Reducing the Verification Effort for Interfaces of Automotive Infotainment Software

2015-01-0166

Published 04/14/2015

Christian Drabek, Annette Paulic, and Gereon Weiss
Fraunhofer ESK

CITATION: Drabek, C., Paulic, A., and Weiss, G., "Reducing the Verification Effort for Interfaces of Automotive Infotainment Software," SAE Technical Paper 2015-01-0166, 2015, doi:10.4271/2015-01-0166.

Abstract

Car infotainment systems feature an increasing number of functions to keep pace with consumer needs. The GENIVI Alliance aims to facilitate this evolution of infotainment systems by developing a common baseline where services of different suppliers can easily be integrated on a single hardware platform. Since the huge number of services creates more dependencies and interactions, more effort is required to ensure the same level of quality. We present a novel approach and effective tooling to reduce the effort for the interface verification of in-vehicle software components. Our models create different views of the system. Consistency checks and automated transformations between the views reduce the modeling effort and ensure compatible interactions of distributed software components. Layered reference models separate the description of the structure and the behavior of the services' communication. This simplifies the behavior descriptions and facilitates the usage of different communication technologies, e.g., D-Bus or CAN. Since the reference models are executable specifications, they can be used to verify the communication of the modeled services. This can be tested live or from a trace. In case of required changes to an interface, regression testing can be performed automatically using only the model. We evaluate the benefits and implications of our approach and tool with a case study of an in-vehicle audio function.

Introduction

In-car infotainment systems are a good example for the increasing complexity of software features in networked embedded systems. Generally, common basic architectures are utilized to enable faster development cycles, reuse, and shared development of non-differentiating functionality. For infotainment systems the GENIVI Alliance [15] defines an interoperable infotainment standard which enables the integration of software components from multiple vendors into one platform. Such integration requires that the interoperability and interactions of these components are guaranteed. For this, the definitions of the interfaces and interactions are a main concern. However, today specifications only consider static definitions of the interfaces. The behavior - which often is the most critical part for the integration - is only described in natural language, if it is explicitly defined at all. This hinders the integration of black-box components from different vendors as their interworking and interaction behavior cannot be ensured. Therefore, also the dynamic part of the software components must be specified and verified during the different development stages.

With this paper we present a methodology, specification and tool which automatically verify multi-vendor software components throughout the phases of the development process. Presently, the application is designed for in-vehicle infotainment systems. However, its concepts and methodology can also be adapted to other domains with integrated software architectures, such as AUTOSAR. Our approach provides:

- Methodology for the specification of interface and behavior definitions
- Consistent views on the system for specific concerns
- Automatically derived regression tests
- Verification of implementations utilizing executable specifications

We use the audio functions of an in-vehicle infotainment system to demonstrate the viability of our approach. We can show that its complex interactions can be specified consistently utilizing different views of the system. Furthermore, this executable specification can be used for the verification of audio function implementations. By following our approach, the development of such interconnected embedded software systems can be improved considerably.

The remaining paper is structured as follows. In the next section, we discuss related work to our approach. Afterwards we present the methodology and concepts for the specification and modeling of software components. Then we introduce the verification which builds upon the presented concepts and executable specifications. Finally we discuss the benefits and limitations of our approach and conclude the paper.

Related Work

Model-based development is widely applied in the automotive domain. For instance, MATLAB/Simulink is utilized for modeling and testing continuous systems or software functions. However, in interactive systems of the automotive infotainment domain, event-driven and state-based characteristics are predominant. In the context of infotainment systems, nowadays models are commonly created on the basis of UML (Unified Modeling Language). State-of-the-art CASE tools (e.g., Enterprise Architect [16] or Rational Rhapsody [17]) enable the creation of models for specification and code-generation. The generated code usually includes target code and test cases. Additional frameworks provide interfaces to certain physical busses, for example, MODENA [18] for the MOST (Media Oriented Systems Transport) [19] bus. Recently the automotive industry also utilizes Eclipse [20] based open source tools such as ARTOP [21] or EATOP [22]. However, there is no framework available today for novel communication mechanisms in upcoming multi-vendor platforms, e.g., for D-Bus [23] used by GENIVI [15]. We present an approach which is close to traditional modeling approaches for infotainment systems but can be easily adapted to new highly integrated platforms and diverse communication mechanisms.

Different suppliers implement infotainment components based on a specification given by the car manufacturer or integrator. The manufacturer commonly provides these specifications in natural language which may additionally be enriched with software models. In today's automotive software engineering, the specification models are often used as a visual representation of specific aspects only. We aim at maximizing the automation of verification processes by using these specification models. Current verification methods for automotive systems rely on sequence-based tests [1]. Considering all the possible interactions, it is hard work to manually create test cases required for sufficient test coverage. Therefore, a common approach is to design distinct test models, which are used for automated generation of test cases (e.g., [2]). In contrast, we reuse the specification models for generating test cases.

Testing usually involves some kind of oracle [3,4] to determine the expected outcome of test cases. If the outcome is predicted manually, the task of creating the oracle becomes extensive with a large number of tests. An automated test oracle can be provided by executing specification models. This provides the possibility to simulate and visually observe the reactions of the model to a given sequence of events and helps to validate the specified mechanisms. The execution of models is a well explored field of science and various tools provide readily available execution environments [5,6,7,8]. With the help of model execution, we use the specification for monitoring executed implementations in order to find deviations. Such a monitor is “a system that observes and analyses the behavior of another system” [4]. Passive testing of the model with a monitor can be seen as a form of runtime verification [9]. Runtime verification checks if a certain run of a system under test (SUT) satisfies or violates a correctness property. It is focused on the detection of deviations and well suited for black box systems, as no details about the inner states of the system under test are needed. Other verification methods like model checking require more details than a black box system can provide, have the problem of state explosion, or do not capture potential differences in the behavior of the model and its implementation [9]. The core of a monitor is an analyzer which is created from the requirements [10]. Different languages can be used to specify an analyzer [10]. For example, Leucker et. al. [9] use linear temporal logic as a high-level language and generate finite state machines as analyzers. Our approach already uses state machines for specification and reuses them as analyzers in our monitor. In this case the specification model is seen as correct reference implementation. The communication of infotainment components is well observable from outside the system; therefore, we keep the verification separate from the target system.

Methodology & Design

In this section the developed modeling methodology is described. It is the basis for our verification approach. The most defining challenges for the verification of infotainment interfaces with respect to their communication are (cf. [11]):

1. **Abstraction**: The interface behavior description abstracts from technical details, but is connected to the middleware interface description.
2. **Parallel interactions**: Independent and dependent communication sequences can be described as needed.
3. **Synchronization (Initialization)**: The verification mechanism must be able to identify and assume the current communication state.
4. **Timeouts**: Timing requirements are captured in the model and checked during verification.
5. **Error detection**: Abnormal operations must be reliably detected and identified.
6. **Executable interface specification**: Only a specification that can be processed by a machine can be used for automated verification.

In this paper we show how these challenges can be solved with minimal additional effort for the specification modeler compared to creating a specification model without any verification support.

Different views are introduced in order to simplify the modeling process. For reducing the modeling effort a UML profile has been created. Additionally, the developed models can be reused by means of model-to-model-transformations. We use open, standardized tools and methods to increase the acceptance and facilitate the use of the presented solution.

Figure 1 depicts an overview of the developed verification framework. The system under test is connected to the verification framework through a communication bus. The behavioral model

describes the expected communication behavior of the SUT. The messages sent to and from the SUT are transferred via the bus and are the input for the behavioral model. The bus messages with their according interfaces are defined by attributes, broadcasts and methods in the Franca IDL (Interface Definition Language) [24]. Franca is used as IDL by the GENIVI consortium. The IDL allows the definition of objects and their methods independent of a programming language. From the defined interfaces, the code to access remote objects can be generated. Before the messages can be processed by the behavioral model they are mapped to events. These events are described in a DSL (Domain Specific Language), in which every event directly references a Franca element. The once defined events are used as triggers of the behavioral model. The behavioral model receives all messages of the SUT as events. If an incoming event cannot be processed by the model, a failure is detected and logged. In this way, the communication behavior of the SUT is verified solely by its bus communication. Therefore, our approach is well suited for black box testing. Another possibility is to verify a trace recorded from the SUT. In this case, the messages contained in this trace are the input for the behavioral model. Besides traces also test cases can be an input for the verification and used for regression testing.

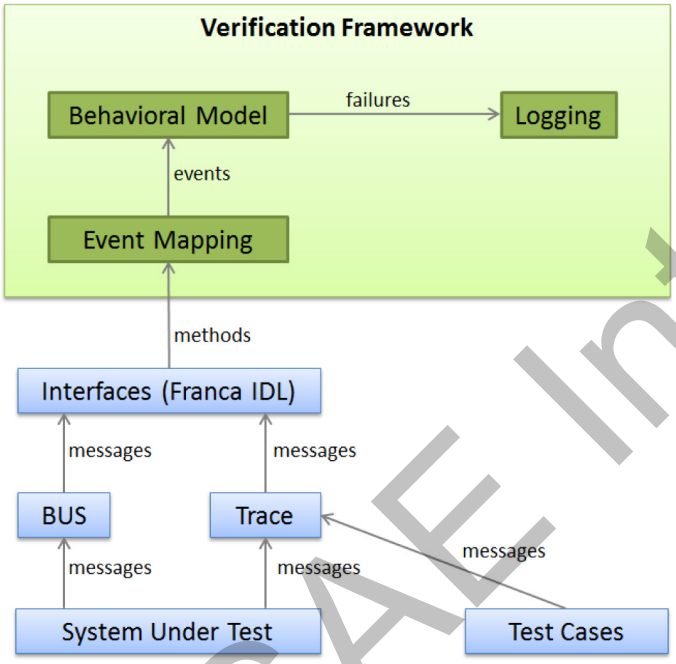

Figure 1. Overview of the verification framework

Description of Interface Behavior

In our approach the communication behavior of components is modeled using different views. As the interface behavior of components has to be verified it is not necessary to consider the internal behavior of components, but only the communication behavior of the components' interfaces. This behavior is defined in the form of messages which are sent to and from the respective component.

Because it is a widely accepted standard for modeling, UML is applied to model the communication behavior. Especially, UML state machines have been selected in this case, because they allow defining different states during the communication of a component. Transitions between these states describe how the communication state is changed. Every transition of a state machine is triggered by a specific event (e.g., a message is sent from one component to another component). The trigger decides which transition is taken. A transition can also contain an optional action, which specifies the response message of the component to the incoming message.

For modeling, we use the LUNA version of the UML tool Papyrus [25], which is integrated in Eclipse [20]. Papyrus adheres closely to the UML specification of the OMG (Object Management Group) [26], and supports the creation and use of UML profiles.

In order to limit the number of elements provided by the general purpose language UML, we have defined a UML profile. This profile only contains the minimum elements required for the models. This reduces the effort for modeling and keeps the models simple and maintainable. The selected subset of modeling elements does not limit the types of systems which can be modeled. If necessary, the behavior of excluded elements can be recreated by combination of the remaining elements. The following elements are used for modeling state machines:

- **State**: A rectangle that describes the state of a communication
- **Initial State**: A circle that indicates the starting point in a state machine
- **Transition**: An arrow that shows the connection between states
- **Region**: Every state can have one or more (parallel) areas which in turn contain further sub states
- **Join**: A bar that combines paths going out of parallel regions
- **Comment**: A text field which may include a comment

Additionally, with our profile, timing information can be added to states. Thereby, modelers can define for how many milliseconds at most a state can be active before the next incoming message is expected. The timeout can also be used to trigger a transition. This can be used to define a minimum time interval in which a state is active and no message is expected to be received.

A Matter of Perspective

A communication between components can be observed from different perspectives. Thus, it is helpful for the modeler to create several models with different views of the same communication. For instance, this may include the view on one service with its reactions to received messages or a different view considering the observation of messages between two services.

From the viewpoint of a certain component, the communication behavior describes which messages are sent to the individual component, and which messages are transmitted in response. Therefore, in this view the trigger (the incoming message) and the action (the outgoing message) of a transition are modeled. This view is called the ComponentView. The modeler uses this view to design the communication behavior of a single component.

From the viewpoint of the communication between components, the communication behavior describes which messages are exchanged between the components. For instance, in this case it is not important which message is seen as request or as reply. These models only

include triggers without actions. This view is called the CommunicationView. The modeler uses this view to design the communication flow between interacting components.

The views describe different concerns and details. In case of the ComponentView, the model contains more detailed information about the complete communication behavior of the regarded component. Whereas in the CommunicationView, the main focus is the interaction between two specific components. The communication behavior of these components with any other component is not the concern in this view, and thus, not part of the model.

For all views the same editor is provided. The only difference is in the UML elements which are available for modeling.

SystemStructure

The SystemStructure models the composition of the system. This contains all components which are involved in the system and the communication relationships between these components. The communication relationships describe which components interact with each other. The SystemStructure is the basis for all other models. For every component in the SystemStructure a ComponentView model can be created. Every communication relationship in the SystemStructure can be defined using a CommunicationView model.

The model in Figure 2 shows an example of a SystemStructure. In this case study, there are three components: the AuxiliaryInput (AuxIn), the AudioManagement (AudioMgmt), and the ConnectionManagement (ConnMgmt). The communication relationships (annotated with the stereotype ≪Communication≫) indicate, that AuxIn exchanges messages with AudioMgmt and ConnMgmt. AuxIn provides the interface PlayerControl, which is accessed by AudioMgmt and ConnMgmt.

For clarity reasons, the example models presented in this paper only show a small excerpt of the actual case study. The models used for testing the presented approach are considerably bigger and contain about 50 states and 60 transitions each.

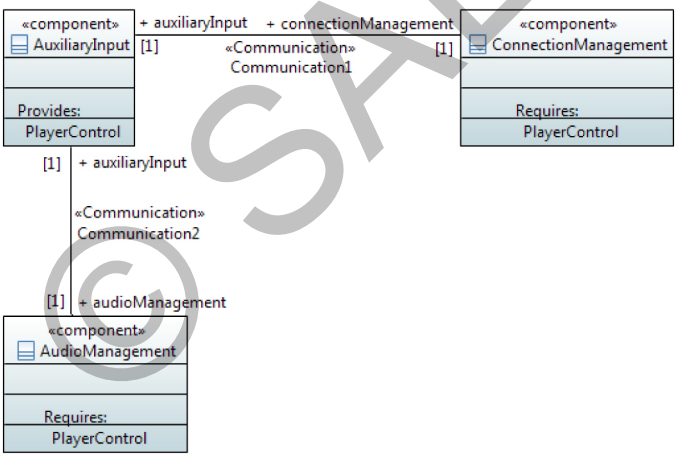

Figure 2. Example of a SystemStructure model

ComponentView

As mentioned before, the ComponentView models the communication behavior from the perspective of a single component. The transitions in this model view contain triggers and actions. The triggers are the messages sent to the component, and the actions denote the reply from the component to incoming messages.

In Figure 5 (see Appendix) the ComponentView model of the component AuxiliaryInput is depicted. After the system has started, the ConnMgmt sends a request to the AuxIn in order to allocate the source (allocate_StartResult). If the allocation worked fine, the AuxIn sends back a response (allocate_Result). After the allocation is finished the source has to be activated by exchanging the messages sourceActivity_StartResult_On and sourceActivity_Result_On between ConnMgmt and AuxIn. Now, the AuxIn is activated and playing music. The AudioMgmt utilizes the interface of the AuxIn to pause or stop the music by sending deckStatus_Set_Pause or deckStatus_Set_Stop. The AuxIn receives the message and replies with its present status.

This view is also used for the simulation of a component. The incoming messages are processed in the model, and the messages defined in the actions of the transitions are sent as a response. If a component is not implemented yet, a simulation of the component can be used for the verification of another component in a so-called restbus simulation. Thereby an early verification of the components is already possible, even though not all implementations of its communication partners are available.

CommunicationView

The CommunicationView describes the communication behavior from the perspective of a communication relationship between two components. All messages sent from one of the two modeled components to the other are modeled as triggers.

Figure 6 (see Appendix) shows the CommunicationView model of the communication between the components AuxiliaryInput and AudioManagement. For instance, the allocation and activation of the AuxIn is not part of this model, because it only describes the communication between AuxIn and AudioMgmt. The CommuncationView model has more states and transitions than the ComponentView model. The reason is that the actions of the transitions in the ComponentView model are triggers of additional transitions in the CommunicationView model.

As there are no actions in this model view, it cannot be used for simulation. However, it can be used for verification, since therefore only a monitor is required. All messages sent between the two components can be processed by the triggers of the model. If an incoming message cannot be processed by the model, an error is detected.

Transformations & Checks

In order to further minimize the effort for modeling, also model-to-model transformations are provided. With these transformations a model view can be transformed to another view automatically. In a ComponentView model the regarded component receives

messages from several other components and responds to them. For every communication relationship between the considered component and another component contained in the ComponentView model, a separate CommunicationView model can be generated automatically. Additionally, for each component contained in a CommunicationView model, a separate ComponentView model can be generated. This reuse of existing models simplifies the modeling process and reduces the effort for the modeler. The automatic transformation moreover ensures the compatibility between the different model views.

In addition, several modeling constraints have been implemented and the compliance with these constraints is checked continuously during modeling. This is necessary in order to enable an automatic transformation between the model views. An example for such a modeling constraint is that every CommunicationView model has to be assigned to a communication relationship in the SystemStructure. The constraints are implemented using Eclipse's model validation framework. The framework allows including additional constraints and checks with little overhead. For example, if lockable resources are managed using the monitored communication, checks for known deadlock conditions could be created.

We also applied further consistency checks for the created models. For example, the modeler can analyze several CommunicationView models to check their consistency and detect contradictory communication sequences. The result of the analysis is displayed using a Labeled Transition System (LTS), and the contradictory parts of the CommunicationView models are marked. This enables the modeler to find inconsistencies between different models immediately, which is nearly impossible to do manually without tool support [12].

Input Classification

The interface behavior description should not contain technical details that are only relevant for a certain middleware or a specific bus technology. This allows the modeler to focus on the description of the actual behavior. He can create the behavior specification independently from the used middleware or bus. The specification can be created before the decision for a certain communication media is made and also be reused if the media is changed. This challenge can be resolved with an additional layer of abstraction.

Franca IDL is used to define software interfaces [24]. In a Franca file the interface of a component is defined. It contains all attributes and methods the interface provides, along with their parameters. It is a description of all the available messages.

The events that are used as triggers and actions in the ComponentView and CommunicationView models are defined in a separate file for each component interface. The events are equivalence classes for the messages. All messages that are mapped to the same equivalence class will have the same impact on the interface behavior. Equivalence classes have already been successfully used to reduce the complexity of learning state machines [13]. We see the same potential for the manual creation. For an event, different child events can be created. In a child event, parameter values or ranges of values can be set. For example, the events sourceActivity_StartResult_On and sourceActivity_StartResult_Off are child events of the base event sourceActivity_StartResult. In the first child event, the parameter sourceActivity is set to the value “On”; in the second event, it is set to “Off”. The advantage of defining these child events in a separate file is that the values of the parameters do not have to be specified in the models. Furthermore, the defined child events can be used in several models, but are only defined once. This also facilitates the consistency checks between the different views, as the events are easier to match than regions in the parameter space.

In order to formally describe the child messages we defined an event definition DSL. An example of the specification of the child messages sourceActivity_StartResult_On and sourceActivity_StartResult_Off using the DSL is depicted in Figure 3. The given constraint makes sure that the child message is only triggered if the constraint is fulfilled. The methodRef relates to an element defined in a Franca file.

```
CallEvent sourceActivity_StartResult {
    methodRef sourceActivity_StartResult
    children{
        CallEvent sourceActivity_StartResult_On{
            constraint{
                sourceActivity == ON
            }
        }
        CallEvent sourceActivity_StartResult_Off{
            constraint{
                sourceActivity == OFF
            }
        }
    }
}
```

Figure 3. Example of message definition

Verification using Executable Specification

In this paper, we want to show how we can reduce the effort for the verification of automotive infotainment software interfaces. The previous sections focused on how the specification can be created with a minimum effort. In this chapter we will show how this specification is used for verification. No further effort by the modeler is required. Our main goal is to compare a run of a system to its specification and find deviations in the communication. Nevertheless, we will also show how our approach can be used to reduce the effort for test case driven testing and regression testing.

The specification provides all information needed to verify the communication of its implementations [11]. Obviously, if the specification was used as implementation it would behave as specified. However, usually there are several steps involved to get from a specification to an implementation. Each step can introduce unwanted deviations. The specification only contains details about the communication, not about internal details, e.g., how certain values are to be retrieved or calculated. So even with code-generation methods in place, some gaps still have to be filled by other means and deviations can occur. However, the specification tells exactly when and what information is expected to be exchanged between the components. For verification we execute the specification in a passive

mode. The executed specification will not generate its own output, but it can monitor the output of an implementation. If the observed output is not expected from the specification, a deviation has been found. With this monitor we can detect the following failures:

- missing messages
- additional messages
- malformed messages
- timing violations

Monitor Maxims

A monitor should adhere to the two maxims impartiality and anticipation to be neither premature nor overcautious during runtime verification [9]. Impartiality requires the monitor to only evaluate to true or false if further events cannot change this result anymore and needs at least three different truth values: true, false and inconclusive. In normal operation our verification mechanism reports deviations from the specification when they are observed. A deviation is an event that was not specified, i.e., the observed event was not expected and no further events can change this observation. As deviating events may occur even with the last message of the system, true can only be reported if no error has been found when the verification ends. Anticipation requires the monitor to report true or false as soon as no further events can change this result. When observing deviations from the specification this coincides with the moment the deviation can be observed. Deviations can only be observed if the communication that just happened was not specified, i.e., the observed event, including timeout events, was not expected. Predicting the deviation before it is observed would require knowledge of an internal error of one of the components in the system which is not available as verification aims to find them.

Monitors that follow these two maxims work on a prefix of an execution [9], i.e., from the start of a component or system up to now. If a failure is detected, the monitor reports this failure. Monitors should follow an additional maxim to be usable efficiently: resumption. Resumption requires that the monitor can ignore failures in a trace that appeared before any given event and starts to reliably report new failures as soon as possible. This is needed to resume the operation of the monitor after a failure was detected. Resumption is especially useful in offline monitoring and batch processing of traces. For example traces of a car recorded during an extended period of use. When analyzing this trace, you want to find all the deviations from the specification, not just the first. This maxim of resumption coincides with the challenge of synchronization. A monitor able to synchronize to the state of a system is always resumable. After a failure it can be restarted and will assume the state of the system and resume operation. A monitor that adheres to resumption can be synchronized by declaring the prefix before the first message as faulty - it is missing.

Implementation Overview

A monitor is composed of an observer and an analyzer [10]. The observer detects events and the analyzer checks them. In our implementation the observer reads messages from a bus trace and maps them to a queue of events. For example, we use the available D-Bus bindings for Java [23] to receive messages from the D-Bus of the tested system. The mapping is specified with the interface and event definition languages. It is generated automatically and filters the reported events to the currently verified set. The mapping also observes the analyzer to detect if a state remained active for a longer period than specified in the state's max-property. It will then generate a timeout event. The timeout event is treated like any event for a communication message. The behavioral model is used as the analyzer. If no transition that is reachable from an active state has a trigger for an observed event, a deviation has been found and is reported. For initialization and after a failure, the monitor can resume operation with the help of a synchronization module. The synchronization module uses the events from the observer to identify the current state of communication and changes the analyzer accordingly.

The analyzer of our monitor needs means to execute the specification given as UML state machines to use it as a reference. UML itself provides no execution semantics. The Semantics of a Foundational Subset for Executable UML Models (FUML) is an addition to UML that gives execution semantics to certain diagrams, but not for state machines. Therefore, we use State Chart XML (SCXML), which provides well-defined semantics for executable state machines. SCXML is not based on UML state machines, but it is sufficiently similar to be used as execution semantics for the selected subset of elements [11]. The executed specification is fed with the events from the mapping. For each of the events it can decide if the event is acceptable and also change its active state to be prepared for the next event. If the event cannot be accepted, the analyzer reports the failure and utilizes the synchronization module to restore an active state concise with the current communication state.

The synchronization modules are generic algorithms that infer the current state of the communication. The algorithm can be exchanged, because each has its strengths and weaknesses. An example for a synchronization algorithm is waiting for a unique event, i.e., an event that is only used in one transition. A detailed examination of the algorithms has not been completed yet and is beyond the scope of this paper.

The monitor mechanism is integrated into the Eclipse [20] debugging framework. An Eclipse Debug configuration is used to specify the state machine used for verification, the source of events, the synchronization module, and the filter for relevant events. We are using the specification model as a passive monitor and no actions may be used in the state machine. Therefore, only the CommunicationView can be used, but a ComponentView may always be transformed into a CommunicationView. The source of events can be a file with a trace, a stream of preprocessed messages or directly the D-Bus of the target system. New communication media are supported by implementing a connector module that reads from the media and converts the messages into API calls.

When the configuration is done, the verification may be started. Eclipse will then switch to the debug perspective that should be familiar to anyone who already used Eclipse for debugging. But instead of running code, the state machine is executed and animated. Animation means that the active states and the transitions used to enter them are highlighted. If the execution of the analyzer is paused, the stack trace shows the history of states passed. The analyzer can pause on found failures, breakpoints or the press of the pause button. While the analyzer is paused, the observer continues to record events. An event queue is used to decouple observer and analyzer. The queue is especially necessary for processing live traces as it is not always

desirable to halt the implementation system while investigating a suspicious sequence of events. The queue also enables to slow down the animation, so that events received in quick succession are still visually observable. Found failures are marked in the state machine and are listed in the problem view of Eclipse.

Test Cases and Regression Tests

The reuse of the specification is not limited to runtime verification. Our specification model can also be used to generate a test suite. The generation has three phases:

1. Elimination of parallel regions
2. Elimination of hierarchical states
3. Selection of test cases

The first two phases facilitate the third phase because many selection algorithms for test suites are readily available for flat finite state machines. Parallel regions can be removed from the state machine by replacing them with several hierarchical states. Each of these states contains one possible sequence of states considering all parallel regions, e.g., first the states of one region, then the states of the second region, and so on. Each time the parallel state would be entered, one of the non-parallel states is chosen randomly instead. Generating all possible parallel regions would lead to a state explosion without much gain in many cases. Therefore, we limit the number of generated states for each parallel state. Hierarchical states can be resolved by moving the inner states out of the containing state and adding all the transitions leaving the containing state to the inner states. Incoming transitions of the containing state are redirected to the first inner state. The containing state can then be removed, as all its semantics were transferred to the inner states. The final test suite is created to fulfill the all-transitions coverage criteria. The algorithm is a reimplementation based on [14]. The selection of the test cases can easily be improved by implementing different coverage criteria.

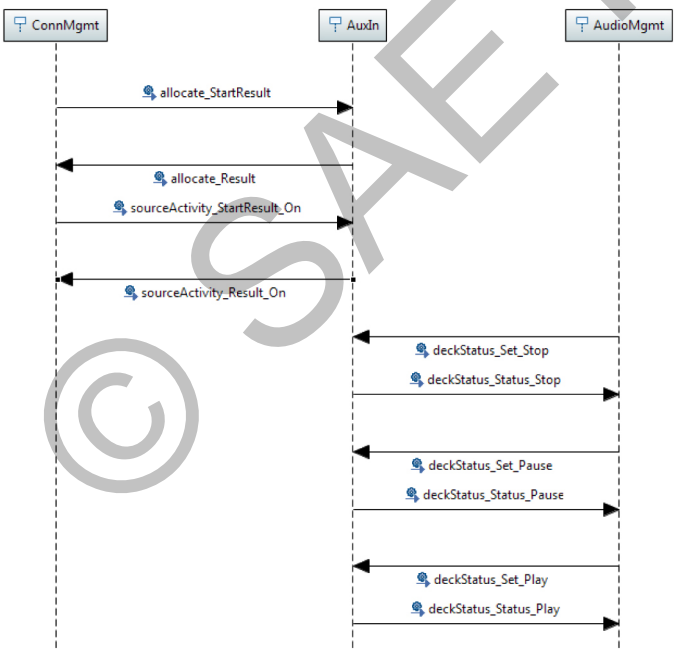

Figure 4. Example of a generated test case

The test cases are generated as UML sequence diagrams. Figure 4 shows an example of a generated test case from the ComponentView model displayed in Figure 5 (see Appendix).

These generated test cases can be used for regression testing. For this purpose, we automatically generate a trace file for every test case. These trace files can then be used as input for testing. With every change in the system these tests can be executed again to see if they are still valid. The following text is a part of the generated trace file of the test case in Figure 4:

```
Do, 2. Okt 10:30:11.130 CEST 2014
TYPE="IPC"
FROM="ConnMgmt"
TO="AuxIn"
INTERFACE="PlayerControl"
EVENT="Method Call"
METHOD="allocate_StartResult"
SERIAL=0

Do, 2. Okt 10:30:11.260 CEST 2014
TYPE="IPC"
FROM="AuxIn"
TO="ConnMgmt"
INTERFACE="PlayerControl"
EVENT="Method Return"
METHOD="allocate_Result"
SERIAL=0

Do, 2. Okt 10:30:11.390 CEST 2014
TYPE="IPC"
FROM="ConnMgmt"
TO="AuxIn"
INTERFACE="PlayerControl"
EVENT="Method Call"
METHOD="sourceActivity_StartResult"
PARAMETERS="sourceActive==true"
SERIAL=1

Do, 2. Okt 10:30:11.520 CEST 2014
TYPE="IPC"
FROM="AuxIn"
TO="ConnMgmt"
INTERFACE="PlayerControl"
EVENT="Method Return"
METHOD="sourceActivity_Result"
PARAMETERS="sourceActive==true"
SERIAL=1

Do, 2. Okt 10:30:11.650 CEST 2014
TYPE="IPC"
FROM="AudioMgmt"
TO="AuxIn"
INTERFACE="PlayerControl"
EVENT="Method Call"
METHOD="deckStatus_Set"
SERIAL=2
```

The trace file is similar to a trace that would be obtained by recording the communication of real implementations of the components executing the same sequence. It can be used also in the same way as the other traces, for example, to perform regression or compatibility tests of models. The trace is fed into the verification framework and checked for failures. If the trace was generated from the same model it is verified with, no failures are expected to be found. However, the generated traces can be used after a model was altered, to see if the new model is still compatible with the execution of the old model.

Discussions

In addition to the audio function, we have successfully applied the introduced methodology for specifying and verifying interface behavior for several other examples. During these applications we made several experiences. Our approach allows hiding technical details from the behavior specification with an additional layer of abstraction. This additional layer, the event mapping, removes the otherwise necessary guards from the behavior model and transforms them into events. Anything that could be expressed with guards can also be expressed with events. However, events can be reused and have a descriptive name. This strongly improves the understanding of the specification.

Moreover, the events are currently organized in a hierarchy. We found that for most of the models this is enough. Only on the rare occasion that a method has numerous parameters this may lead to complex and repeating child events, since every combination has to be captured. On the one hand there are possible solutions for this problem. For example, multiple orthogonal groups of equivalence classes could be defined for each method. Each group checks only certain aspects of a message. The equivalence classes could then be combined in the behavioral model using binary logic. This is more similar to using guards but still abstracts from technical details. The added complexity for triggers and actions in the behavioral description would require additional checks for consistency. On the other hand this might be seen as an indication to revise the interface design as its high complexity may be hardly manageable and maintainable in the end.

Another limitation we experienced is that it is hard to track the communication of individual instances which are contained in a single component. For example, if the AuxiliaryInput was altered and starts a new playback instance on the event sourceActivity_StartResult_On with a certain id, all the deckStatus calls need to include this id. This id is the only identifier for the correct playback instance during verification. The current state machine model does not allow having multiple active markers in one state. This however is necessary to track multiple playback instances which run in parallel. Though, if the maximum number of parallel instances is known beforehand, this can be circumvented by explicitly modeling the state for each instance.

Conclusions

Because of the complex interaction behavior of software components integrated in today's cars, the verification of these components is an expensive task. Our methodology aims to significantly reduce this expense. The approach allows for a multi-purpose specification of in-vehicle infotainment software components' interfaces and interactions. Different views and the separation of the communication and application logic support the developers and testers throughout the phases of the development process. For example, the specification can be used for the verification of the distributed software components. Furthermore, our approach enables a lightweight expandability for other applications or communication technologies. The applicability and advantages of our approach have been shown by the example of a car's audio functions.

Future work will be the improvement of the verification algorithms and the open source release of the tool framework, enabling further application and customizations of the presented methodology.

References

1. Braun, A., Bringmann, O., Rosenstiel, W., “Testing with Virtual Prototypes”, Elektronik Automotive Special Issue MOST, 49-51, 2011
2. Benz, S., “Combining test case generation for component and integration testing”, Proc. of the 3rd Int. Workshop on Advances in model-based testing (A-MOST), 23-33, 2007, doi:10.1145/1291535.1291538
3. Staats, M., Whalen, M., Heimdahl, M., “Programs, Tests, and Oracles: The Foundations of Testing Revisited”, Proc. of the 33rd Int. Conf. on Software Engineering (ICSE), 391-400, 2011, doi:10.1145/1985793.1985847
4. Peters, D., “Automated Testing of Real-Time Systems”, technical report, Memorial Univ. of Newfoundland, 1999.
5. Fuentes, L., Manrique, J., Sánchez, P., “Pópulo: a tool for debugging UML models”, Proc. of the 30th Int. Conf. on Software Engineering (ICSE), 955-956, 2008, doi:10.1145/1370175.1370205
6. Harel, D., Kugler, H., “The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)”, Integration of Software Specification Techniques for Applications in Engineering, 325-354, 2004, doi:10.1007/978-3-540-27863-4_19
7. Mayerhofer, T., “Testing and debugging UML models based on fUML”, Proc. of the 34th Int. Conf. on Software Engineering (ICSE), 1579-1582, 2012, doi:10.1109/ICSE.2012.6227032
8. Moura, R.S., Guedes, L.F., “Simulation of industrial applications using the execution environment SCXML”, Proc. of the 5th IEEE Int. Conf. on Industrial Informatics, 255-260, 2007, doi:10.1109/INDIN.2007.4384765
9. Leucker, M, and Schallhart, C. “A brief account of runtime verification”. *The Journal of Logic and Algebraic Programming*, The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07), 78(5): 293-303, 2009, doi:10.1016/j.jlap.2008.08.004.
10. Delgado, N., Gates, AQ., Roach, S., “A taxonomy and catalog of runtime software-fault monitoring tools”. IEEE Transactions on Software Engineering 30(12): 859-872, 2004, doi:10.1109/TSE.2004.91.
11. Drabek, C., Pramsohler, T., Zeller, M., Weiss, G. “Interface Verification Using Executable Reference Models: An Application in the Automotive Infotainment”, ACESMB@ MoDELS, 2013.
12. Pramsohler, T., Kafkas, M., Paulic, A., Zeller, M. et al., “Control Flow Analysis of Automotive Software Components Using Model-Based Specifications of Dynamic Behavior,” SAE Int. J. Passeng. Cars - Electron. Electr. Syst. 6(2):2013, doi:10.4271/2013-01-0435.
13. Berg, T., Jonsson, B., Raffelt, H., “Regular Inference for State Machines with Parameters”, FASE 2006, LNCS 3922, pp. 107-121, 2006, doi:10.1007/11693017_10.
14. Duan L., “Model-Based Testing of Automotive HMIs with Consideration for Product Variability”, Dissertation, Ludwig-Maximilians-Universitaet Muenchen, 2012
15. GENIVI Alliance, http://www.genivi.org/, October 2014
16. Enterprise Architect, http://www.sparxsystems.de/, October 2014
17. Rational Rhapsody, http://www.ibm.com/, October 2014

18. MODENA, http://www.berner-mattner.com/, October 2014
19. Media Oriented Systems Transport, http://www.mostcooperation.com/, October 2014
20. Eclipse, https://www.eclipse.org/, October 2014
21. AUTOSAR Tool Platform User Group, https://www.artop.org/, October 2014
22. EAST-ADL Tool Platform, https://www.eclipse.org/eatop/, October 2014
23. D-Bus, http://www.freedesktop.org, October 2014
24. Franca, http://eclipse.org/proposals/modeling.franca/, October 2014
25. Papyrus, http://eclipse.org/papyrus/, October 2014
26. Object Management Group (OMG), http://www.omg.org/, October 2014

Acknowledgments

The work has been funded by the Bavarian Ministry of Economic Affairs and Media, Energy and Technology.

Definitions/Abbreviations

AudioMgmt - AudioManagement

AuxIn - AuxiliaryInput

ConnMgmt - ConnectionManagement

DSL - Domain Specific Language

FUML - Foundational Subset for Executable UML Models

IDL - Interface Definition Language

LTS - Labeled Transition System

MOST - Media Oriented Systems Transport

SCXML - State Chart XML

SUT - System Under Test

UML - Unified Modeling Language

APPENDIX

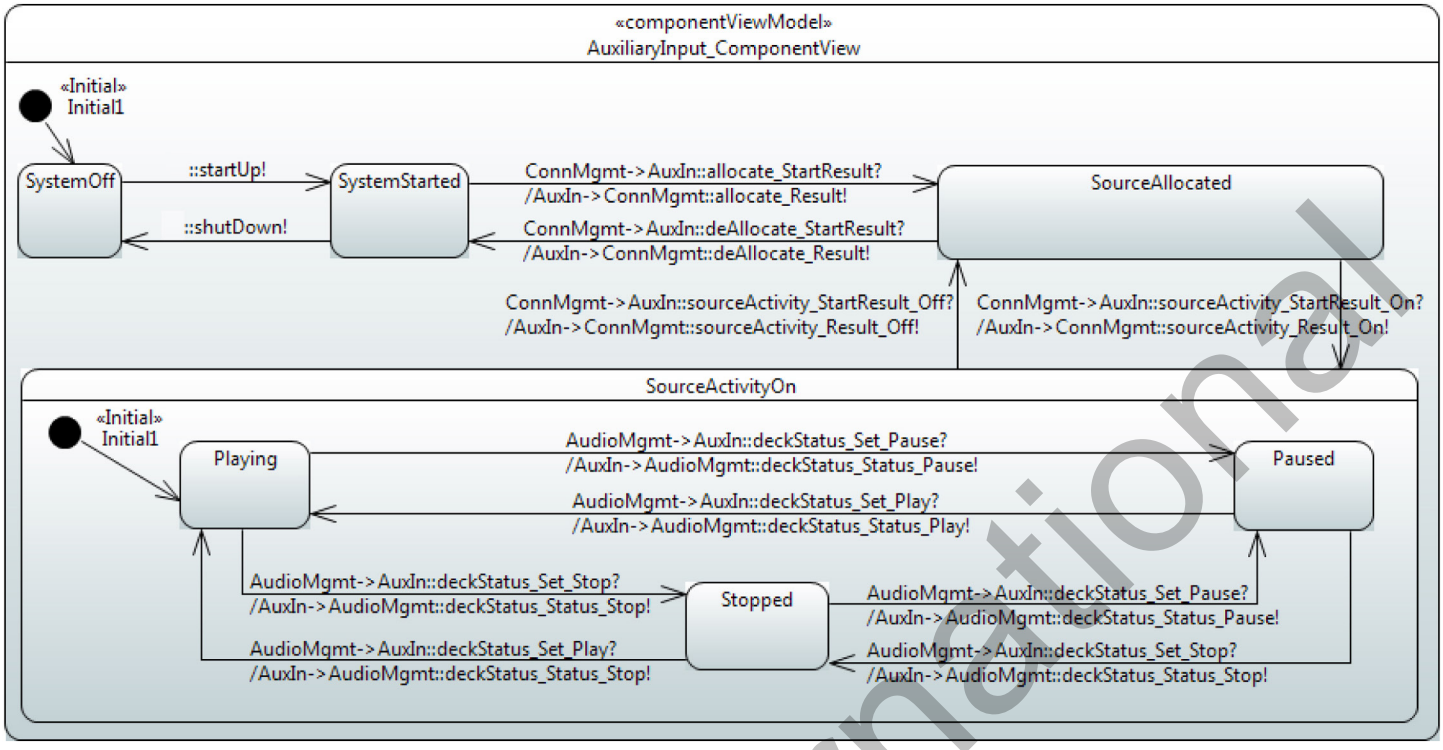

Figure 5. ComponentView model of component AuxiliaryInput

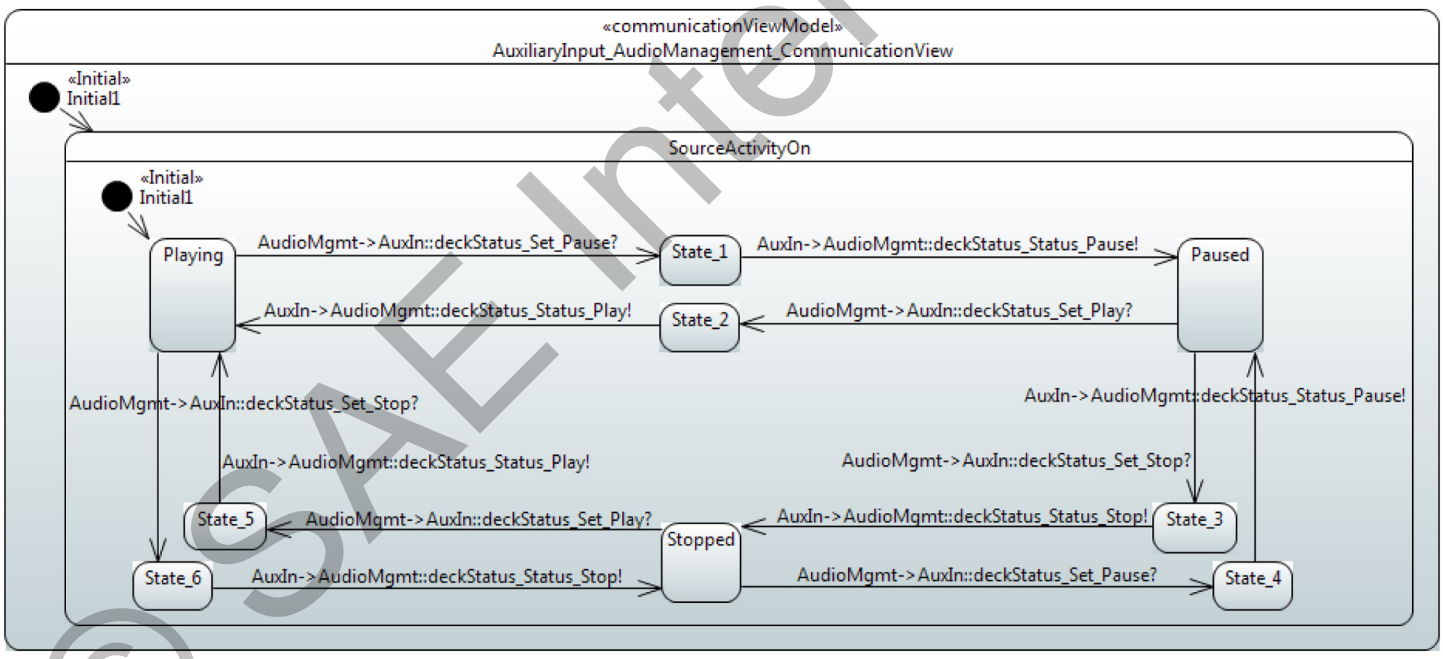

Figure 6. CommunicationView model of communication between components AuxiliaryInput and AudioManagement

The Engineering Meetings Board has approved this paper for publication. It has successfully completed SAE's peer review process under the supervision of the session organizer. The process requires a minimum of three (3) reviews by industry experts.

ISSN 0148-7191

http://papers.sae.org/2015-01-0166