

Interface Verification Using Executable Reference Models: An Application in the Automotive Infotainment^{*}

Christian Drabek¹, Thomas Pramsöhler², Marc Zeller¹, and Gereon Weiss¹

¹ Institute for Embedded Systems and Communication Technologies ESK
Hansastr. 32, 80686 München, Germany

{christian.drabek, marc.zeller, gereon.weiss}@esk.fraunhofer.de

² BMW Forschung und Technik GmbH
Hanauer Str. 46, 80992 München, Germany
thomas.pramsöhler@bmw.de

Abstract. Modern in-vehicle infotainment systems comprise highly interactive software components. The verification of the interfaces of such components poses a major challenge for developers. In this work, we present an approach for model-based verification of distributed infotainment components. We define a layered reference model which specifies the interaction between two components at syntactical and behavioral level. The layers abstract from the used middleware so developers may focus on the components' actual interface behavior. Additionally, we define a model execution framework which enables the reuse of the reference model for verification of interface implementations. We demonstrate the applicability of the approach using an industrial case study. Our approach aims at reducing errors in the communication behavior and increasing the overall product quality.

Keywords: automotive, infotainment, interface verification, model execution

1 Introduction

The growing amount and the increasing interdependency of functions in modern automobiles [6, 21] pose a major challenge for the software development. One of the most rapidly evolving in-vehicle domains is the infotainment (see Figure 1). Up to now this has led to various infotainment functionalities with many interdependencies on different interconnected Electronic Control Units (ECUs). Especially, the ongoing trend of integrating external services into the in-vehicle infotainment increases the complexity of the interfaces of related ECUs. Such services may come from connected consumer devices (e.g. smart phones) or from the Internet (e.g. cloud services). Moreover, there is a movement in the automotive industry towards the use of multi-vendor platforms (such as *GENIVI* [7]),

^{*} The project was funded by the Bavarian Ministry of Economic Affairs, Infrastructure, Transport and Technology

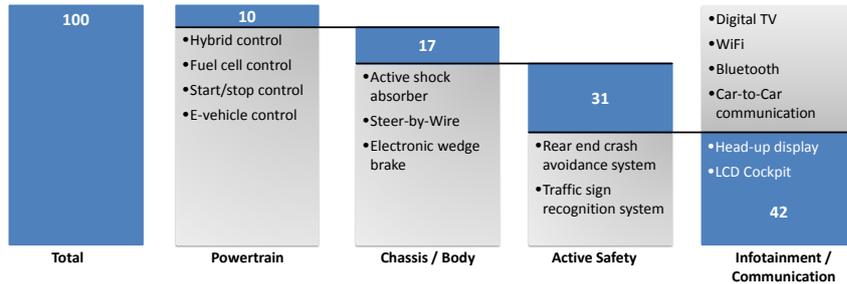


Fig. 1: Infotainment and communication as innovation driver in today's cars. [11]

where services of different suppliers and the OEM are integrated in a single hardware platform using an *IPC bus* for the message exchange. For such integration scenarios, the verification of software components and their interfaces is needed to assure compatibility and enable a well-defined user experience and premium-class quality. This can be difficult when third-party services are provided as black-boxes, where only their communication can be monitored.

Within the automotive application development, model-based approaches have been applied successfully over time [3]. Model-based techniques used during the design and integration phase of new infotainment applications play also a major role in the process of validation and verification [19]. In the infotainment domain the verification of application interfaces is complex, due to the interactive and fast evolving nature of this domain. Traditionally constituting a not safety critical domain, it comprises very diverse and multifaceted interactions, reaching from rich media information exchange to single control interactions.

We identified the following items as most defining challenges for the verification of infotainment interfaces with respect to their communication:

1. **Abstraction:** Middlewares define their own protocols and interface description languages. The modeling approach should abstract from these technical details so developers may focus on the components' actual interface behavior.
2. **Parallel interactions:** Manifold software-based features are assembled and executed in parallel with interfaces partially interdependent. The many potential system states and interactions are very difficult to capture.
3. **Synchronization (Initialization):** The parallel interactions make it hard to identify the actual communication state. Anyway, the verification mechanism needs to be able to assume this state for carrying out the verification.
4. **Timeouts:** Infotainment is not a safety-critical domain, but the meeting of timing requirements is essential for orchestration of interactive components.
5. **Error detection:** The verification should reliably detect an abnormal operation and encircle the actual fault leading to the erroneous functionality.
6. **Executable interface specification:** Especially with complex infotainment systems manual verification is a tedious and error-prone task. Automatic verification requires input for the interpretation by a machine. For model-based specifications this can be achieved with executable semantics.

Within this paper, we focus on an approach for model-based verification of the communication behavior of distributed infotainment components considering the above characteristics. Special emphasis is put on the design and reuse of so-called reference models from the specification phase for the verification. Therefore, a concept for the verification of infotainment components' interfaces by using executable reference models is presented. The main goals are to reduce the verification costs while preserving or even increasing the product quality.

The contribution of our work is twofold. First, we define a layered reference model which is used to specify the interaction between components at syntactical and behavioral level. Thereby we take into account that interfaces are usually defined at syntactical level in an existing proprietary language. We seamlessly integrate those middleware dependent specifications in our modeling approach. In addition, we define a framework for executing such reference models. This enables us to reuse the models defined for specification in a verification process.

The remainder of this paper is organized as follows: In Section 2, we give a brief overview of related work. Afterwards, Section 3 outlines our approach for the model-based verification of infotainment interfaces using reference models. In Section 4, we present the software architecture and execution semantics used to enable the verification of component interfaces in the infotainment domain. In Section 5 our approach is demonstrated using an industrial case study. We conclude the paper in Section 6.

2 Related Work

Model-based development of application interfaces is widely applied in the automotive domain. For instance, *MATLAB/Simulink* [23] are generally utilized solutions for modeling and testing embedded systems. However the main focus of this tool chain is the development of continuous systems or software functions. Instead one major challenge in the automotive infotainment domain is the event-driven, state-based characteristic of the interactive systems. In the context of infotainment systems, models today are commonly created on the basis of UML [17]. There are already frameworks available complementing UML case tools with interfaces to a physical network bus, e.g. *MODENA* [12] for the *MOST (Media Oriented Systems Transport) bus* [15], which generally also enables the testing of communication interfaces of infotainment components [14]. However, for upcoming multi-vendor platforms (e.g. GENIVI) using novel communication mechanisms no such framework is available today.

Infotainment components are usually implemented by different parties based on a specification given by the car manufacturer. Specifications are written in natural language and enriched with software models. In today's automotive software engineering, the specification models are often only used as visual representation of specific aspects. We aim at maximizing the automation of verification processes using these specification models. Current verification methods rely on sequence-based tests [2]. However, by specifying single test sequences the view on the complex interactions and the system states of the components is very lim-

ited. Other approaches (e.g. [1]) include the manual design of test models, which are used for automated generation of test cases. A major drawback of all these approaches is the significant overhead for modeling the complex (infotainment) test models or numerous test cases during the development. Moreover, testing usually involves an oracle [22] to determine the expected outcome of test cases. If created manually, the task grows tedious with a large number of tests.

An automated test oracle can be provided by execution of the specification models. Further, the possibility to simulate and observe the reactions of a model to a given sequence of events helps to understand the specified mechanisms. The frameworks *Pópulo* [5] and *Moliz* [13] are two academic examples for UML model execution. Both use class diagrams to declare classes and operations. The behavior of latter can be defined with activities. *Pópulo* uses the UML action semantics and stereotypes in the UML activity diagram. *Moliz* is based on *Semantics of a Foundational Subset for Executable UML Models (fUML)* [18], a subset of UML with execution semantics. *UML Model Debugger (UMD)* [4] enables execution of activity and state diagrams, but uses Java code to specify actions. Similar the case tool *Rational Rhapsody* [10] allows the user to enter code for actions but uses its own semantics for execution of state diagrams [9]. A different approach is to design a model language specifically for execution of state-based behavior. *State Chart XML (SCXML)* [24] is a general-purpose event-based state machine language that is based on semantics of *Harel statecharts* [8]. Harel statecharts were also adopted in UML state machines, but SCXML includes execution semantics that allow running the model, e.g. for a simulation [16].

In this paper, we introduce an applicable approach for the verification of communication interfaces in the automotive infotainment domain. We exploit executable reference models created during the specification phase of infotainment components for the verification of implementations in the integration phase.

3 Model-based Verification of Interfaces

We address the challenges of interface verification in automotive infotainment systems by using layered interface specifications. A so-called layered reference model is manually derived from the requirements. It describes the communication between two components including all involved interfaces and behavioral interactions. Figure 2 shows the different layers of the reference model.

The *interface definition model* describes the artifacts (e.g., broadcasts, methods, types) of the involved interfaces on a syntactical level. This layer uses the proprietary interface description language usually provided by a middleware technology. By changing this layer, the model can be adapted to the utilized target middleware. Figure 3a shows an example interface definition model for automotive infotainment components.

The *event definition model* bridges the gap between the middleware dependent interface definition model and the middleware independent behavioral model. The modeler specifies the invocation of an interface artifact using an *event definition* and refines this event with a constraint. A constraint is defined by ba-

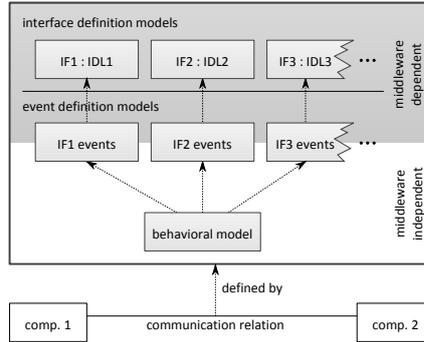


Fig. 2: The layered reference model describing a communication relation.

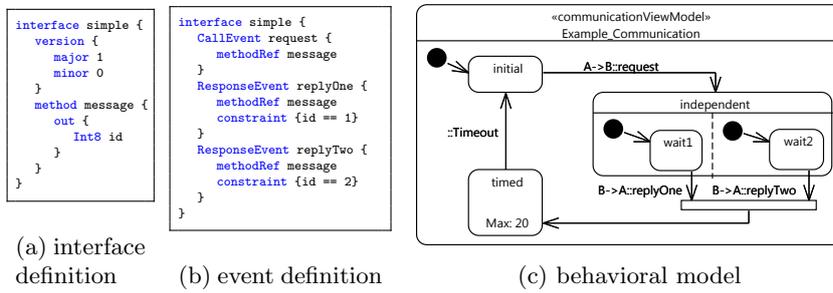


Fig. 3: Exemplary reference model.

sic arithmetic and logical expressions using the parameters which correspond to the interface artifact. The specified events are used in the behavioral model as triggers for transitions. Therefore, only events needed in the behavioral models need to be defined. An example of our textual language for this task is shown in Figure 3b. It may be adapted to other target interface definition models.

In order to model the ordering of events in a communication relation between two components we use stereotyped *UML state machines* [17]. State machines are currently widely applied for behavioral modeling in the automotive domain. We do not use all concepts provided by the formalism but restrict the expressive power with an UML Profile. Figure 3c shows an exemplary behavioral model for the communication between two components. A state in our model is not the status of single components; rather it represents their common communication state. A state can be annotated with a timeout event which defines the maximum duration the state may be active. When the state has been active for the specified amount of time, the corresponding timeout event is emitted. This allows specifying timing requirements in the model (i.e. deadlines and cyclic events).

In UML state machines transitions are defined with trigger and guard. In our experience with real models, the guard conditions are a powerful concept but might become complex pieces of program code. Therefore, we allow the guard

condition to be specified using the constraints of the event model only. Triggers may reference an event of the event model or a timeout of the behavioral model. The sender and the receiver of the event are also annotated on the transition.

The limited number of elements without nested code snippets prevents the use of ambiguous concepts and facilitates a precise description of the communication relationships. The approach is precise enough for direct code generation of an executable state chart for a specific middleware. We could already show the applicability with a subset of this model for adapter-code generation in [20]. In this work, we extended the approach with timing, and parallel and partially interdependent communication. Parallel behavior is modeled by states with parallel regions. A join element can be used to coordinate the exit of those regions.

4 Execution Framework for Reference Models

In this section, we introduce our execution framework for reference models used for the verification of interfaces in automotive infotainment systems. The framework monitors the component interaction and reports failures (see Figure 4).

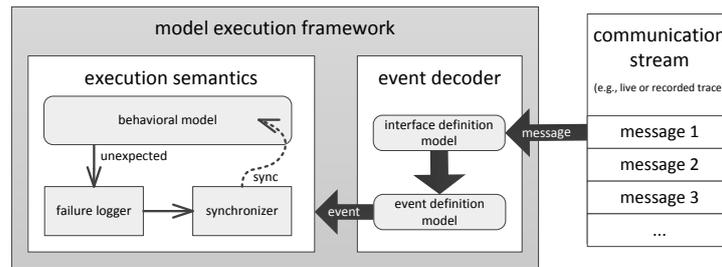


Fig. 4: Execution framework for reference models.

With the information from the interface and event models, the *event decoder* maps incoming messages to events and discards messages not relevant for the reference model. The behavioral model gets triggered by the resulting event stream. Every event which is not consumed by the model in a specific state will be logged. If such a failure in the communication stream is detected, the current state of the communication is unclear and the active state in the behavioral model has to be determined by the *Synchronizer* using the next messages.

For executing the reference models, we use open and standardized semantics. fUML defines execution semantics for a subset of UML, but this subset does not include state machines. Therefore, we use SCXML, which provides well-defined semantics for executable state machines. SCXML is not based on UML state machines but it is sufficiently similar. Most of the modeling elements and concepts in our behavioral models can be mapped straight forward to SCXML: initial nodes, transitions, states with regions for nesting as well as parallel behavior.

Special treatment is only needed for the join element, which is not available in SCXML. To compensate for this discrepancy in the mapping, we redirect transitions targeting the join to a new final state in each region. The join’s outgoing transition is connected to the parallel state as source, and triggers when all final states have been entered.

In SCXML our error semantics can be modeled by containing the original state machine in a state, where a single transition leads to the error state. This transition’s trigger is set to match any event. As transitions of inner states precede those of outer states in SCXML, the error transition fires only if the event is not consumed by the original state machine. We implemented a straightforward type of Synchronizer with transitions for all unique events to their corresponding target states. A more detailed description of the synchronization mechanism is beyond the scope of this paper. This mapping grants execution, error checking, and synchronization semantics to the proposed modeling approach enabling detection of the following failures:

- missing messages,
- additional messages,
- malformed messages, and
- timing violations

Thereby, our reference models together with the execution framework can be used to verify interfaces of infotainment components.

5 Application in an Automotive Use Case

In this section we show the application of our approach using an automotive example. An automotive infotainment ECU implements many functions which communicate with other ECU’s using the in-car network. This includes reading and setting values. This communication with the hardware services is often realized using a gateway component with respective software proxies. In the following, we introduce a fictive parking assistant (*ParkA*) software proxy. This service is used for producing a top-view of the car showing the surrounding obstacles. The ParkA component provides a reliable link in order to obtain up-to-date sensor values. In this case the reference model describes the inter-process communication inside the infotainment ECU between a Client (e.g. a surround view) and the ParkA-service.

Figure 5a shows the syntactical interface of the ParkA component. The interface provides methods for startup, shutdown and resolution settings. The ParkA service sends the sensor values using the broadcast `sensorValues`. Figure 5b shows the respective event definitions for the ParkA interface. In this model we specify concrete occurrences of broadcasts, method calls or responses.

In addition to the syntactical interface and events we define the protocol for a specification-conformant interaction with a ParkA service. Figure 5c shows the behavioral model. To initiate communication, the client has to invoke the `startUp-Method`. If successful (`startUp_response_OK`), the client is connected

```

interface ParkA{
  method startUp{
    in{Resolution res}
    out{AM_Error returnValue} }
  broadcast sensorValues {
    out{
      Int32[] front_lmr
      Int32[] rear_lmr
      Resolution res } }
  enumeration Resolution{ cm mm }
  enumeration AM_Error{ OK ERROR }
  ... }

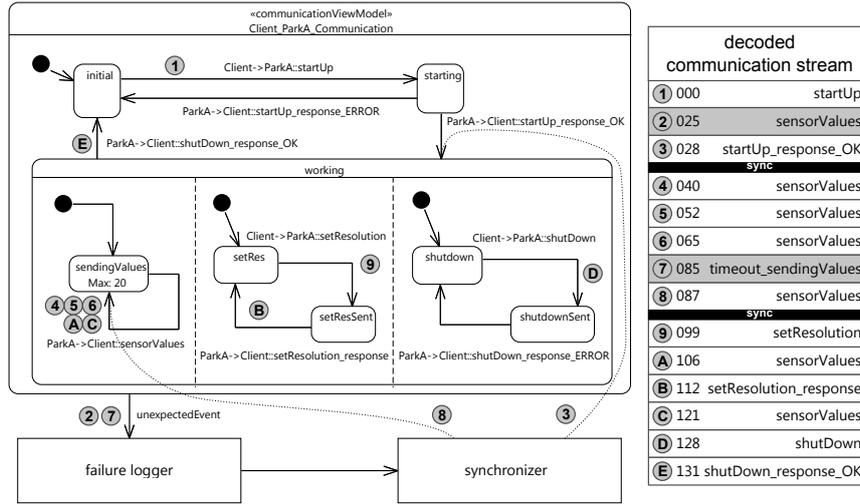
```

```

Interface ParkA {
  CallEvent startUp {
    methodRef startUp }
  ResponseEvent startUp_response_OK {
    methodRef startUp
    constraint {returnValue == OK} }
  ResponseEvent startUp_response_ERROR {
    methodRef startUp
    constraint {returnValue != OK} }
  BroadcastEvent sensorValues {
    methodRef sensorValues }
  ... }

```

(a) ParkA interface definition excerpt. (b) ParkA event definition excerpt.



(c) ParkA behavioral model and example trace.

Fig. 5: Reference model for the communication between the ParkA component and a client. Additionally the decoded input communication stream is shown.

and the service starts sending cyclic sensor values using the `sensorValues`-broadcast. The time between two of these broadcasts should not be longer than 20ms (`Max: 20`). At any time in the `working`-state the client may change the resolution of the sensor values using the `setResolution`-method or shut down the service using the `shutDown`-method. This is described using a parallel region.

We show an example for a communication trace in Figure 5c. Each line defines an event number, the timestamp of the event and the already decoded message as string. In this example we annotate the transitions of the behavioral model with the order in which the trace is executed. The ParkA service starts sending sensor values immediately after the `startUp` method call. This is a deviation from the specified behavior and causes the behavior model to trigger the `unexpectedEvent`-transition. The failure gets logged and the next event (`sensorValues`) is used for synchronization. The next failure is detected at message number 7. The timeout in state `sendingValues` elapses and fires the `timeout_sendingValues`-event. This event does not trigger any transition in the

active state and will be handled by the `unexpectedEvent`-transition. Synchronization is done after receiving the next `sensorValues`-broadcast. The trace completes without further failures. Other unique events have transitions from the Synchronizer into the behavior model, too, which are not shown for clarity.

The case-study illustrates how we address the challenges for the verification of automotive infotainment interfaces introduced in Section 1:

1. The event model abstracts from technical details, thus allowing for a clear behavioral model.
2. All potential system states of parallel and partially interdependent components can be captured by using parallel regions and multiple interfaces in the reference model.
3. Using unique messages the presented reference model can be (re-)synchronized to the tested components for initialization or after the model detected a message not conforming to the modeled behavior.
4. The synchronization mechanism allows to detect timeouts during the verification (e.g. when a message is not delivered in time) by treating timeouts of states like other events.
5. Reliable error detection is provided by the logged failures, which can later be evaluated by the developer, in order to verify, if the deviant behavior of the implemented interface is identified as a definite error or not.
6. The presented execution framework enables running the reference models in parallel to the components for verification.

Thereby, our approach enables the verification of infotainment components' interfaces by detecting failures within the communication behavior using executable reference models.

6 Conclusions & Future Work

In this work, we outlined the major characteristics and challenges for the verification of communication interfaces in in-vehicle infotainment systems. Moreover, we have introduced an approach for the verification of interfaces which addresses these challenges by using reference models created during the specification phase. We showed the applicability of our approach by an example of a parking assistant proxy component. Our approach allows exploiting the effort once spent for a model-based specification in the integration phase, by verifying the quality of the implemented components.

Future emphasis will be laid on the enhancement of the proposed methodology for complex data-depending models. Furthermore, reusing the reference models for so-called restbus simulation with real ECUs will be investigated in future work.

References

1. Benz, S.: Combining test case generation for component and integration testing. In: Proc. of the 3rd Int. Workshop on Advances in model-based testing (A-MOST). pp. 23–33 (2007)

2. Braun, A., Bringmann, O., Rostenstiel, W.: Testing with Virtual Prototypes. *Elektronik automotive Special Issue MOST*, 49–51 (2011)
3. Broy, M., Chakraborty, S., Ramesh, S., Satpathy, M., Resmerita, S., Pree, W.: Cross-layer analysis, testing and verification of automotive control software. In: *Proc. of the Int. Conf. on Embedded Software (EMSOFT)*. pp. 263–272 (2011)
4. Dotan, D., Kirshin, A.: Debugging and testing behavioral UML models. In: *Proc. to the 22nd ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications Companion*. pp. 838–839 (2007)
5. Fuentes, L., Manrique, J., Sánchez, P.: Pópulo: a tool for debugging UML models. In: *Proc. of the 30th Int. Conf. on Software Engineering (ICSE)*. pp. 955–956 (2008)
6. Fuerst, S.: Challenges in the Design of Automotive Software. In: *Proc. of Design, Automation, and Test in Europe (DATE)* (2010)
7. GENIVI Alliance: The GENIVI Alliance, <http://www.genivi.org/>
8. Harel, D.: Statecharts: A visual formalism for complex systems. *Science of computer programming* 8(3), 231–274 (1987)
9. Harel, D., Kugler, H.: The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). In: *Integration of Software Specification Techniques for Applications in Engineering*, pp. 325–354. Springer (2004)
10. IBM: Rational Rhapsody, <http://www.ibm.com/>
11. Krust, M.: Weg vom Blech! *Automobilwoche* 23, 10 (2010)
12. Mattner, B.: MODENA - The specification and test tool for infotainment components, <http://www.berner-mattner.com/>
13. Mayerhofer, T.: Testing and debugging UML models based on fUML. In: *Proc. of the 34th Int. Conf. on Software Engineering (ICSE)*. pp. 1579–1582 (2012)
14. Meyer, J., Werner, A., Bichler, L.: Model-based testing for infotainment systems. *ATZelextronik worldwide* 1(2), 20–22 (2006)
15. MOST Cooperation: Media Oriented Systems Transport (MOST), <http://www.mostcooperation.com/>
16. Moura, R.S., Guedes, L.A.: Simulation of industrial applications using the execution environment SCXML. In: *Proc. of the 5th IEEE Int. Conf. on Industrial Informatics*. pp. 255–260 (2007)
17. Object Management Group (OMG): Unified Modeling Language Specification Ver. 2.0 (2005), <http://www.omg.org>
18. Object Management Group (OMG): Semantics of a Foundational Subset for Executable UML Models (FUML) (2011), <http://www.omg.org/spec/FUML/1.0/>
19. Pramsöhler, T., Kafkas, M., Paulic, A., Zeller, M., Baumgarten, U.: Control Flow Analysis of Automotive Software Components Using Model-Based Specifications of Dynamic Behavior. In: *Model-Based Design and In-Vehicle Software*. SAE World Congress (2013)
20. Pramsöhler, T., Schenk, S., Baumgarten, U.: Towards an optimized software architecture for component adaptation at middleware level. *Lecture Notes in Computer Science*, vol. 7957, pp. 266–281. Springer (2013)
21. Pretschner, A., Broy, M., Kruger, I., Stauner, T.: Software Engineering for Automotive Systems: A Roadmap. In: *Future of Software Engineering*. pp. 55–71 (2007)
22. Staats, M., Whalen, M., Heimdahl, M.: Programs, Tests, and Oracles: The Foundations of Testing Revisited. In: *Proc. of the 33rd Int. Conf. on Software Engineering (ICSE)*. pp. 391–400 (2011)
23. The MathWorksTM Inc.: MATLAB[®] & Simulink[®], <http://www.mathworks.com/>
24. World Wide Web Consortium (W3C): State Chart XML (SCXML): State Machine Notation for Control Abstraction, <http://www.w3.org/TR/scxml/>