Scenario-Based Verification of Automotive Software Systems

Matthias Gehrke, Petra Nawratil
Software Quality Lab (s-lab),
University of Paderborn,
D - 33095 Paderborn, Germany
[mgehrke|penaw]@uni-paderborn.de

Oliver Niggemann
dSPACE GmbH,
Technologiepark 25,
D - 33100 Paderborn, Germany
ONiggemann@dspace.de

Wilhelm Schäfer, Martin Hirsch*
Software Engineering Group,
University of Paderborn,
D - 33095 Paderborn, Germany
[wilhelm|mahirsch]@uni-paderborn.de

1 Introduction

Within the automotive industry, software engineering becomes more and more important. Especially component-based design is a popular approach in order to specify large, complex and reusable software systems.

The reuse of software components has become a major challenge for the automotive software development. By reusing components, manufacturers and suppliers can (*i*) revert to already tested software modules, thus minimizing potential software hazards, (*ii*) save development efforts and (*iii*) transfer product line approaches more easily into the world of software development. Several approaches exist to specify the structure, including interface definitions, for such reusable software components. AUTOSAR is the most well-known approach in the field of automotive software development.

In order to specify automotive systems with the component-based approach it is necessary to specify some aspects of the behavior of the components as well. More precisely, the interface descriptions of the components have to be enhanced with additional information, mainly timing aspects[1]. Just when the interfaces fit together, a feasible component structure arises.

After connecting the components, a large component structure may arise and therefore a very complex structure of connected behavior models emerges. This complexity may lead to several problems, because a wrong definition of the behavior models may invoke a not wanted behavior. At a specific degree of complexity, an engineer is not able to control the connected components and guarantee that no unwanted behavior will happen. So, some

*Supported by the University of Paderborn

[1]The internal structure of a software component will not be available, because that is the specific knowledge of the manufacturer and therefore constitutes one of his assets.

kind of automatic tests is needed, e.g. model checking.

This paper addresses one problem occurring when such reusable software components are used: How can be checked whether a system constructed from black boxes whose internals are unknown to him meet important constraints? This holds especially true for temporal requirements.

One solution is of course to put the software components on an evaluation board or electronic control unit (ECU) and have extensive runtime tests. This solution costs time and therefore money and still fails to verify all possible scenarios. It also assumes that component implementations already exists. But in many cases software developers first specify component interfaces which are then implemented by function developers. But even in this case the software developer may want to verify whether the designed system works correctly — presuming that the function developers fulfill all requirements.

A different approach solves these problems: component interface descriptions are enhanced with additional information, especially with timing models. Based on these information, formal verification methods such as model checking[2] are used to answer typical questions such as: (*i*) Given specific input, does the system reach a specific state within a given deadline? (*ii*) Does the system start up within a given time period? (*iii*) Do deadlocks exist?

These techniques of course do not replace runtime test; they only allow the early verification of some system properties. With this approach it is possible to check the behavior of an automotive software system in an early phase of the development process.

2 Example: Adaptive Cruise Control

The following example was developed within a project between the dSPACE GmbH and the Software Quality Lab (s-lab) of the University of Paderborn.

2.1 Structure

Figure 1 shows the simplified structure of an adaptive cruise control[3]. An adaptive cruise control keeps the car at a given constant speed. If an obstacle like another car make the retention of the predefined velocity impossible or dangerous, the car's speed is reduced accordingly. The system accelerates the car automatically, if the obstacle disappears. Details may be found in [Rob03]. The system is specified with UML 2.0 [OMG03]. The shown diagram is a combination of the deployment- and component diagram.

To simplify matters, not all software components and not all connections are shown. Nevertheless this example is typical for several reasons:

- The adaptive cruise control uses several electronic control units (ECUs). The reader may note that these ECUs are normally produced by different suppliers. In Figure

[2] In this paper we use the model checker Uppaal to test our approach.

[3] For several reasons, this example is far from being a realistic adaptive cruise control system: (*i*) intellectual property had to be protected, (*ii*) the example's purpose is comprehensibility, not completeness and correctness, and (*iii*) the authors do not develop adaptive cruise controls.

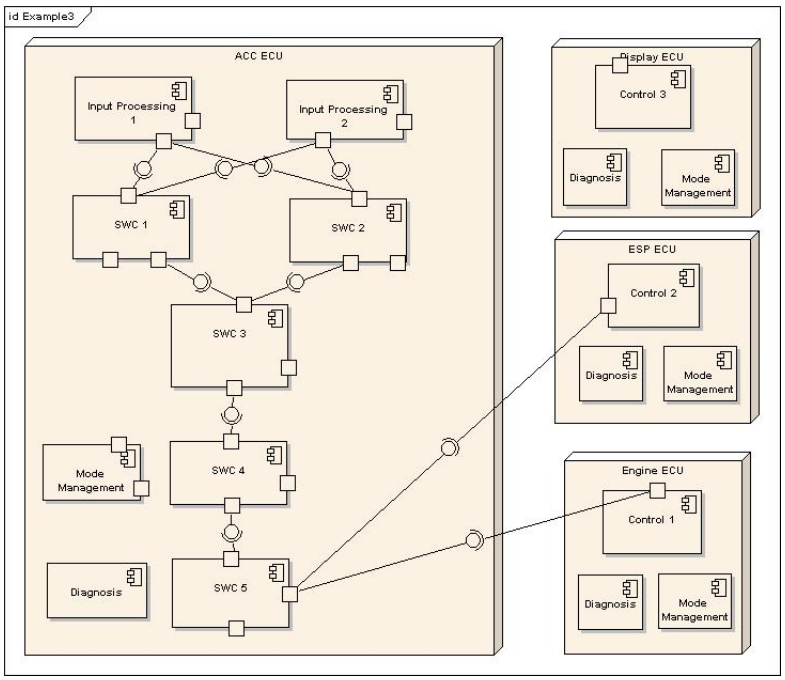

Figure 1: *A simplified Adaptive Cruise Control*

1 an adaptive cruise control unit (ACC ECU), an ESP (electronic stability control) ECU[4], an engine ECU, and the display ECU are shown.

- The adaptive cruise control is a real-time, safety critical system. Its reactions must meet strict timing constraints. Since any non-conformance to timing or behavioral requirements can result in fatal consequences, the highest safety standards are mandatory.
- The software is structured into (hierarchical) software components. This software structure defines the system's functionality while the distribution of software components to ECUs is responsible for the system's timing.
- The system's functionality is specified in more detail in form of timing models associated with the component interfaces (cf. Section 2.2).

2.2 Timing behavior

The next step is to specify the behavior of the interfaces of the components in consideration of timing requirements. In this approach we use Timed Automata, to specify the timing description of the interfaces.

[4]responsible for the deceleration

The example in Figure 2 models the behavior of a sensor data processing component (*Input Processing 1* in Figure 1), i.e. a software module that gets new sensor data from some sensor hardware and sends it to other adaptive cruise control software modules (e.g. *SWC 1 and SWC 2* in Figure 1). Three states are modeled: (*i*) running, (*ii*) shutting_down, and (*iii*) not_running.

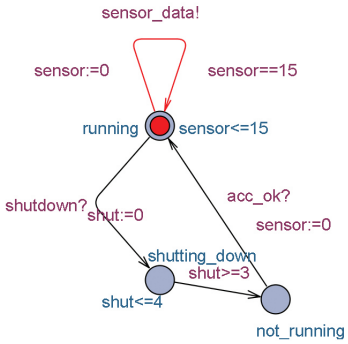

Figure 2: *A Timed Automaton in Uppaal*

The component sends new data every 15 time units (e.g. ms). The “sensor_data!” notion at the transition from and to the “running” state denotes the firing of a “sensor_data” event. The “shutdown?” notion at the transition to the “shutting_down” state means that an event “shutdown” serves as a trigger for that transition. The “not_running” state may be left by the reception of a “acc_ok” event.

The example in Figure 3 models the behavior of the port of the SWC 1 component, which is connected to the *Input Processing 1* component. This Timed Automaton synchronizes with the first Timed Automaton via the ”sensor_data” event. That is, the transition from the starting state to the ”processing_data” state is triggered by this event. The processing of the data should happen in the time interval [4, 7] which is annotated in the invariant of the state and the guard of the outgoing transition, which fires the ”object_list” event, to which another Timed Automaton will react. A shutting down is modeled just as in the first Timed Automaton.

This small example indicates that there will be a lot of synchronization between many, different Timed Automata, especially when a lot of components are connected to each other.

In automotive systems several constraints exist concerning the temporal behavior of the components. In such a case, model checking can be used, to verify these constraints.

2.3 Limits of component-based Verification

The verification of the behavior of components within a component structure has some limits. For example, in our approach we have the constraint, that a specific state within a specific Timed Automaton has to be reached within a specific period of time. The verification of this constraint is not possible with the standard mechanism of Uppaal (Timed

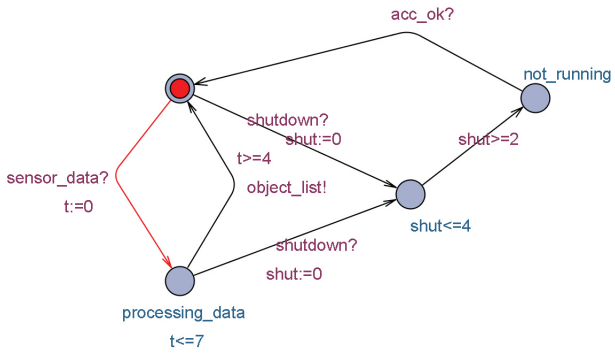

Figure 3: *A second Timed Automaton in Uppaal*

Automata and TCTL). The reasons are:

- Every Timed Automaton has its local clock. Global clocks are not available.
- The triggering event and the state, that should be reached, can be in different Timed Automaton.

Without a global clock it is not possible to specify all needed TCTL formulas. That means, that it is not possible to specify a constraint in TCTL, in order to verify, if a specific state will be reached within a specific period of time. Within Uppaal it is generally possible to define global clocks, but this is a contradiction to the component-based design. So another solution is needed.

2.4 Global Property Verification

As described in the last section, scenarios exists, which cannot be verified by Uppaal. So what is missing. Mainly a way to formulate the questions with the means provided by model checkers such as Uppaal.

A query, i.e. a question which should be answered by the model checker, consists of two main parts:

1. A so-called *scenario timed automaton* is used to formulate the query scenario. This scenario mainly comprises the dynamics of the situation-to-be-verified, i.e. the sequence of fired and received events that define the situation-under-test.

   To specify these scenario timed automaton we primarily select states and transitions from the individual Timed Automata, modeled for the component interfaces. Then we combine these selected states and transitions with further states and transitions in order to get a proper Timed Automaton. The result is a scenario timed automaton, which can be used to verify the mentioned constraints.

   The scenario timed automaton provides a global time since its local clock is synchronized via fired and received events with the local times of the individual Timed

Automaton. This is important because the Timed Automaton associated with the interfaces may only use information that is local to the corresponding software component, i.e. it must be possible to reuse these Timed Automata in other software systems and with other query scenarios. So the use of a global clock within the individual Timed Automaton is not desired.

2. Temporal logic formulas are used to define the invariants that should be guaranteed in a given scenario. As outlined above, for timing constraints these formulas may refer to the local clock of the scenario timed automaton.

For the adaptive cruise control example typical scenarios (model queries) are:

- If a new obstacle appears, does the car reach the modified speed within a given time limit?
- If that obstacle disappears, is the optimum speed reached again within a given speed limit?
- Is the active cruise control functionality turned off within a given time limit, if a severe problem is detected within the system?
- If a severe problem occurs within the system, is the driver informed within a given time limit?
- Is the system started within a given time limit?
- Are any deadlocks in the system?
- Many model queries are defined between two specific components, i.e. they check the correctness of one connection. E.g. between the sensor driver component and the next data processing component it makes sense to verify that sensor data is delivered within a given time interval.

A simplified scenario timed automaton for the first query is depicted in Figure 4.

The Timed Automaton begins in the "start" state. If new data is available from the sensors, i.e. if the "sensor_data" event is received, the Timed Automaton turns to the "start_of_speed_adjustment" state. After the system has adjusted its speed, a "speed_ok" event is received and the system turns to the "speed_is_ok" state. This state is a so-called committed state, i.e. it directly moves on to the "end" state.

This scenario timed automaton defines the dynamics of one specific scenario. A local clock t is reset at the beginning which allows temporal formulas to refer to a scenario specific time scale. The committed state at the end is necessary because it allows such formulas to refer to the point of time when the speed is adjusted, i.e. the state "speed_is_ok" is reached.

The temporal formula that checks for all possible cases whether the speed is adjusted after 10 time units than looks like this:

$Scenario1.speed\_is\_ok -- > Scenario1.t <= 10$

Scenario1.speed_is_ok refers to the scenario timed automaton from Figure 4. Scenario1.t is the local clock of that scenario timed automaton. Uppaal can now take the timed automata associated with ports and connections and the scenario timed automaton for a verification of the temporal formula.

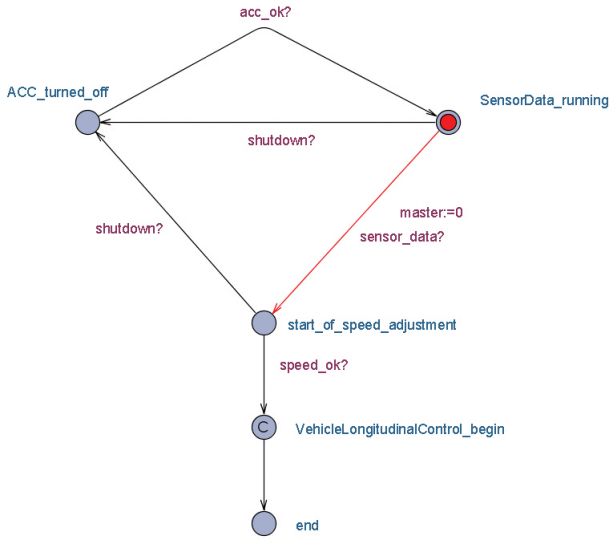

Figure 4: *A scenario timed automaton*

3 Conclusion & Future Work

In this paper we presented an approach for the verification of distributed, component based automotive systems, modeled with Timed Automata. One problem in this context is the appropriate description of the required specification. Therefor, so-called "scenario timed automata" are introduced which enables the engineer to verify more complex requirements.

One of the next steps now is to systematize and to automate the specification of the scenario timed automata. Therefore a mapping from the individual Timed Automaton to the scenario timed automaton is needed. To support this mapping, an adequate tool-support is needed. Unfortunately Uppaal can not be used, because with Uppaal it is for example not possible to specify component- or deployment diagrams and associate them with the Timed Automata.

3.1 FUJABA integration

In order to support specifying the architecture of embedded real-time systems which are usually distributed systems, Fujaba[5] has been extended by UML2.0 component diagrams in the Fujaba Real-Time Tool Suite [BGH+05]. The behavior of single components is specified by *Real-Time Statecharts* [GTB+03]. Real-Time Statecharts introduce clocks and allow the specification of worst-case execution times for activities and deadline for transitions. With the Fujaba Real-Time Tool Suite we are able to transform these Real-

[5] `http://www.fujaba.de`

Time Statecharts into Timed Automata in order to be used within the model checker Uppaal. One future task is to integrate the idea of the "scenario timed automata" in the Fujaba Real-Time Tool Suite and to extend the available transformation if necessary.

References

[BGH+05] Sven Burmester, Holger Giese, Martin Hirsch, Daniela Schilling, and Matthias Tichy. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, May 2005.

[GTB+03] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[OMG03] OMG. *UML 2.0 Superstructure Specification*. Object Management Group, 2003. Document ptc/03-08-02.

[Rob03] Robert Bosch GmbH. *ACC Adaptive Cruise Control*, 2003.