

The Fujaba Automotive Tool Suite*

Kahtan Alhawash, Toni Ceylan, Tobias Eckardt, Masud Fazal-Baqaie, Joel Greenyer, Christian Heinzemann, Stefan Henkler, Renate Ristov, Dietrich Travkin, Coni Yalcin

Software Engineering Group

University of Paderborn

Warburger Str. 100

D-33098 Paderborn, Germany

[alhawash|crowdy|tobie|masudf|jgreen|chris227|shenkler|renate|travkin|coni81]@upb.de

ABSTRACT

Automotive systems contain a large number of software controllers that interact in order to realize an increasing number of functions. The controllers are typically developed separately by different suppliers. Therefore, errors in the overall functionality are often detected late in the development process or even remain undetected. This affects the quality and safety of the systems and may lead to expensive recalls.

We propose to specify the communication behavior more precisely in the early system design by modeling the controllers' interactions using formal sequence diagrams (LSCs). Based on these behavior models we're able to automatically synthesize state machines, which can be used to generate code for the communication behavior of the controllers. We provide a prototypical integrated modeling environment based on Fujaba which supports specifying requirements, modeling the component architecture, and component behavior as well as the state machine synthesis.

1. INTRODUCTION

Today's cars provide a continuously growing number of enhanced functions like a car's windows when it starts raining and no one is in the car, or automatically adapting the speed to that of a preceding car. The rising complexity of these functions requires the interaction of numerous controllers, sensors, and actuators to achieve the desired behavior. In order to handle the system's complexity and increase its flexibility, more and more functions are realized by software.

The controllers are usually developed separately by different suppliers. These rely on interface and requirements specifications given by car manufacturers. Therefore, the controllers are tested as single units. Integration tests, especially tests addressing communication behavior, are run very late in the development process. Not sufficiently considering the communication behavior in the early development process, negatively affects the system's quality and safety. Failures and errors due to wrong communication behavior are revealed too late and may result in product recalls.

Furthermore, inconsistencies in system design may arise due to heterogenous tools which are typically used in the automotive industry today: DOORS¹ is often used for re-

quirements engineering, an AUTOSAR²-compatible editor is used to specify the system architecture, and MATLAB Simulink³ for the specification of the components' internal controller behavior. Finally, code generators like TargetLink⁴ are used to generate the controller code.

We approach the problems stated above by formally specifying the communication behavior earlier in the development process, starting in the requirements engineering phase. The requirements are refined step-wise to iteratively build the system architecture and formally specify its behavior very early in the system design. This would enable car manufacturer to validate the overall system functionality in earlier development phases and to provide suppliers with more precise specifications of single controllers. Additionally, we show how to establish traceability between the design artifacts and the elaborated requirements. To support the overall process, we implemented a modeling environment based on Eclipse and Fujaba. Since all editors operate on the same underlying model and provide just different views on it, consistency between all artifacts and across all development phases is achieved.

2. EXAMPLE

The example that we use throughout this paper is that of an adaptive cruise control (ACC). The ACC supports the driver of a car by keeping the current speed and by adapting the speed to preceding cars automatically in order to maintain a safe distance.

A simplified system architecture for the ACC is shown in Figure 1. The sensors, actuators, and the user interface are omitted here. The ObjectRecognition component is responsible for detecting preceding cars by interpreting inputs from a front radar sensor. The information about preceding cars is sent to the AdaptiveControl component, which computes the maximum speed of the car so that the safety distance is maintained. The new speed of the car is set in the AccelerationControl component, which is responsible for controlling the actual speed of the car. The internal (controller) behavior, e.g. the calculation of the new speed by the AdaptiveControl, is not considered here. These are typical tasks for control engineering, where control behavior is specified by continuous block diagrams, usually using MATLAB Simulink.

However, the three components have to interact in order

*This work was partly 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.

¹<http://www.telelogic.com>

²<http://www.autosar.org>

³<http://www.mathworks.com>

⁴<http://www.dspace.com>

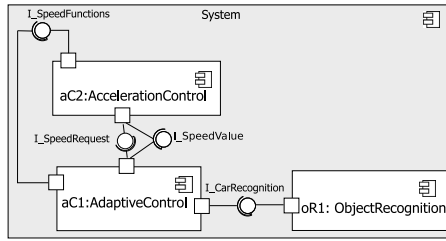


Figure 1: Simplified ACC component diagram

to achieve the correct ACC behavior. To achieve a correct system functionality, a correct communication behavior of the components is needed in addition to the internal behavior. In this paper, we concentrate on the communication behavior, which is modeled in terms of asynchronous messages that are sent via ports.

3. THE DEVELOPMENT PROCESS

The development process in the automotive industry today captures the system’s requirements in a structured way, typically using the tool DOORS. In order to ease integration of structured requirement specifications into our tool environment, we specify requirements in a goal-oriented way [6], using goal trees (Figure 2). This helps to iteratively define requirements, identify components and describe communication behavior. These goals are therefore directly linked to the according parts in the system architecture (components) and communication behavior descriptions.

The system architecture is modeled using UML component diagrams (see Figure 1). Component diagrams have become the standard means to specify the architecture of automotive systems today, since a variant of component diagrams also forms part of the AUTOSAR standard. Components can send and receive messages via ports. The ports are typed over interfaces which contain a set of message types that can be sent or received (Figure 3).

In the early design process found in practice today, it is specified what kind of data values or messages can be communicated between the components, but the order of messages and the events which trigger communication sequences are not formally specified. Thus, the behavioral specification given to suppliers is often incomplete. Different from industry processes, we propose to specify communication between components in the early design phase by using a formal variant sequence diagrams (see section 5). This enables an early analysis of the communication behavior before the specification is given to suppliers. In our implemented automotive tool suite, the sequence diagrams are modeled based on the system architecture model and thus syntactic consistency is ensured. Furthermore, we link components and sequence diagrams to the goals which they fulfill so that design problems can be traced to the according requirements.

All parts of the system description are iteratively refined during the specification process. The communication behavior for each component can then be synthesized automatically. Our synthesis generates one statechart per component. From these statecharts, source code can be generated by standard techniques [2]. This code can be used for early prototyping, or code generation may be combined with the

refined (control) behavior of the component, which is developed by component suppliers [1].

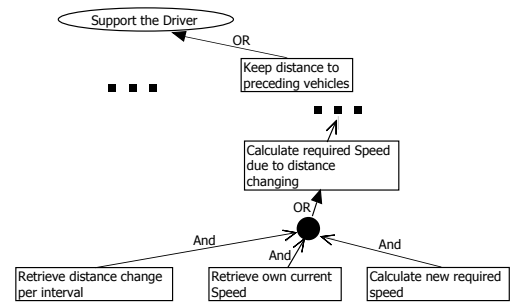


Figure 2: Goal tree excerpt of the ACC example

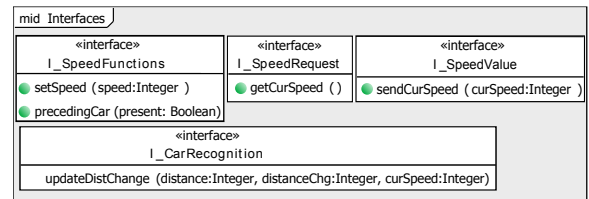


Figure 3: Interfaces – Specifying communication between components

4. MODELING COMMUNICATION BEHAVIOR

In this section, we show how to specify the communication behavior by considering two refined goals from the example “Retrieve own current speed” and “Calculate new required speed” (Figure 2).

This communication behavior is described with a subset of Live Sequence Charts (LSCs) [5]. The syntax, however, is based on UML 2.1.

We specify a separate LSC for each goal – Figure 4a refines the behavior for “Retrieve own current speed”, Figure 4b refines the behavior for “Calculate new required speed”.

According to [5], we distinguish between mandatory behavior and possible scenarios. To describe what must happen, we split sequence diagrams into a prechart and a main chart. If the behavior of the prechart is observed, the behavior of the main chart is executed. The LSC in Figure 4a accordingly specifies that if the AccelerationControl receives `getCurSpeed()` from the AdaptiveControl, it will answer with `sendCurSpeed(curSpeed)`. The LSC in Figure 4b specifies that if the current speed has been retrieved, the AdaptiveControl has to calculate the new speed by calling `calcMaxSpeed(...)` and send it to the AccelerationControl using `setSpeed(...)`. Specifying system behavior in such manner thus enables early execution and analysis of specifications.

Several LSCs can be active at the same time. Since the execution of one LSC can lead to the fulfillment of the prechart of another, one LSC execution can be a trigger for another LSC execution. This way, the execution of the main chart in Figure 4a triggers the LSC in Figure 4b.

Further on, we distinguish between synchronized method calls for intra-component behavior (`calcMaxSpeed(...)`) and asynchronous messages for inter-component communication.

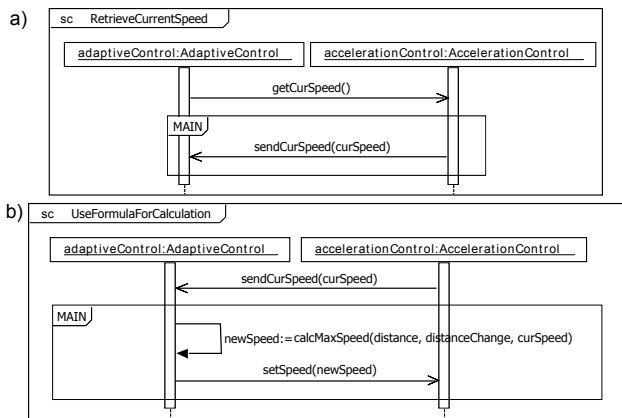


Figure 4: a) Sequence diagram for speed retrieval
b) Sequence diagram for calculation of new speed

The central advantage of using LSCs is their formality, which enables synthesis to one statechart for each component. Additionally, the specification can already be simulated and analyzed during the requirement specification phase. This solves one of the major problems of the current development process employed in industry where incorrect communication behavior is often discovered not until implementation phase.

5. SYNTHESIS OF COMMUNICATION BEHAVIOR

Based on the system's inter-component communication behavior specified by a number of LSCs, we are now able to automatically synthesize intra-component behavior for each component as statecharts similar to the methods proposed in [3] and [4]. We adapted the implementation of [3] to synthesize LSCs [4].

We generate one statechart for each component with one orthogonal state for each LSC. The top orthogonal state of the statechart for the AdaptiveControl (Figure 5) results from the LSC RetrieveCurrentSpeed, the bottom orthogonal state results from the LSC UseFormulaForCalculation (Figures 4a and b).

For generating the states for each LSC, we insert states on the component's lifeline before and after each sending or receiving of a message. The transition between those states becomes either a raised – sending a message – or a trigger event – receiving a message – with the message as the transition label. In the example, two states `InitReceivesendCurSpeed` and `ReceivesendCurSpeed` are generated for the reception of `sendCurSpeed(...)`. Furthermore, a trigger `sendCurSpeed(curSpeed)` is added to the transition between those states.

If the message is located in the prechart of the sequence diagram, the event is always a trigger, whether the message is sent or received. The reason for this is that the prechart is observed behavior and not executed behavior. In the example, the sent message `getCurSpeed()` is a trigger of the

corresponding statechart transition between the states `InitiategetCurSpeed` and `SentgetCurSpeed`.

Intra-component method calls are realized as entry actions in one state. Accordingly for the method `calcMaxSpeed(...)` a state `IncalcMaxSpeed` is generated with the method call as its entry action.

To realize correct behavior for the main chart semantics, coordination messages are employed as proposed in [4]. To give notice to all participating components that the prechart of the LSC is satisfied, a message is sent to the corresponding components. As soon as those receive such a message, they change their state for being able to react on or execute the ensured behavior of the main chart. In the example statechart (top orthogonal state), the AdaptiveControl first has to receive `coordadaptiveControlgetCurSpeed()` before it starts with the execution of the main chart. Accordingly, coordination messages have to be inserted to notify all participating components that the execution of the main chart has finished (message `overaccelerationControl()`).

This automatic synthesis of the components' communication behavior facilitates analysis of communication based component and system properties at an early stage of development. It also enables automatic code generation. For this, the intra-component method calls have to be extended with internal controller behavior.

6. RELATED WORK

Our sequence diagrams are based on LSCs [5], using a synthesis scheme as presented later by Harel, Kugler and Pnueli [4]. The main difference of our subset of LSCs, besides the UML syntax, is that they only employ hot semantics and only describe conditions within alternative and loop frames. In addition to [4], we are able to synthesize these alternative and loop frames. The original work on LSCs relies on a simple object system to describe the system architecture, whereas we consider a component-based software architecture. Our synthesis is designed to route coordination messages (see section 5) only over existing connectors between the components instead of allowing arbitrary communication between the components. This becomes important when deploying the components on a distributed platform.

The SceBaSy approach uses time-annotated sequence diagrams to synthesize reusable, state-based coordination patterns [3]. The synthesis focuses on calculating admissible time bounds for real-time statecharts, but in turn requires the sequence diagrams to be deterministic. This requires a detailed behavior description and, therefore, the approach is not suitable for early development phases. SceBaSy and its real-time analysis could, however, still be integrated in a later development stage. To integrate SceBaSy, it should be investigated in the future how to perform time-analysis to find admissible alternatives in early, non-deterministic communication behavior descriptions. Furthermore, the SceBaSy synthesis employs rules for optimizing the resulting statecharts and it is to be investigated how to employ such rules in our synthesis scheme.

7. CONCLUSION AND FUTURE WORK

We presented a development process for automotive software systems with special emphasis on formal specification of communication behavior. We proposed a sequence diagram dialect that facilitates early analysis of communication

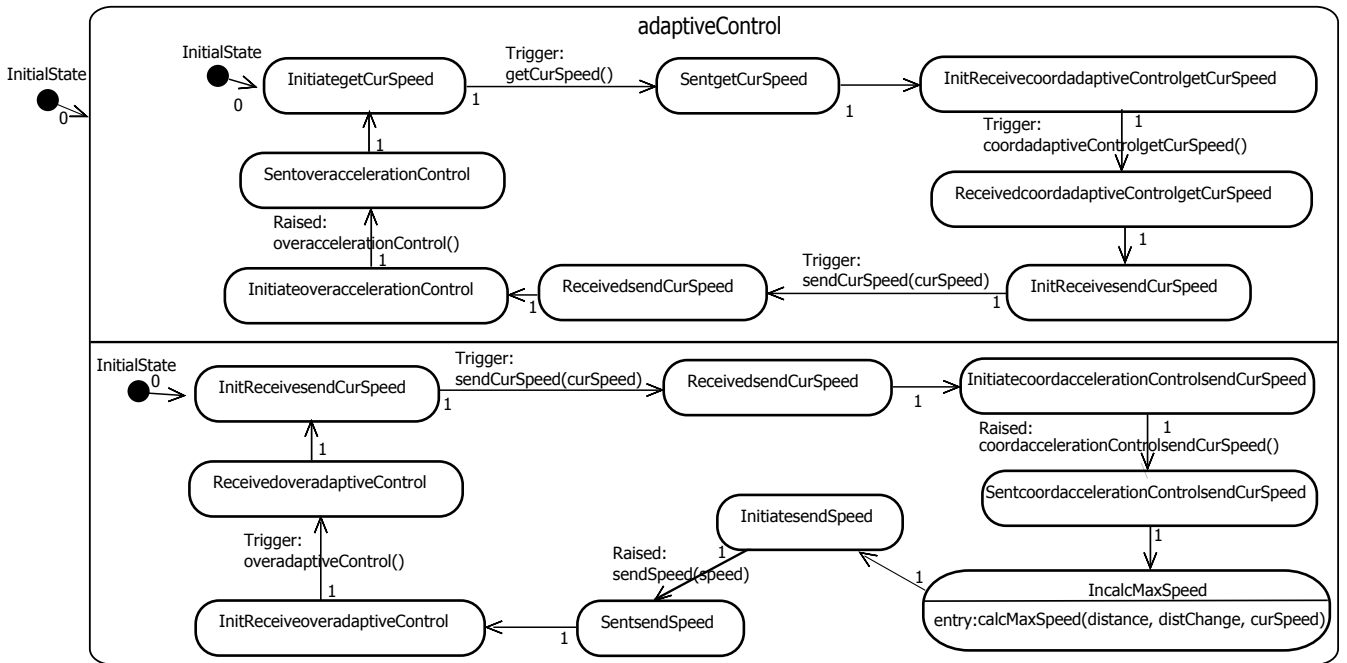


Figure 5: Synthesised statechart for the adaptiveControl component

behavior and automatic synthesis to communication based component behavior by means of statecharts. We realized tool support for this development process and proposed concepts using the Fujaba4Eclipse platform.

Future work embraces scalability analysis of the synthesized statecharts. We suppose that, as the approach uses parallel states, it provides better scalability for statecharts than synthesizing one non-parallel statechart for each component. Accordingly, synthesized statecharts should be optimized to eliminate superfluous states.

Also important for future work are consistency checks on LSCs, for example showing deadlock freedom or discovering contradictions between several LSCs. This is important, because a high modularity in the LSC specification can lead to many inconsistency that are not obvious to see.

It is already possible to annotate time constraints in our LSCs. The synthesis approach needs to be extended with timing analysis as the specified systems act in real-time environments. One possibility could be to integrate SceBaSy for use in later design phases of the overall development process while using our approach in the requirement specification. Controller behavior has to be integrated for example using MATLAB Simulink models. Another extension could be the inclusion of a deployment model (cf. AUTOSAR) in order to specify which software components shall be realized on which hardware components. Finally, an AUTOSAR conform code generation should be developed to support the implementation phase of the automotive software system.

8. REFERENCES

[1] S. Burmester, H. Giese, S. Henkler, M. Hirsch, M. Tichy, A. Gambuzza, E. MÜch, and H. Vöcking. Tool support for developing advanced mechatronic systems: Integrating the fujaba real-time tool suite

with camel-view. In *Proc. of the 29th International Conference on Software Engineering (ICSE)*, Minneapolis, Minnesota, USA, pages 801–804. IEEE Computer Society Press, May 2007.

[2] S. Burmester, H. Giese, and W. Schäfer. Code generation for hard real-time systems from real-time statecharts. Technical Report tr-ri-03-244, University of Paderborn, Paderborn, Germany, October 2003.

[3] H. Giese, S. Henkler, M. Hirsch, and F. Klein. Nobody’s perfect: interactive synthesis from parametrized real-time scenarios. In *SCESM ’06: Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 67–74, New York, NY, USA, 2006. ACM.

[4] D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling*, pages 309–324, Berlin/Heidelberg, Germany, 2005. Springer-Verlag.

[5] D. Harel and R. Marelly. *Come, Let’s Play: Scenario-Based Programming Using LSC’s and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[6] A. V. Lamsweerde. Goal-oriented requirements engineering: A guided tour. *RE ’01: Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, page 249, 2001.