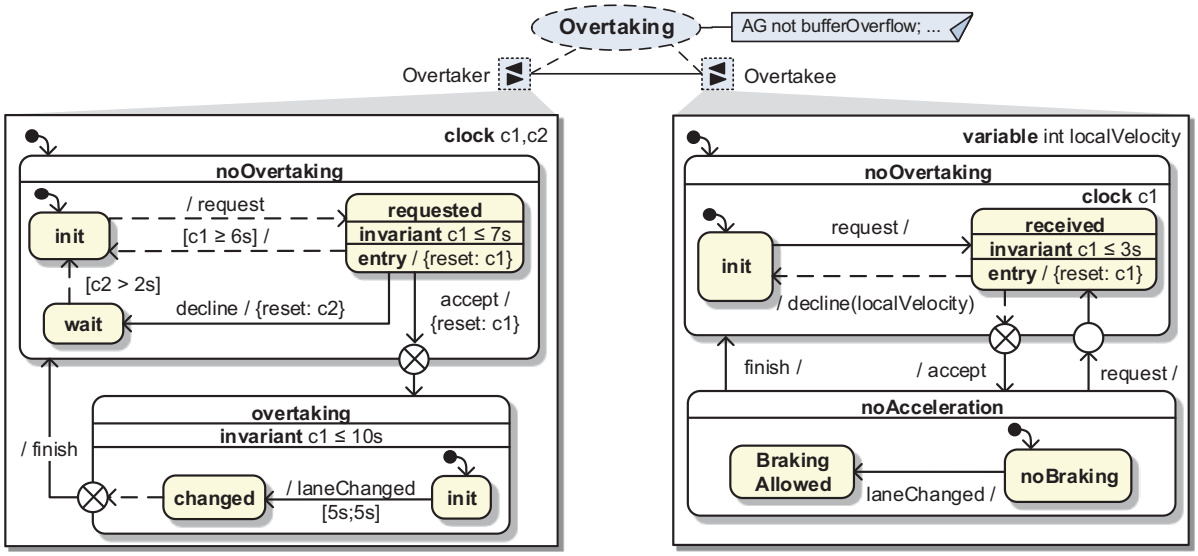

Figure 3: RTCP Overtaking including its two role RTSCs

2.3 Model Discrete Component Behavior

The behavior of discrete components and their discrete ports is defined by RTSCs, too. Each component has exactly one RTSC (the component RTSC) that embeds the RTSCs of the discrete ports (the port RTSCs) in parallel regions. Hybrid ports are not described with a RTSC as they either only read an incoming signal or write an outgoing signal. Thus, they are used as variables inside the component RTSC. Hybrid in-ports correspond to read-only variables, while hybrid out-ports may be read and written.

Discrete ports of different components communicate with each other according to a RTCP. Therefore, each discrete port refines one role of a RTCP, i.e., its behavior may differ but must be compliant to the behavior of the role. In our tool, we support the software engineer by automatically copying the properties of the role, e.g., its message buffer specification and its role RTSC, to the port.

Usually, the ports of a discrete component are interdependent, i.e., the behavior of one port depends on the behavior of other ports. Among others, discrete port RTSCs need access to the hybrid ports for reading and writing their values. In our example, the port RTSC Overtaker of RedSw reads values from the hybrid port Distance and writes values to the hybrid port Velocity for controlling the overtaking. Typically, discrete ports of one component are also depend on each other. For example, assuming that YellowSw also has a discrete port overtaker for overtaking another car, YellowSw may only start overtaking if it is currently not being overtaken itself.

Timed Refinement Check

The software engineer needs to resolve the dependencies mentioned above manually. As a consequence, he needs to modify the port RTSC by inserting additional states, transitions, and conditions. These modifications, however, must not invalidate the model checking results that were obtained for the RTCP in Step 2 of our process in Fig. 1. Therefore, a port RTSC must be a correct refinement of the role RTSC with respect to a formal refinement definition. If the port RTSC is not a correct refinement of the role RTSC, then the communication between systems will be unsafe because the incorrectly refined port does not behave as expected by a communication partner. Therefore, we integrated an automatic refinement check into our tool that selects out of six refinement definitions one that is suitable and then checks whether the port RTSC is a correct refinement of the role RTSC [11].

2.4 Integrate Continuous Behavior and Environment

In parallel to the software engineer that models the discrete component behavior, control and mechanical engineers model the behavior of the continuous components and the environment model. For doing this, they typically use languages/tools like MATLAB/Simulink [16] or Modelica/Dymola [2] as they enable a simulation by numerical integration. Afterwards, the combination of discrete and continuous components (the so-called holistic system) needs to be validated concerning the correct interaction of all parts. Due to the complexity of this model, the validation is performed using numerical simulations. Therefore, the software software engineer has to transform its discrete components (incl. the behavior) into input models of these simulation languages. For automating this task, we developed transformations from MECHATRONICUML to MATLAB/Simulink [12] and to Modelica/Dymola [17].

By performing different simulation runs, engineers can systemically test the hybrid behavior of the holistic system within the simulation tool. As an exemplary test result, Fig. 4 shows a plot of a Dymola simulation run of our overtaking scenario that uses the physics of small robots that represent our cars. Within this run, the red car drives behind the yellow car with a higher velocity (red solid line) than the yellow car (blue dashed line with diamonds). At the simulation time of 6.69 s, the red car is 0.1 m behind the yellow car (green dotted line). As defined in the RTSC, it sends an overtaking request to the yellow car (the purple dashed dotted line switches to 1). The yellow car confirms the request and the red car accelerates its speed (red solid line). At the simulation time of 10.08 s, the distance of the red car to the overtaken yellow car gets greater than 0.1 m and the overtaking procedure is finished successfully.

3. RELATED TOOLS

We consider related tools for specifying and analyzing software for cyber-physical systems. Component models for cyber-physical systems and related real-time embedded systems [14] support (timing) analyses but do not integrate simulation tools. Simulation tools

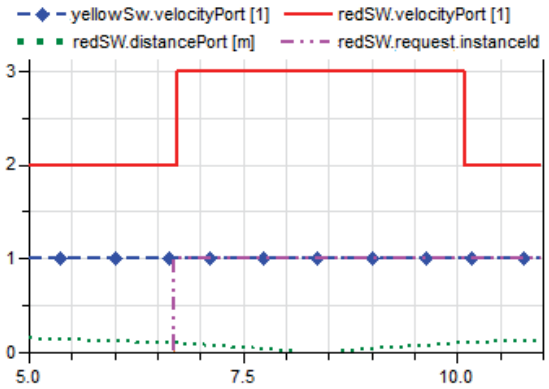

Figure 4: Plot of a Simulation Run for the Overtaking Scenario

like Dymola [2] and MATLAB/Simulink [16] do not natively support message-based communication and provide no scalable verification of the system models. Uppaal Port [10], a derivative of the model checker UPPAAL [18] for specifying component-based systems, focuses on formal verification but only supports passive components and has no support for message-based communication. Development tools supporting the automotive standard AUTOSAR [6] like SystemDesk [3] or ASCET [5] rely on external tools like MATLAB/Simulink for specifying component behavior and, therefore, do not support formal verification. Modeling tools for UML and SysML such as Papyrus [4] and Rhapsody [15] support the specification of the software and typically provide integration with simulation tools but do not support formal verification. A detailed discussion of further related approaches and tools can be found in our technical report [1].

4. CONCLUSION

In this paper, we introduced the MECHATRONICUML TOOL SUITE for carrying out the MECHATRONICUML method. In particular, our tool enables software engineers to develop the discrete software of CPSs with focus of correctness of the models and interoperability with models of other disciplines.

At the moment, we extend our tool suite in three directions. First, we develop editors as well as analysis techniques for the reconfiguration of the system at runtime. Second, we implement a back-transformation of UPPAAL counterexamples into the domain of MECHATRONICUML. Thus, a software engineer can detect errors within its own models without the need to understand UPPAAL's language. Third, we enrich our tooling concerning platform-specific development tasks like allocation, deployment, middleware configuration, and target code generation.

5. ACKNOWLEDGEMENTS

We thank our students that helped us to develop the MECHATRONICUML TOOL SUITE. This work is partially developed in the Leading-Edge Cluster 'Intelligent Technical Systems OstWestfalenLippe' (it's OWL). it's OWL is funded by the German Federal Ministry of Education and Research (BMBF).

6. REFERENCES

[1] S. Becker et al. The MechatronicUML design method - process and language for platform-independent modeling. Technical Report tr-ri-14-337, Heinz Nixdorf Institute, University of Paderborn, 2014. Vers. 0.4.

[2] Dassault Systemes. *Dymola*. http://www.dymola.com.

[3] dSPACE GmbH. *SystemDesk Product Homepage*. https://www.dspace.com/systemdesk.

[4] The Eclipse Foundation. *Papyrus*. http://www.eclipse.org/papyrus/.

[5] ETAS GmbH. *ASCET Software-Produkte*. http://www.etas.com/ascet.

[6] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. Autosar - a worldwide standard is on the road. In *Proc. of the 14th Intern. VDI Congress Electronic Systems for Vehicles 2009*, 2009.

[7] J. Gausemeier, F.-J. Rammig, and W. Schäfer, editors. *Design Methodology for Intelligent Technical Systems*. Lecture Notes in Mechanical Engineering. Springer, 2014.

[8] C. Gerking. Transparent Uppaal-based verification of MechatronicUML models. Master's thesis, University of Paderborn, May 2013.

[9] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time UML designs. In *Proceedings of the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 38–47. ACM, 2003.

[10] J. Håkansson, J. Carlson, A. Monot, P. Pettersson, and D. Slutej. Component-based design and analysis of embedded systems with UPPAAL PORT. In S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, editors, *Automated Technology for Verification and Analysis*, volume 5311 of *Lecture Notes in Computer Science*, pages 252–257. Springer Berlin Heidelberg, 2008.

[11] C. Heinzemann, C. Brenner, S. Dziwok, and W. Schäfer. Automata-based refinement checking for real-time systems. *Computer Science - Research and Development*, 2014. doi: 10.1007/s00450-014-0257-9.

[12] C. Heinzemann, J. Rieke, and W. Schäfer. Simulating self-adaptive component-based systems using MATLAB/Simulink. In *IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, SASO '13, pages 71–80. IEEE, Sept. 2013.

[13] C. Heinzemann, O. Sudmann, W. Schäfer, and M. Tichy. A discipline-spanning development process for self-adaptive mechatronic systems. In *Proc. of the 2013 Intern. Conf. on Software and System Process*, ICSSP 2013, pages 36–45. ACM, 2013.

[14] P. Hosek, T. Pop, T. Bures, P. Hnetynka, and M. Malohlava. Comparison of component frameworks for real-time embedded systems. In L. Grunske, R. H. Reussner, and F. Plasil, editors, *Component Based Software Engineering*, volume 6092 of *Lecture Notes in Compute Science*, pages 21–36. Springer, Berlin/Heidelberg, 2010.

[15] IBM. *Rational Rhapsody Designer for Systems Engineers*. http://ibm.com/software/products/rhapsody.

[16] The MathWorks, Inc. *Simulink Product Homepage*. http://www.mathworks.com/products/simulink.

[17] U. Pohlmann, J. Holtmann, M. Meyer, and C. Gerking. Generating Modelica models from software specifications for the simulation of cyber-physical systems. In *Proceedings of the 40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE Xplore, Aug. 2014.

[18] Uppsala and Aalborg University. *Uppaal*. http://http://www.uppaal.org.