

Control Flow Analysis of Automotive Software Components Using Model-Based Specifications of Dynamic Behavior

Thomas Pramsohler and Mahmut Kafkas
BMW Forschung & Technik GmbH

Annette Paulic and Marc Zeller
Fraunhofer ESK

Uwe Baumgarten
Technische Universität München

ABSTRACT

The amount of software in the automotive domain is steadily increasing. Different ways of abstraction are used to meet this challenge. One of the most promising approaches is *Model-Based Development*. Software components are currently designed and developed with model-based approaches which describe syntactical and behavioral aspects of the components.

Being able to analyze the models in concern to compatibility and traceability of messages and function calls is an important goal. Especially for the behavioral models this task is complex and time consuming. In this paper we focus on the infotainment domain and present three use-cases for control flow analysis in behavioral models. The first is the compatibility analysis between components, the second is the analysis of control flow over multiple connected components and the third is the consistency analysis of sequence diagrams. We present an approach to address these tasks using model transformation and graph-search algorithms.

Finally we show the applicability of our approach using an open-source specification of automotive infotainment software components and an implementation of the algorithms.

CITATION: 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.

INTRODUCTION

Existing software functions in the automotive domain are adapted or enhanced on a regular basis and new functions are introduced from other domains. Model-based development has a huge benefit in this domain driven by functional evolution as it can bring cost and time savings in development and maintenance. Models can be easier reused than code and improve the overall product quality. In the automotive domain the model driven development is applied together with component oriented design and leads to well defined component interfaces. The key issue is to define the interaction of the component with the environment. For this purpose Interface Definition Languages (IDL) and behavioral UML models like state diagrams and sequence diagrams are widely accepted modeling techniques. These models are used mostly for specification purposes and often also for code

generation and verification. The syntactical models are used in the daily work by the developers but behavioral models are not yet introduced continuously in the whole development process. The main reason is the complexity of behavioral models which makes it harder to understand and use the models. System or component behavior in the infotainment domain is currently modeled with UML sequence diagrams and state diagrams. Although it is considerably straightforward to create and use UML models for definition of behavior in small size systems, they are harder to be explored and maintained by the modeler when the system gets bigger.

In this contribution we present an approach to enhance the component centric modeling with analysis methods which allow for the cross-checking of behavioral models of several components. The approach can be used to ensure the model

consistency and to compare behavioral aspects of component variants. We examine a system modeled with different types of behavioral models. One model type is the behavioral model for the definition of component behavior which is defined together with the syntactical interface of the component. Additionally, sequence diagrams describe interface use-cases with interactions between multiple components. In such a modeling approach, it is likely that the same behavioral aspects of one component are defined in multiple models. We define the steps needed to analyze every defined behavior and to compare the models in regard to their behavior. For the analysis, we transform every behavior model in a Labeled Transition System (LTS). The LTS is used as common basis for the model comparison. The analysis results in detailed information about inconsistent behavior. The mismatches can be visualized in the models and have to be resolved by the modeler. The approach not only includes the compatibility check between components but also the analysis of control flow which results due to the combination of the single component behaviors.

The analysis is evaluated on the models of the current GENIVI *AudioManager* specification for future infotainment platforms [10]. We give examples for the application of the analysis methods to ensure the integrity of these models. Finally we show the practical application of the analysis with a prototype implementation using the Eclipse Papyrus modeling framework [14].

In the next section we briefly summarize the current applied methods of software modeling and analysis. Then we describe the three use-cases for behavioral model analysis. In the following section we present the syntactical and behavioral software models using UML profiles and an open-source specification of infotainment components. In the following two sections we present our approach for model analysis and show our results using a case study. The last section summarizes the results.

STATUS QUO

In the automotive infotainment domain UML software models are widely applied in different development steps. State of the art modeling tools include structural system modeling with package, component and class diagrams and behavioral modeling with state diagrams or sequence diagrams. State-of-the-art CASE Tools like Enterprise Architect [15] or Rational Rhapsody [16] provide a common data basis and support the collaboration between teams. The models are basically used for specification and code generation. The generated code includes target code and validation code for the testing phases. Control flow analysis is poorly introduced in the modeling processes and poorly supported by the tools. In this section we present related scientific work in the field of control flow analysis in software models.

Model consistency problems are a common issue and are usually caused by multiple views and models which are used

in the development process. There exist numerous approaches addressing consistency analysis for UML models. A systematic review was provided by Lucas et al [1]. The authors include several types of consistency including horizontal semantic consistency which is addressed in this paper. The review reveals that there are many formal techniques and approaches but they often lack tool support or CASE tool integration. Some of the approaches [3,4] use model checkers to solve the task of consistency checking, and other approaches use model transformation and finite transition systems [5] or Petri nets [6]. The approaches are formally defined and evaluated with example models. In our contribution we do not focus on the formal definition of the approach, but on seamless integration of a consistency check in a system modeled with both, structural and behavioral information. Additionally we present an implemented solution for CASE tool integration and visualization of results.

For the extraction and visualization of control flow paths several different research fields have to be involved. The mayor problem is the synthesis of models with related information. After the synthesis, the flow path can be extracted with graph search algorithms and different heuristics. Several approaches for model synthesis are revised by the authors but none of them supports the extraction of control flow paths over several components. It depends on the specification model which additional information can be obtained with the synthesis. For interface automata [1] the composition of two interface automata is defined and used to check the compatibility between two components. The approach is extended in several works including the automatic composition of safety critical components. Interface automata are similar to the modeling approach used in this contribution but allow only for the description of the boundary between two components. For this reason additional information has to be added to get the connection between the different interfaces of one component in order to calculate the control flow through a system. Uchitel et al [7] present an approach for the synthesis of sequence diagrams. It allows the modelers to translate several sequence diagrams in one behavioral specification mainly for the purpose of model construction and model reuse. Other approaches highlight the analysis of control flow in UML models. Garousi et al [8] analyze behavioral dependencies in UML sequence diagrams and propose views for the visualization of the results.

SOFTWARE MODEL AND CASE STUDY

The modeling approach presented in this paper is described using the GENIVI *AudioManager* as a case study [12]. The *AudioManager* is a framework which is used to manage audio in the in-vehicle-infotainment context. Therefore it shall provide a common API for other components and allow the usage of interchangeable audio routing plugins [12]. The main purpose of the *AudioManager*

is to connect sources to sinks and to decide whether an audio signal should be transmitted to a sink in a specific situation or not.

The GENIVI AudioManager consists of the components *AudioManagerController*, *AudioManagerDaemon*, *CommandInterfacePlugin* and *RoutingInterfacePlugin*. The *AudioManagerController* provides all logic and control functions of the AudioManager. It has to be implemented according to the requirements of the specific system. The *AudioManagerController* decides if a connection or volume change request is processed or not. The *AudioManagerController* communicates with the controlling instances and the routing plugins and provides an interface with all system and communication events it has to handle. The *AudioManagerDaemon* defines all interfaces and implements a database as well as base functions. It is also responsible for starting up and shutting down all other components. The *CommandInterfacePlugin* handles the communication between the *AudioManagerDaemon* and the AudioManager commanders like the HMI for example. The *RoutingInterfacePlugin* is needed to bond the AudioManager to different busses.

In order to understand the communication between the components of the AudioManager the startup and rundown of the components is described in the following.

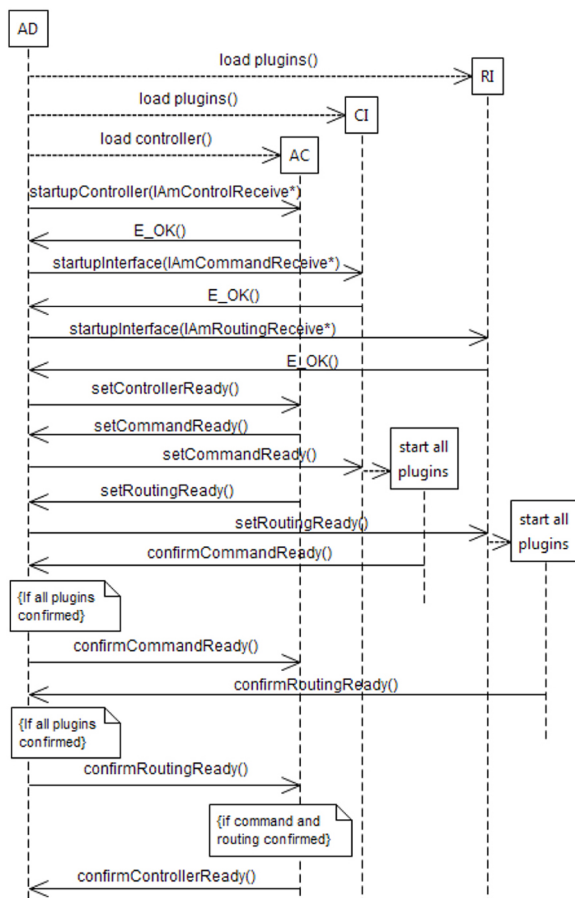


Figure 1. Sequence Diagram of startup behavior

If the *AudioManagerDaemon* (AD) is initialized it first of all loads all *CommandInterfacePlugins* (CI), *RoutingInterfacePlugins* (RI) and the *AudioManagerController* (AC). Figure 1 shows the startup behavior as UML sequence diagram. By communicating with the interfaces provided by these components it starts up the controller and the interfaces, which in return send an OK message if the startup was successful. In a next step AD sends the message 'setControllerReady' to AC which then asks AD to set also the command interfaces and routing interfaces ready. Thereupon these interfaces start all their plugins and afterwards confirm AD that they are ready. After AD received the confirmation of the *CommandInterfacePlugins* and *RoutingInterfacePlugins* it passes that information on to AC which then completes the startup by sending its confirmation.

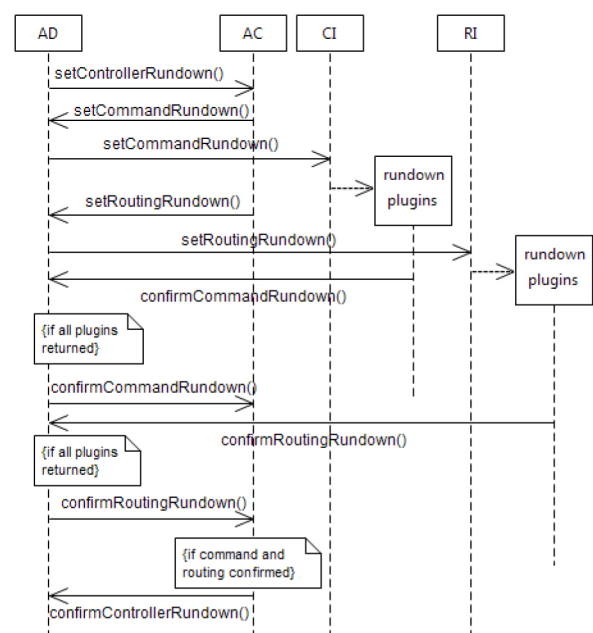


Figure 2. Sequence Diagram of rundown behavior

Figure 2 displays a sequence diagram with the rundown behavior of the AudioManager. If the *AudioManagerDaemon* (AD) gets the message to start the rundown it requests the *AudioManagerController* (AC) to run down at first. AC then again asks AD to initialize the rundown of the *CommandInterfacePlugins* (CI) and the *RoutingInterfacePlugins* (RI). After those interfaces finished to rundown all of their plugins they send a confirmation message to AD which forwards the confirmation to AC. Now AC also confirms its rundown to AD which completes the rundown process. [11]

In our modeling approach different types of models are provided to describe the components of a system and their communication behavior. The *System Structure Model* specifies all components of the system and the communication between them. The *Component Behavior Models* describe the behavior of a single component at its

boundary, whereas the *Interface Use-Case Model* is used to define interface use-cases between several components.

System Structure Model

The *System Structure Model* is used to specify all relevant components of a system and the communication between these components via their interfaces. Figure 3 shows the System Structure Diagram of the AudioManager. *AudioManagerController* (AC), *AudioManagerDaemon* (AD), *CommandInterfacePlugin* (CI) and *RoutingInterfacePlugin* (RI) are modeled as components. Like in UML component diagrams provided and required interfaces have to be assigned to the components but are not visualized in our diagram.

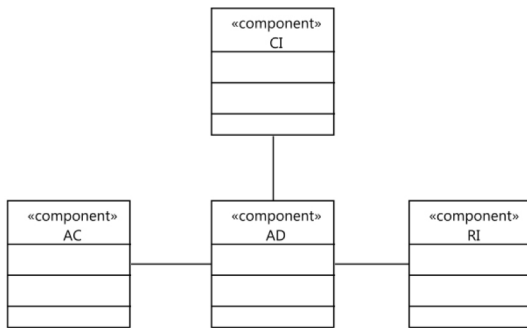


Figure 3. System Structure Diagram of the GENIVI AudioManager System

In each interface a set of *signal events* can be defined and mapped to a function call of the interface. These signal events will be used later in the behavioral models to define the interaction. An association between two components illustrates that they can communicate with each other. Such a *communication association* implies that one component can invoke the functions provided by the interface of the other component and vice versa. After the System Structure Model is completed, a Component Behavior Model can be created for every component contained in that model.

Component Behavior Model

For component behavioral modeling we use a subset of UML state diagrams with restricted transition semantics. The Component Behavior Model is composed out of *initial states*, *states*, *transitions*, *parallel regions* and *join states*. Figure 4 describes a simplified version of the *AudioManagerDaemon* (AD) behavior which only includes some startup messages and the processing of a connect command. A transition is used to depict a possible change from one state to another (displayed as a directed arrow). A *trigger* belongs to a transition and it contains a signal event provided by the interface of the modeled component (e.g. CI->AD::connect). Each event occurrence includes information about the sender (in this case CI) and the receiver (AD). If the information about sender and receiver is missing, it is an internal signal event of the component. If AD is in the state

‘AcceptingCommands’ and CI fires a connect event the transition will be executed and the target state of that transition is now active (state ‘connectHandled’). Every transition can also provide an *action*, which is an event provided by the interface of another component the modeled component can communicate with (in this example AD->AC::hookUserConnectionRequest). If the transition is executed the component fires the event defined in the action of the transition. In addition to that every state in a Component Behavior Model can have one or more regions. If a state has two or more regions it represents parallel behavior of a component (e.g. ‘State2’ in Figure 4). If a state contains only one region a hierarchical structure can be modeled (see states ‘working’ and ‘startup’). Finally a join state is used to combine different paths of a parallel region.

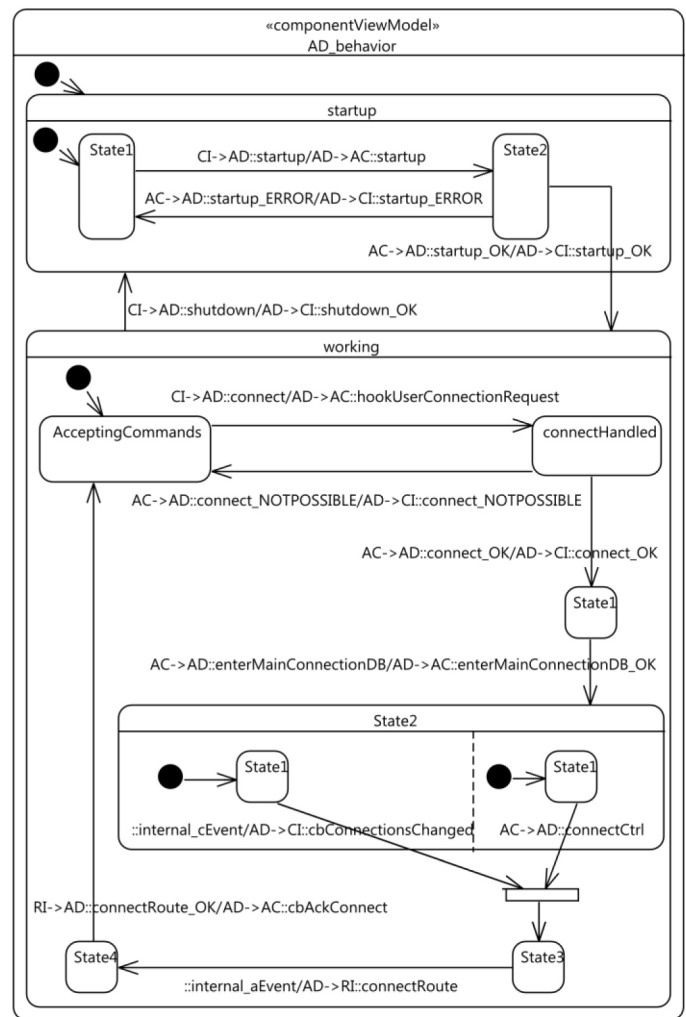


Figure 4. Component Behavior Diagram of the AudioManagerDaemon

The modeled behavior in Figure 4 specifies the initialization of a new connection between a source and a sink. At first the startup has to be done between the *AudioManagerDaemon* (AD) and the *AudioManagerController* (AC). This example does not