Tool Support for Developing Advanced Mechatronic Systems: Integrating the Fujaba Real-Time Tool Suite with CAMeL-View

Sven Burmester, Holger Giese, Stefan Henkler, Martin Hirsch, Matthias Tichy
Software Engineering Group
University of Paderborn, Germany
{burmi,hg,shenkler,mahirsch,mtt}@uni-paderborn.de

Alfonso Gambuzza, Eckehard Münch, Henner Vöcking
Control Engineering Group
University of Paderborn, Germany
{gamba,muench,hevoe}@rtm.uni-paderborn.de

Abstract

The next generation of advanced mechatronic systems is expected to use its software to exploit local and global networking capabilities to enhance their functionality and to adapt their local behavior when beneficial. Such systems will therefore include complex hard real-time coordination at the network level. This coordination is further reflected locally by complex reconfiguration in form of mode management and control algorithms. We present in this paper the integration of two tools which allow the integrated specification of real-time coordination and traditional control engineering specifically targeting the required complex reconfiguration of the local behavior.

1 Introduction

For mechatronic systems [2], which have to be developed in a joint effort by teams of mechanical engineers, electrical engineers, and software engineers, the advances in networking and processing power provide many opportunities. It is therefore expected that the next generation of advanced mechatronic systems will exploit these advances to realize more intelligent solutions where software is employed to exploit local and global networking capabilities to optimize and enhance their functionality by operating cooperatively. The cooperation in turn permits these systems to decide when to adapt their local behavior taking the information of cooperating subsystems into account.

The development of such advanced mechatronic systems will therefore at first require means to develop software for the complex hard real-time coordination of its subsystems at the network level. Secondly, software for the complex reconfiguration of the local behavior in form of mode management and control algorithms is required, which has to proper coordinate the local reconfiguration with the coordination at the network level.

The envisioned approach is complicated by the fact that classical engineers and software engineers employ different paradigms to describe their aspect of these systems. In software engineering discrete models such as state charts are frequently used to describe the required interaction, while the classical engineers employ continuous models to describe and analyze their control algorithms.

To enable the model-based development of the outlined advanced mechatronic system, an integration between these two paradigms is required which fulfills the outlined requirements. To provide an economically feasible solution, the required integration must further reuse the concepts, analysis techniques, and even tools of both involved paradigms where possible.

We present in this paper the integration of two tools to bridge the development of software engineering for real-time systems with control engineering: the open source UML CASE tool Fujaba Real-Time Tool Suite and the CAE tool CAMeL-View. The employed concepts for the real-time coordination [9] and tool support for it [4] have
been presented in earlier work. For the the local reconfiguration only the concepts have been presented [7], while in this paper the developed tool support is described.

In the remainder of this paper, we first introduce the modeling concepts for the integrated description of discrete and continuous models in Section 2. Then, we outline in Section 3 how this conceptual integration has to be paralleled at the execution level. Afterward, we discuss the benefits of our approach with respect to analysis capabilities in Section 4 and compare our approach with the related work in Section 5. We finally provide our conclusions and an outlook on planned future work. In an additional appendix, we describe how we plan to demonstrate the integration of the FUJABA REAL-TIME TOOL SUITE and CAMeL-View by means of an example.

2 Integration at the model level

Modeling advanced mechatronic systems require the integration of modeling approaches used in software engineering and traditional engineering disciplines. To describe our approach for modeling these systems, we first introduce MECHATRONIC UML for specifying discrete parts of a system in Section 2.1. As mechatronic systems have continuous parts too, we introduce in Section 2.2 block diagrams. We finally provide our approach for modeling the required integration of the different modeling paradigms in Section 2.3.

2.1 Discrete Specification

The software architecture of the considered mechatronic systems is specified in MECHATRONIC UML [6] with components which are based on UML[11] components. The components are self-contained units, with ports as the interface for communication. A component can contain other components and events can be delegated from the top-level component to its subcomponents. The internals of a component are modeled in an extended version of UML State Machines. Ports are the only external access point for components and their provided and required interfaces specify all interaction which occurs at the port. The interaction between the ports of the components takes place via connectors, which describe the communication channels.

As UML state machines are not sufficient to describe complex time-dependent behavior (cf. [10]), we introduced Real-Time Statecharts (RTSC) [5] as an appropriate modeling language for the discrete real-time behavior of a component and for the event-based real-time communication between components. Real-Time Statecharts contain various Timed Automata [1] related constructs. In contrast to Timed Automata, firing a transition in a RTSC consumes time.

2.2 Continuous Specification

Mechatronic systems contain software to continually control the mechanic behavior. The standard notation for control engineering is block diagrams which are used to specify feedback-controllers as well as the controlled plant. Consequently, our approach uses block diagrams for the structural and behavioral specification of continuous components.

Block diagrams generally consist of basic blocks, specifying behavior and hierarchy blocks that group basic and other hierarchy blocks. Each block has input and output signals. The unidirectional interconnections between the blocks describe the transfer of information. The behavior of basic blocks is usually specified by differential equations, specifying the relationship between the block’s inputs and outputs.

2.3 Hybrid Specification: Integration of Feedback-Controller Configurations

As mentioned in Section 1, for mechatronic systems, it is not sufficient to specify how discrete states of a component change: Dependent on the current state, the components have to apply different feedback-controllers. In [6], we introduced a new approach for the specification of hybrid behavior, which integrates discrete and continuous behavior, and even supports reconfiguration.

The idea is to associate to each discrete state of a component a configuration of subordinated components. Such a configuration consists of the subordinated components and their current connection. These components are either pure continuous components (feedback-controllers) or discrete or hybrid components. If the subordinated components are discrete or hybrid components, the configuration of these subordinated components consists also of the current state of the subordinated discrete or hybrid component. As hybrid components have a dynamic interface and show just the ports required in their current state, also a configuration of components show just the in- and out-ports which are really required or used in the current state of the superordinated component.

This kind of modeling leads to implied state changes: When the superordinated component changes its state, this implies reconfiguration of the subordinated components. The subordinated components reconfigure their communication connections and –if specified– their discrete state. Such implied state changes can imply further state changes, if the subordinated components embed further components.

This kind of modeling leads to reconfiguration across multiple hierarchical levels. An example is presented in the appendix. Compared to state of the art approaches, this approach has the advantage that models are of reduced size and that analyses require reduced effort (see Section 4).
3 Integration at runtime

Our approach for developing reconfigurable mechatronic systems applies the model-driven development approach to develop software systems at a high level of abstractions to enable analysis approaches like model checking. Therefore, ideally, we start with platform independent models to enable the compositional formal verification. Afterward, the platform independent model must be enhanced with platform specific information to enable code generation. The needed platform specific information is based on a platform model, which specifies the platform specific worst case execution times. After providing the platform specific information, we generate code from the models. In the remainder of this section, the generated code is described. Therefore, we introduce the evaluation of the system’s components. Due to continuous and discrete parts of the considered mechatronic systems we have to consider data flow and event based evaluation. Complex reconfigurations lead to a lot of possible evaluation configurations and requires synchronization between the discrete and continuous parts. Our approach assures reconfigurations by the evaluation of the data flow and further we assure the data flow despite of reconfiguration. In the next subsections (Section 3.1, Section 3.2, and Section 3.3) we introduce our approach by considering first the discrete evaluation, then the continuous evaluation, and finally the hybrid evaluation.

3.1 Discrete Evaluation

When a component is evaluated, it triggers periodically the evaluation of its embedded components. As not every embedded component belongs to every configuration, it depends on the current discrete state of the component which of the embedded components are evaluated. Then the triggered components will themselves trigger their embedded components (in dependency of their discrete states) and so forth. Further, the association of configurations to discrete states leads to reconfiguration when a component changes its discrete state (e.g. when receiving an event). Due to the implied state changes (see Section 2.3), this leads to reconfiguration of the embedded components which are pure discrete, pure continuous or hybrid components.

3.2 Continuous Evaluation

The continuous parts of a configuration describe the data flow between the continuous inputs and the continuous outputs of the system. To ensure stability of the continuous part of the system, the data flow may not be interrupted within a computation step. Consequential, reconfiguration may only occur between two computation steps. Therefore, we separate execution of the continuous, data flow-orientated, and the discrete, event-based, parts: At the beginning of a period, the continuous system parts are evaluated. This is followed by the evaluation of the discrete system parts. Thus, we ensure that the reconfiguration—which takes place in the discrete part– occurs after the computation of the continuous system part is finished.

3.3 Hybrid Evaluation

Besides separating the continuous system parts from the discrete ones, it has to be managed which components need to be evaluated in which discrete state. Enhancing the top-level component with this information is usually not feasible as the number of global states grows exponentially with the number of components. Therefore, we compose the whole system as a tree structure consisting of single Hybrid Components to obtain an efficient implementation. Each Hybrid Component contains the information about its discrete and continuous parts—which may consist of system parts of the embedded components— itself. By the presented integration at runtime, we ensure consistent, correct data flow and an efficient implementation in spite of complex reconfiguration. Our seamless approach is realized by the Fujaba Real-Time Tool Suite in combination with the CASE Tool CAMeL. Both tools export hybrid components for the integration on the modeling level (see Section 2.3) and they export C++ code which is integrated to realize the hybrid, reconfiguration behavior.

4 Analysis capabilities

For the outlined Mechatronic UML approach, two specific verification tasks for the resulting systems are supported.

First, the Mechatronic UML approach supports model checking techniques for real-time processing at the network level. It addresses the scalability problem by supporting a compositional proceeding for modeling and verification exploiting the component model and the corresponding definition of ports and connectors as well as patterns [9].

Secondly, a restricted subset of the outlined hierarchical component structures for modeling of discrete and continuous control behavior can be checked for the consistent reconfiguration and real-time synchronization w.r.t reconfiguration taking proactive behavior into account [7, 8].

As the second approach can be embedded into the first one, a combination of both approaches cover the whole real-time coordination issues from the network level down to the reconfiguration of the lowest level components.
5 Related Work

Related tools and techniques to MECHATRONIC UML are CHARON, Hybrid UML with HL³, HyROOM/HyCharts/Hybrid Sequence Charts, Massacio and Giotto, Matlab/Simulink/Stateflow, Ptolemy II, and UML² [3].

All presented tools and techniques support the specification of a system’s architecture or structure by a notion of classes or component diagrams. All approaches support modular architecture and interface descriptions of the modules. Nevertheless, they do not respect that a module can change its interface due to reconfiguration which can lead to incorrect configurations.

CHARON, Masaccio, HybridUML with HL³, UML², HyROOM, and HyCharts have a formally defined semantics, but due to the assumption of zero-execution times or zero-reaction times, most of them are not implementable, as it is not realizable to perform a state change infinitely fast on real physical machines. CHARON is the only approach providing an implementable semantics. HyCharts are implementable after defining relaxations to the temporal specifications. They respect that idealized continuous behavior is not implementable on discrete computer systems. Further, CHARON provides a semantic definition of refinement which enables model checking in principle. Ptolemy II even provides multiple semantics and supports their integration.

Although most of these approaches enable ruling the complexity by a modular, component-based architecture and by behavioral models that support history and hierarchical and orthogonal states, reconfiguration across multiple hierarchical levels as required for the advanced mechatronic systems envisioned and provided by the presented approach is supported by none of them.

6 Conclusion and Future Work

The tool integration of the CAE Tool CAMeL-View and the CASE Tool Fujaba Real-Time Tool Suite enables the application of our approach by continuing using well-approved tools. It does not only integrate models, but also the synthesized source code.

Acknowledgments

We thank all students who helped building the FUJABA REAL-TIME TOOL SUITE within student research projects, master, and bachelor theses, namely Vadim Boiko, Margarete Kudak, Vladimir Pauls, Matthias Schwarz, Björn Schwerdtfeger, and Andreas Seibel.

References


A Demonstration

The example stems from the RailCab\(^1\) research project at the University of Paderborn. In this project, autonomous shuttles are developed which operate individually and make independent and decentralized operational decisions.

The modular railway system combines sophisticated undercarriages with the advantages of new actuation techniques as employed in the Transrapid\(^2\) to increase passenger comfort while still enabling high speed transportation. In contrast to the Transrapid, the existing railway tracks will be reused.

One particular problem is to reduce the energy consumption due to air resistance by coordinating the autonomously operating shuttles in such a way that they build convoys whenever possible. Such convoys are built on-demand and require a small distance between the different shuttles such that a high reduction of energy consumption is achieved. Coordination between speed control units of the shuttles becomes a safety-critical aspect and results in a number of hard real-time constraints, which have to be addressed when building the control software of the shuttles. Additionally, different controllers are used to control the speed of a shuttle as well as the distance between the shuttles. The controllers have to be integrated with the aforementioned real-time coordination.

In this demonstration, we will first show how the structure of the software is specified by component diagrams (Section A.1). Then we will present the behavior of the discrete components and the continuous control components respectively (Section A.2). Thereafter, we show how the continuous components are embedded into the discrete real-time behavior (Section A.3). After code generation and compilation, we show the behavior of the system using a simulation environment (Section A.4).

A.1 Structural Modeling

We use the Fujaba Real-Time Tool Suite for modeling the architecture. Figure 1 shows the internal structure of Shuttle. Similar to the Shuttle component, which is composed of multiple other component instances, the types of the subordinated instances are defined by further compositions. This leads to an architectural description of Shuttle, consisting of multiple layers.

Figure 2 shows the DriveTrain component which is composed of instances of the components AccelerationControl, PosSensor, VelSensor, Storage, and ReferenceSpeed. The instance ac:AccelerationControl determines the required acceleration which is delegated to the output of the superordinated DriveTrain component. To determine this output, it obtains the current position from ps:PosSensor, the current velocity from vs:VelSensor, the required position from st:Storage where the required trajectory is stored, and the required velocity from the rs:ReferenceSpeed instance which determines the required velocity based on optimizations e.g. for fast traveling or for slow low cost traveling. Thus, DriveTrain encapsulates ac:AccelerationControl and the components that provide the input signals.

AccelerationControl embeds again five further instances (cf. Figure 3). AccelerationControl embeds the component instances velCtrl:VelocityControl, posCtrl:PositionControl, pilCtrl:PilotControl, sum:Sum, and fade:Fading. velCtrl:VelocityControl, posCtrl:PositionControl, and pilCtrl:PilotControl are the velocity controller, the position control block, and the pilot control block. The blocks pilCtrl:PilotControl and sum:Sum are used to build a configuration that realizes the position controller with and without pilot control. The block fade:Fading implements cross-fading. The inputs of AccelerationControl can be connected to the inputs of the embedded component instances. The output is provided either by velCtrl or by fade.

A.2 Behavioral Modeling

After modeling the structure of the shuttle convoy example, we specify the behavior of each component. We first show the specified discrete real-time behavior, on the level of the real-time coordination patterns [4] and the individual discrete components which is modeled by the Fujaba Real-Time Tool Suite. After the presentation of the real-time behavior specification, we show the behavior of the velocity control component (see Figure 4). Input for the controller is the velocity set point. Output is the force set point for the shuttle’s engine. This component is modeled by CAMeL-View.

A.3 Hybrid Integration

Figure 5 shows a hybrid reconfiguration chart that describes the behavior of the AccelerationControl component (see Figure 3). It consists of three discrete states: The states VelocityControl, PositionControl, or PositionControlWithPilotControl are associated with the continuous controller component configurations which have been sketched previously. Further, note that in this example all embedded components are basic continuous components that do not show discrete or hybrid behavior. In Real-Time Statecharts, the transitions are associated with deadlines, indicating a time interval when firing the transition has to be finished at the earliest and at the latest.

In the following, we describe the hybrid reconfiguration

\(^1\)http://nbp-www.upb.de/en/
\(^2\)http://www.transrapid.de/en/index.html
chart of the component Drive Train (cf. Figure 6). We define two states for component Drive Train: Vel which represents the system to be under velocity control and Pos which represents that the system is under position control. In substate Vel, Drive Train’s configuration consists of the instance $ac$, and of the instances $vs$ and $rs$, which are connected with two inputs of $ac$. In this configuration, $ac$ is required to be in state VC (VelocityControl), indicated by the angular brackets. When Drive Train switches to state Pos, it reconfigures as follows: The state change of Drive Train implies a state change of $ac$ from VC to PC. Due to this implied state change, $ac$’s interface changes so that it requires in addition to $v_{\text{cur}} - x_{\text{cur}}$ and $x_{\text{req}}$ instead of $v_{\text{req}}$ as input signals. To feed the input signals correctly, we specified that they are connected with the outputs of $ps$, $st$, and $vs$.

Next, we describe Shuttle’s hybrid reconfiguration chart. It realizes the discrete real-time coordination and it embeds—besides others—an instance of the hybrid component Drive Train. The coordination leads to state changes, indicating if the shuttle is part of a convoy or not and—in case it is part of a convoy—if it is the leading shuttle of the convoy. Each of these states is associated with a different configuration where the Drive Train-instance is in different states. Therefore, a state change of shuttle implies a state change of Drive Train which implies a state change of AccelerationControl. This models reconfiguration via multiple hierarchical levels.

### A.4 Runtime

To generate C++ code from our models as described in Section 3 for a PC platform, we annotate our models, as described in Section A.3 and Section A.2, with platform specific details, e.g. the worst case execution time (WCET) of models/model parts. Then, we can perform a schedulability analysis to check if the platform specific model is feasible. After code generation and compilation, we show the behavior of the system using a simulation environment of the CAE Tool CAMeL-View. Figure 7 (simple convoy) shows four shuttles in a convoy (bottom window). On the upper left hand plot the position of the shuttles are shown. On the upper right hand plot the acceleration of the first and second shuttle is shown. On the lower left plot the reference values for the second shuttle is shown. The lower trajectory shows the target position of the shuttle to avoid a crash. The middle trajectory shows the target position of the distance controller and the upper trajectory shows the minimum of bother other trajectories. The lower right plot shows the distance between the shuttles.

### B Screen dumps
Figure 1. Structure of Shuttle

Figure 2. Structure of Drive Train
Figure 3. Structure of Acceleration Control

Figure 4. Velocity control block diagram
Figure 5. Hybrid reconfiguration chart of AccelerationControl with fading transitions

Figure 6. Hybrid reconfiguration chart of DriveTrain reconfiguring its embedded components
Figure 7. Simulation Environment