Verification of AUTOSAR Software Architectures with Timed Automata

Steffen Beringer[1(✉)] and Heike Wehrheim[2]

[1] dSPACE GmbH, Paderborn, Germany
sberinger@dspace.de
[2] Paderborn University, Paderborn, Germany

Abstract. Today, automotive software is getting increasingly complex while at the same time development cycles are shortened due to time and cost constraints. For the validation of electronic control unit software, this results in a major challenge. Especially for safety critical software, like automotive software, high quality must be guaranteed. Formal verification of automotive software architecture *models* enables early verification of safety constraints, before the complete system is assembled and ready for simulation. One option for formal verification of safety critical software is modeling and verification using timed automata. In this paper, we present a method for the verification of AUTOSAR software models by transforming the software architecture as well as the corresponding AUTOSAR timing constraints into timed automata.

1 Introduction

Complexity of electronic control units (ECUs) and controller algorithms in cars increases, for example due to more comfort functionalities and more complex controllers for electric vehicles. Therefore development and test of these types of systems becomes time consuming. Late availability of prototype ECUs hinders the early validation of the overall system. One necessary condition for the integration of various controller functionalities from different vendors into a combined system is to have a standardized description of the software architecture and integration methodology. In this regard, AUTOSAR [1] has become the de facto standard in the automotive domain as it provides a common infrastructure for automotive systems of all vehicle domains based on standardized interfaces.

The company dSPACE[1], in which this work has been carried out, is the world's leading provider of solutions for developing ECU software and mechatronic controls. The dSPACE product area *Virtual Validation* comprises tools for using virtual (i.e. software-based) ECUs for testing and validating ECU software throughout the whole development process by a PC-based simulation. By using virtual validation, development, verification and validation tasks can be performed much earlier and also reduce the number of additional tests, prototype systems and ECU prototypes needed. Virtual Validation needs a virtual ECU

[1] http://www.dspace.de.

M.H. ter Beek et al. (Eds.): FMICS-AVoCS 2016, LNCS 9933, pp. 189–204, 2016.
DOI: 10.1007/978-3-319-45943-1_13

(V-ECU) for the PC-based simulation. Therefore in a first step, the V-ECU has to be configured, generated and compiled out of an existing AUTOSAR software architecture. However, this step takes some time to execute. Furthermore, when errors in the simulation are detected, it is necessary to repeat this step. Another point is that all controller algorithms have to be fully implemented, but in early validation phases this case is rather rare.

Therefore for the early validation of an AUTOSAR architecture, analysis methods have to be investigated, which exclusively rely on the existing software architecture, because controller software is not available. Furthermore model properties exist which even cannot be validated by elaborated simulation scenarios. This applies for example for timing requirements which have to be met under all possible circumstances. The validation of timing requirements therefore needs special analysis methods, which cover all possible corner cases. An established method for verification of timed systems is modeling and verification of the system as a network of timed automata and the specification of properties with the help of temporal logic.

This work presents an approach for the transformation of AUTOSAR architecture models into a network of timed automata. Furthermore, AUTOSAR timing constraints as part of the AUTOSAR model are transformed. By exclusively considering the architecture model and not the controller functionalities, analysis can be performed early in the development process. Model checking of timed automata in addition can prove correctness of the architecture with respect to the timing requirements.

Related Work. There are different methods for the analysis of timing requirements. Besides the modeling and verification of timed systems via timed automata, methods exists that are based on scheduling analysis methods. In the works presented in [2,3] a compositional scheduling approach based on traditional scheduling theory in real-time systems is presented. The approach assumes that signals can arrive at components only in a restricted fashion, e.g. with fixed frequency and maximum jitter. The arrivals are specified in event functions. If signal arrivals do not match the predefined models, timing analysis becomes imprecise [4]. *Real-Time Calculus* is a framework for performance analysis of real-time systems, which is based on the network calculus [4]. By specification of an *Event Stream Model* a signal flow through a system can be analyzed. This is a more generic framework than the one in [2]. Both methods apply a different sort of abstraction on analysis level than our method. Furthermore, we directly apply our method to AUTOSAR timing extensions, while other methods only partly describe the application onto the AUTOSAR standard. A similar approach described in the work presented in [5] also utilizes timed automata for the analysis of AUTOSAR architectures. In contrast to this approach, the transformations enable general timing error detections, but do not apply transformations to the AUTOSAR Timing Constraints, which is nessessary for the analysis of timing requirements. Further approaches for timed automata suggest the method of constructing test automata (or *Scenario-Automata*) for the specification of requirements [6], but also do not consider AUTOSAR Timing Extensions.

In the work presented in [7] tool support for the verification of AUTOSAR timing requirements is presented. The requirements are verified by comparing them against specified *timing guarantees*. For this approach, timing guarantees have to be specified, which is not necessary in our approach. Besides methods for timing analysis of software architectures there is a lot of work dealing with timing based on single program tasks available as code snippets or binary artifacts [8]. These methods can determine upper bounds for the Worst-Case Execution Time and thus are a necessary prerequisite for the analysis on architecture level, where the artifacts are assembled.

2 Background

This chapter discusses the foundations of AUTOSAR and the integrated timing extensions, because most of the software in automotive contexts is currently AUTOSAR-based. Furthermore, foundations of timed automata are treated. Timed automata are used for the verification of the AUTOSAR architecture.

2.1 Introduction to AUTOSAR

AUTOSAR[2] is short for **AUT**omotive **O**pen **S**ystem **AR**chitecture and is the established standard for the development of automotive software. AUTOSAR defines the architecture and interfaces of the software as meta-model as well as the file format for data exchange. Furthermore, the standard defines its own development methodology. The concepts of this paper are based on the current AUTOSAR version 4.2.

On the outer level AUTOSAR software is structured as layered architecture (see Fig. 1). There are three different layers:

- The *application layer* is the upper software layer. It contains the actual controller software, which includes mostly controller algorithm implementations in the automotive domain. Inside of this layer software is structured in a component-based architecture. Therefore software components are modeled, which can communicate via ports and connections.
- The *Runtime Environment layer* (RTE) administrates the communication between software components, and furthermore the communication between software components and basic software parts (see below). It realizes a standardized interface for the software on application level.
- The *basic software layer* incudes modules for basic functions of ECUs. The basic software layer is subdivided into a *Service Layer* (purple), an *ECU abstraction layer* (green) and a *Microcontroller Abstraction Layer, MCAL* (red) (see Fig. 1). The service layer contains the main ECU services like operating system, ECU state management, services for diagnosis, memory services and communication services. The ECU abstraction layer realizes an abstraction between ECU hardware for the upper layers and contains modules for the

[2] http://www.autosar.org.

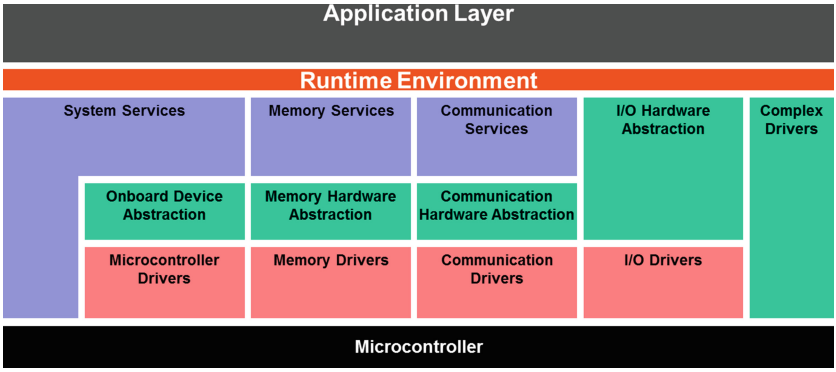

Fig. 1. AUTOSAR layered architecture taken from [9] (Color figure online)

access of hardware peripherials. The MCAL provides low level driver modules and acesses the hardware directly. A detailed description of all modules can be found in [9].

The AUTOSAR Authoring Tool SystemDesk®. SystemDesk®[3] is the tooling environment for AUTOSAR models from dSPACE. It supports sophisticated and extensive modeling of AUTOSAR architectures by providing a rich graphical user interface as well as code generation for virtual ECUs. Graphical model representations are available for important elements. For example software components, ports and connections within a software composition can be visualized in a *Composition Diagram*. Furthermore, single software components with their ports, interfaces and data types can be visualized in a *Component Diagram*. Other model elements are ordered hierarchically in a tree structure.

Example 1. In the following we will consider a simple example AUTOSAR software architecture, which manages the left and right direction indicators of a vehicle. The application layer consists of several software components, which comprise several so-called runnable entities, which contain executable software. The example architecture is shown in Fig. 2. The two software components on the left read in sensor data and check for errors before forwarding the signal data to the next software component. The *IndicatorComposition* software component receives the raw sensor values and encapsulates several runnable entities for preprocessing of the signal values as well as the logic of the system. The actuator software components on the right are responsible for activating the left respectively right bulb of the direction indicator. Furthermore, the example contains a configuration of the RTE, and on the *basic software layer* the configuration for the Operating System. Other basic software modules are not considered in this example.

[3] http://www.dspace.com/en/pub/home/products/sw/system_architecture_software/systemdesk.cfm.

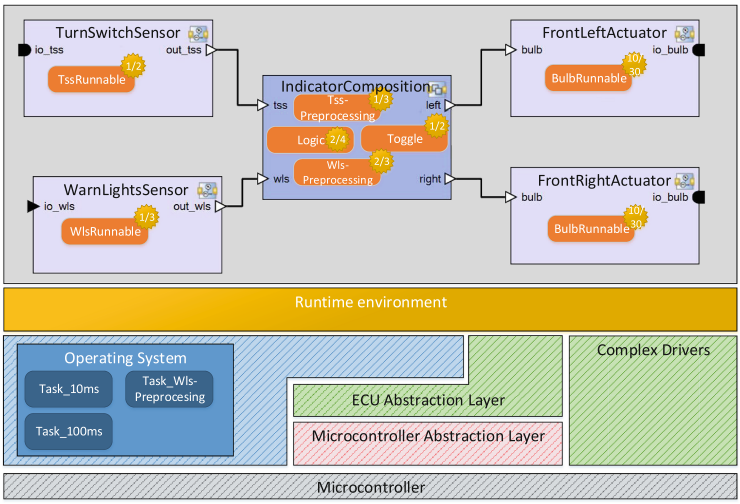

Fig. 2. Example software architecture

2.2 Timed Automata

While AUTOSAR specifies a formal syntax defined as an OMG meta model, its semantics is only described in a textual manner. To formally verify timing requirements on AUTOSAR, we need to define a formal semantics for the timing relevant meta model elements. Here, we employ timed automata as they are capable of formally describing timing behavior. Timed Automata were first introduced 1994 by Alur and Dill [10]; in the following, we follow the notation of [11].

Definition 1 (Timed Automata). *A* timed automaton *is a tuple* $\mathcal{A} = (L, B, B^*X, I, U, E, I_{ini})$ *with a finite set of locations* L*, a set of signals communicating via handshake* B*, a set of signals communicating via broadcast channels* B^**, a set of clocks* X*, an assignment of invariants to locations:* $I : L \rightarrow \Phi(X)$*, a mapping for the locations whether they are urgent (so that time is not allowed to increase)* $: U : L \rightarrow \{true, false\}$*, a set of edges labeled with an action, a guard and a set of clocks, which need to be reset:* $E \subseteq L \times B \cup B^* \times \Phi(X) \times \mathcal{P}(X) \times L$*, and an initial location* $I_{ini} \in L$*.*

Here, $\Phi(X)$ specifies a set of clock constraints (like $x < 3$, see [11]). A *configuration* of a timed automaton is a pair of a location and a *clock valuation* $\nu : X \rightarrow Time$, where $Time \in \mathbb{R}^{(\geq 0)}$ are the real numbers. We use $\nu \models \phi$ for a clock constraint $\phi \in \Phi(X)$ if the constraint is true for the clock valuation. In Fig. 4 an example automaton is shown.

Definition 2 (Semantics of Timed Automata). *The* operational semantics of a timed automaton $\mathcal{A}$ *is defined as a labelled transition system* $T(\mathcal{A}) = (Conf(\mathcal{A}), \rightarrow, C_{ini})$*, where* $Conf(\mathcal{A}) = \{\langle l, \nu \rangle \mid l \in L, \nu : X \rightarrow Time,$

$\nu \models I(l)\}$, *an initial configuration* $C_{ini} = \{\langle l_{ini}, \nu_{ini}\rangle\}$ *and a transition relation* $\rightarrow \subseteq Conf(\mathcal{A}) \times (Time \cup B) \times Conf(\mathcal{A})$ *with two different types of transitions:*

- *delay-transition:* $\langle l, \nu\rangle \xrightarrow{t} \langle l, v + t\rangle$ *if* $\nu + t' \models I(l) \forall t' \in [0, t] \wedge \forall l \in L : U(l) = false$
- *action-transition:* $\langle l, v\rangle \xrightarrow{\alpha} \langle l', v'\rangle$ *iff* $(l, \alpha, \phi, Y, l') \in E$ *with* $v \models \phi$ *and* $\nu' = \nu[Y := 0]$ *and* $\nu' \models I(l')$

Single timed automata can be combined using parallel composition resulting in a network of timed automata. In the network, automata can communicate in two ways: *synchronously* via handshake communication (like in the process algebra *CCS* [12]) or in a broadcast manner. The sender in a broadcast communication can communicate with an abitrary number of receivers, namely all of those which are currently enabled for a communication. In the following, we will use synchronous communication as a means of synchronising the behaviour of components in the AUTOSAR architecture while we use broadcast for synchronisation with test automata modelling timing requirements.

To express properties on Timed Automata the query language *Timed Computation Tree Logic (TCTL)* is used. It allows specifying real-time constraints on Timed Automata, which can be checked in tools like *UPPAAL*[4] [13]. In TCTL, different types of formulas can be expressed: In *state formulas* properties on states can be specified, while *path formulas* quantify over paths or traces of the model [13,14].

3 Transformation of AUTOSAR Models

In this section we describe the transformation of AUTOSAR meta-model elements into timed automata. The AUTOSAR meta-model is very large. However, many model elements do not influence the dynamic behavior of the system. Furthermore, many specialized classes exist, but only for some of them the specified transformations are performed. Therefore we only give an introduction for timing relevant meta model elements and afterwards give a simplified formalization of the meta model. In this work we focus on the timing of ECUs and abstract from bus communication. As there is no formal semantics defined for AUTOSAR, we cannot prove the correctness of the transformations.

Timing on Application Layer. The AUTOSAR application layer consists of application software. Software is encapsulated in so-called *RunnableEntities* (abbreviated: *runnable*). For modeling timing behavior on application layer it is necessary to represent the runnables, variable accesses and their interconnections by appropriate timed automata. We abstract from the concept of software components and ports as it is not relevant for the timing whether two runnables in different software components are connected via ports or directly in a single software component as we assume that all software components are mapped onto a single ECU.

[4] http://www.uppaal.org.

Timing on RTE Layer. The RTE-Layer is a standardized interface for the software on application layer and is responsible for triggering runnables as specified in the operating systen, which is located on the basic software layer. The operating system has a scheduler and maintains the execution of resources by OSTasks. For this reason runnables have to be mapped onto OSTasks to specify the execution order of runnables. This is done in the RTE configuration using the so-called *RTEEventToTaskMapping*, which maps events, representing the triggering of a runnable, onto tasks.

Timing on Basic Software Layer. On the basic software layer AUTOSAR specifies many modules, which can be specified for every ECU. Most important for the runtime behavior are the modules which have influence on the execution order of the runnable entities. This is mainly the AUTOSAR operating system, which is based on the OSEK standard[5].

We consider the following parts of an AUTOSAR architecture during transformation.

Definition 3 (AUTOSAR Architecture). *The simplified formal* AUTOSAR architecture $AR = (R, C, VA, T, TRM)$ *consists of*

1. *a set of VariableAccess elements* VA,
2. *a set of RunnableEntities*

$$R \subseteq \{(VA_{read}, VA_{write}, wcet, bcet) \mid VA_{read} \subseteq VA, VA_{write} \subseteq VA, bcet \leq wcet\}$$

   with VA_{read} *a set of variable read accesses,* VA_{write} *a set of variable write accesses with* $VA_{read} \cap VA_{write} = \emptyset$, $wcet \in \mathbb{N}$ *the worst case and* $bcet \in \mathbb{N}$ *the best case execution time,*
3. *a set of AssemblyConnections* $C \subseteq \{(left, right) \mid left \in VA, right \in VA\}$, *which connect two variable access elements,*
4. *a set of periodically triggered tasks* T *with period* p *and*
5. *a Task-Runnable-Mapping* $TRM : R \rightarrow T$ *mapping runnables to operating system tasks.*

3.1 Transformation

For the verification of timing requirements in AUTOSAR, a mapping from AUTOSAR models onto timed automata was modelled, where the AUTOSAR model contains the software architecture and timing requirements, which are formulated as AUTOSAR timing extensions. The AUTOSAR software architecture is transformed into a network of timed automata, while each timing requirement is transformed into a test automaton and a TCTL-query (see Fig. 3). In the resulting overall network test automata and architecture automata communicate via broadcast channels.

For a given AUTOSAR model $AR = (R, AC, VA, T, TRM)$ a network of timed automata $\mathcal{N} = (A_1 \parallel .. \parallel A_n)$ is constructed. Below the transformations are described in a bit more detail, where we – due to lack of space – however cannot formally define all parts.

[5] http://osek-vdx.org/.

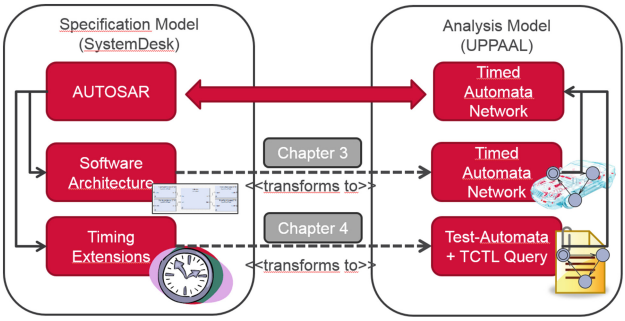

Fig. 3. Transformation of AUTOSAR models into a set of timed automata and TCTL queries

RunnableEntities. RunnableEntities represent the code fragments which are integrated into the architecture. Triggering is controlled by the RTE. Furthermore, runnables have access to a defined set of variables. Variables with reading access are read directly when a runnable is started, while write accesses are executed before termination[6]. Execution of runnable code requires time.

For every RunnableEntity in the analysis model a timed automaton is generated, which considers the variable accesses as well as the runtime behavior. In the case that all software components are executed on the same ECU, it is negligible whether the runnable entities communicate via interrunnable variables in a single software component or via ports. The generation of locations and transitions is therefore identical for ports and interrunnable variables.

For every RunnableEntity $r \in R$ with $r = (VA_{read}, VA_{write}, wcet, bcet)$ a timed automaton $\mathcal{A} = (L, B, B^*, X, I, U, E, I_{ini})$ is generated. Let $VA_{read} = \{r\_VA_{read_1}, \ldots, r\_VA_{read_n}\}$ be the set of read accesses (VA_{write} analogously). In the following, we use an arbitrary ordering 1 to n of these sets.

- Locations: $L = \{r\_ready\_loc, r\_running\_loc\} \cup \{r\_Va_{read}\_loc \mid Va_{read} \in VA_{read}\}$
  $\cup \{r\_Va_{write}\_loc \mid Va_{write} \in VA_{write}\}$,
- Handshake Communication: $B = \{r\_start, r\_finished\}$,
- Broadcast Communication: $B^* = \{r\_a_{read} \mid Va_{read} \in VA_{read}\}$
  $\cup \{r\_va_{write} \mid Va_{write} \in VA_{write}\}$
- Clocks: $X = \{x\}$, Invariants: $I(r\_running\_locc) = \{x <= wcet\}$,
- Urgency: $\forall Va \in VA_{read} \cup VA_{write} : U(r\_Va) = true$,
  $U(r\_ready) = false, U(r\_running) = false$,
- Edges: $E = \{(r\_ready, r\_start?, \emptyset, \{x\}, r\_Va_{read_1}),$
  $(r\_Va_{read_n}, r\_va_{read_n}!, \emptyset, \emptyset, r\_running)\} \cup$
  $\{(r\_Va_{read_j}, r\_va_{read_j}!, \emptyset, \emptyset, r\_Va_{read_{j+1}}) | 1 \leq j \leq |VA_{read}| - 1\} \cup$
  $\{(r\_Va_{write_j}, r\_va_{write_j}!, \emptyset, \emptyset, r\_Va_{write_{j+1}}) | 1 \leq j \leq |VA_{write}| - 1\} \cup$

[6] This is called implicit variable access and in this work only implicit access will be considered, while there is also an explicit access method where the access is not controlled by the RTE.

$\{(r\_running, r\_va_{write_1}!, \{x \geq bcet\}, r\_Va_{write_1})\} \cup$
$\{(r\_Va_{write_n}, r\_finished!, \emptyset, \emptyset, r\_ready)\}$,
– Initial location: $I_{ini} = r\_ready$.

The generated timed automaton consists of at least the locations *ready* and *runninng* (prefixed by the name of the runnable). The automaton is in location *ready* when the RunnableEntity is currently not running and in location *running* otherweise. Initially, the RunnableEntity is in location *ready*. Every implicit variable access of a RunnableEntity is also represented as a location. Identification of the access is done by signals on the transitions. These signals are not only used for synchronization, but also, if available, for existing test automata, which need to detect the data flow in the architecture. Therefore channels (for communication) are defined as *broadcast* channels.

Figure 4 exemplifies a transformed runnable with one incoming and two outgoing variable accesses. It shows the runnable *TssPreprocessing* located in the software component *IndicatorLogic* (see Fig. 2), which reads the raw turn switch sensor value *tss_value*, preprocesses it and writes its results in *tss_status*. Furthermore *wcet* and *bcet* are assumed to be 5 ms and 2 ms respectively.

AssemblyConnections. AssemblyConnections $C = (left, right)$ connect write and read accesses of variable access elements. For every AssemblyConnection a timed automaton is generated which describes the data flow between runnables. There is one location, and for the variable accesses $left$ and $right$, there is a transition to track the connections in the software architecture. Thus, for each AssemblyConnection $C = (left, right)$ we get a timed automaton $\mathcal{A} = (L, B, B^*, X, I, U, E, I_{ini})$ with:

– Locations: $L = \{ac\_start\}$, Signals: $B = \{left, right\}, B^* = \{\}$,
– Clocks: $X = \emptyset$, Invariants $I : \emptyset$, Urgency: $U(ac\_start) = false$,
– Edges: $E = \{(ac\_start, left?, \emptyset, \emptyset, ac\_start), (ac\_start, right?, \emptyset, \emptyset, ac\_start)\}$,
– Initial location: $I_{ini} = ac\_start$.

TaskRunnableMapping. For the correct execution order of runnables in the analysis model, a timed automaton A is generated for every OsTask. This automaton triggers the contained runnables in an OsTask in the defined order.

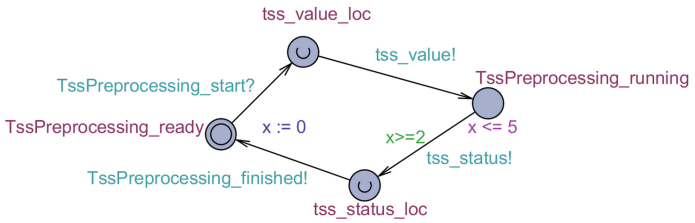

Fig. 4. Timed automata for the runnable preprocessing turn switch sensor values

The automata sends *Start*-signals to the receiving runnable automata. Afterwards the runnable is set to *running*-location and it leaves the *running*-location when the runnable automaton sends the *finish*-signal back to the runnable mapping automaton. Since no time passes between starting and stopping, the corresponding locations are marked as *urgent* locations.

Let T be the set of all OsTasks and for every OsTask $t \in T$, let $R_t = \{r \in R \mid TRM(r) = t\}$ be the set of all *RunnableEntities*, which are triggered by the OsTask t. Again we impose an arbitrary ordering on the set R_t, using indexes 1 to n. Then for every OsTask $t \in T$, a timed automaton A in the analysis model exists with

- $L = \{t\_ready, t\_running\} \cup \{t\_r\_start, t\_r\_stopped \mid r \in R_t\}$
- $B = \{t\_run, t\_processed\} \cup \{t\_r\_start, t\_r\_finished \mid r \in R_t\}, B^* = \{\}$,
- Clocks: $X = \{x\}$, Invariants: $I(t\_running) = \{x == 0\}$,
- Urgency: $\forall r \in R_t : U(t\_r\_finished) = true, U(t\_running) = true$,
- $E = \{(t\_ready, t\_run?, \emptyset, \emptyset, t\_running)$,
  $(t\_running, t\_r_1\_start!, \emptyset, \emptyset, t\_r_1\_running)$,
  $(t\_r_n\_stopped, t\_processed!, \emptyset, \emptyset, t\_processed)$,
  $\cup\{(t\_r\_stopped, t\_r\_start!, \emptyset, \emptyset, t\_r\_running)$,
  $(t\_r\_running, t\_r\_finished?, \emptyset, \emptyset, t\_r\_stopped) \mid r \in R_t\}$,
- Initial location: $I_{ini} = t\_ready$.

OS Tasks. Every AUTOSAR-based ECU includes an AUTOSAR-compliant OSEK operating system, which maintains the execution of *OsTasks* on the ECU. OSEK differentiates between *Basic-Tasks*, which can only be interrupted by the operating system itself, and *Extended-Tasks*. Extended tasks can be interrupted and set into *waiting* state. For this work we focus on basic tasks. Basic tasks have states *suspended*, *ready* and *running*. A task is in state *ready*, if it can be scheduled by the scheduler. If the scheduler selects the task for running, it is set in *running* state. After termination, but before the timing period is passed, the task is set to state *suspended*.

For every OsTask $t \in T$ a timed automaton A is generated:

- $L = \{t\_ready, t\_starting, t\_running, t\_terminating, t\_suspended\}$,
- $B = \{t\_startTask, t\_run, t\_processed, t\_terminateTask, t\_isNotReady\}$,
- $B^* = \{\}$, $X = \{x\}$,
- $I(t\_running) = \{x <= p\}, I(t\_suspended) = \{x <= p\}$,
- $U(ready) = false, U(starting) = true, U(running) = false, U(terminating) = true, U(suspended) = false$,
- $E = \{(t\_ready, t\_startTask?, \emptyset, \emptyset, t\_starting)$,
  $(t\_starting, t\_run!, \emptyset, \emptyset, t\_running)$,
  $(t\_running, t\_processed?, \emptyset, \emptyset, t\_terminating)$,
  $(t\_terminating, t\_terminateTask!, \emptyset, \emptyset, t\_suspended)$,
  $(t\_suspended, \phi, \{x == p\}, \{x\}, t\_ready)$,
  $(t\_suspended, t\_isNotReady!, \emptyset, t\_suspended)\}$,
- $I_{ini} = t\_ready$.

The behavior of an OsTask is modeled by generation of locations for *ready*, *running* and *suspended* and additional (urgent)-locations for sending and receiving multiple signals for synchronization with the *RunnableToTask-Mapping-automaton*. The OsTask starts in the *ready*-location and can be triggered by the Task Scheduler. By receiving the signal *startTask* the *EventToTaskMapping* is signaled and the OsTask is set to *running*. Afterwards the EventToTaskMapping is executed, i.e. all RunnableEntities have been executed, the signal *processed* is received and the signal *terminateTask* is sent to the *Scheduler*. The OsTask then stays in *suspended* until the period of the OsTask is due. In between the automaton only synchronizes via the signal *isNotReady* to the scheduler. Afterwards the OsTask is set back to *ready* and can again be executed by the *scheduler*.

4 AUTOSAR Timing Extensions

The transformations described before cover the behavior of the AUTOSAR system. To verify timing constraints on the system, the requirements also need to be formalized. To this end, for each timing requirement specified as AUTOSAR timing constraint, a test automaton as well as a TCTL-query for checking the requirement are created.

We start with explaining timing requirements. AUTOSAR Timing Extensions extends the AUTOSAR meta model with timing annotations for different model elements [15]. A *TimingExtension* contains a set of *TimingDescriptions* and *TimingConstraints*. *TimingDescriptions* are elements that describe events and event chains within a system, whereas *TimingConstraints* formulate timing requirements and timing guarantees for these events.

4.1 Timing Events

Formally, the set of Timing Events $E \subseteq (R \cup VA \cup T)$ is a subset of the AUTOSAR model elements, for which the dynamic behavior needs to be observed. Thus, runnables, variable accesses and tasks can be observed.

Requirements for Data Latency on Events. A *LatencyTimingConstraint* describes the latency requirement from the start to the event of a sequence of events. Formally, a LatencyTimingConstraint is defined as $lc = (chain, maximum)$ where

- $chain = (e_1, \ldots, e_n)$ is an ordered sequence of events,
- $maximum \in \mathbb{N}$ is the maximum time for the constraint.

In the transformation of the *TimingExtension* with *LatencyTimingConstraint* the event chain is transformed to a test automaton, which models the event chain as chain of locations. In between every location a transition is generated which receives the corresponding signal defined in the event chain. Verification of the required latency is achieved by a clock which measures the time spent in the event chain and which is reset when the event chain is due. Maximum latency

is checked by a TCTL-query which checks the maximum clock value in the test automaton. Hence for every *LatencyTimingConstraint* lc, a timed automaton A is generated as follows:

- Locations: $L = \{lc\_e | e \in chain\}$, Signals: $B^* = \{e | e \in chain\}$, Clocks: $X = \{x\}$,
- Invariants $I(lc\_e_1) = \{x \leq 1\}$,
- $E = \{(lc\_e_j, e_j?, \emptyset, \emptyset, lc\_e_{j+1}) | 1 \leq j < n-1\}$
  $\cup \{lc\_e_n, e_n, \emptyset, \{x\}, lc\_e_1 \cup \{(lc\_e_1, e_1?, \emptyset, \{x\}, lc\_e_1)\}$,
- Initial location: $I_{ini} = lc\_e_1$.

In the first location lc_e_1 (i.e. *before* the first event is received) the automaton cyclically resets its clock (implemented by a self-transition and invariant on lc_1) so that the clock value only exeeds 1, when the first event is received. Note that according to the definition of B^*, the generated signals are using broadcast communication. Additionally the TCTL-query $\varphi = AG(x < maximum)$ is generated. Here, AG requires the property to hold always on all paths.

Figure 5 shows a latency timing constraint automaton measuring the time from the start event when the turn switch sensor receives the raw signal to the bulb actuator which switches the indicator bulbs.

Requirements for Ordered Execution of Runnables. Requirements on the ordered execution of runnables are captured by the *ExecutionOrderConstraint*. An *ExecutionOrderConstraint* $eoc = (r_1, \ldots, r_n), r_i \subseteq R$, is defined by an ordered sequence of a subset of the available runnable entities for which the execution order is specified.

For every *ExecutionOrderConstraint* eoc a timed automaton is generated as follows:

- Locations: $L = \{r_i\_EOC\_started, r_i\_EOC\_finished \mid 1 \leq i \leq n\} \cup \{init, error\}$,
- Broadcast Communication: $B^* = \{r\_EOC\_start, r\_EOC\_finished \mid i = 1, \ldots, n\}$, Handshake Communication: $B = \{\}$,
- Clocks: $X = \{\}$, Invariants: I is true for all locations,
- Urgency: $U(r_n\_EOC\_finished) = true$,
- $E = \{init, r_1\_start?, \emptyset, \emptyset, r_1\_EOC\_started\} \cup$
  $\{r_i\_EOC\_started, r_i\_finished?, \emptyset, \emptyset, r_i\_EOC\_finished \mid i = 1, \ldots, n\} \cup$
  $\{r_i\_EOC\_finished, r_{i+1}\_start?, \emptyset, \emptyset, r_{r+1}\_EOC\_started \mid i = 1, \ldots, n\} \cup$
  $\{r_n\_EOC\_finished, \tau, \emptyset, \emptyset, init\}$
- $I_{ini} = init$.

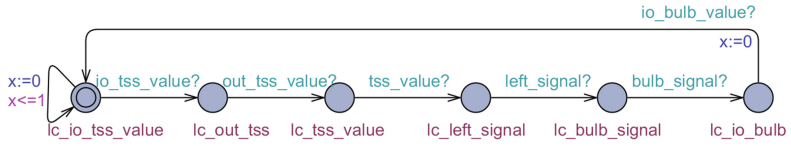

Fig. 5. Timed automaton of a latency timing constraint

Note that according to the definition of B^*, the generated signals are using broadcast communication. Furthermore, for every location $l \in L$, a TCTL-query $\varphi = AF(l)$ is generated. This property requires that on all paths of the system run every location is eventually visited eventually (i.e., the events are received in the specified order).

Requirements for Synchronized Execution of Events. A *Synchronization-TimingEvent* $sc = (scopeEvents, tolerance)$ consists of

- $scopeEvents \subseteq E$ describing the set of events, which have to occur only nearly simultaneously and
- $tolerance \in \mathbb{N}$ describing the maximum time which may occur between all $scopeEvents$, so that the execution can still be categorized as being simultaneous.

The requirement is fulfilled if $\forall e_i, e_j \in scopeEvents : |t_{e_i} - t_{e_j}| \leq tolerance$, where t_i is the time when event i occurs.

For every sc, a timed automaton is generated as follows:

- Locations: $L = \{sc\_init\}$, Signals: $B^* = \{e | e \in scopeEvents\} B = \{\}$,
- Clocks: $X = \{x\}$, Invariants: $I = \{\emptyset\}$, Urgency: U is false for all locations
- Edges: $E = \{sc\_init, e?, \emptyset, \emptyset, sc\_init\}$, $I_{ini} = \{sc\_init\}$.

Again, the generated signals are using broadcast communication. Furthermore, for the generated transitions functions are specified which are called each time the transition is taken. For each transition $e_i \in E$ the function e_i_receiving is called. In addition, local declarations are defined for each automaton as described in Listing 1.

Listing 1: Local declarations in UPPAAL

```
1  bool e_i_received = false;
2  void e_i_receiving()
3  {isRunning(); e_i_received = true; isCompleted();}
4
5  clock x;
6  bool running = false;
7
8  void isRunning()
9  {if(!running){x=0;running=true;}}
10
11 void isCompleted(){ if (e_1_received && ..e_n_received)
12 {e_i_received = false;
13 x=0; running=false;}}
```

Finally, a TCTL-query is generated as follows: $AG(running \implies x \leq tolerance)$. Figure 6 exemplifies the transformation of a Synchronization Timing Constraint which requires the runnables for the left and right actuator to be triggered synchronously. Analogously to the automaton the required local

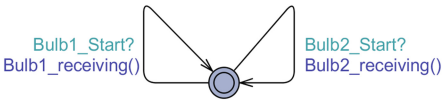

Fig. 6. Example for a SynchronizationTimingConstraint synchronizing the bulb lights

declarations are generated, i.e., two flags *Bulb1_received* and *Bulb2_received*, two functions *Bulb1_receiving* and *Bulb2_receiving*.

For the verification of the AUTOSAR architecture all generated automata $A_i = (L_i, B_i, X_i, I_i, E_i, I_{i_{ini}})$ are connected to a network of timed automata $\mathcal{N} = (A_1 \mid\mid .. \mid\mid A_n)$. Then a *TimingConstraint* T is fulfilled by the model, iff $(\mathcal{N} \mid\mid T) \models \varphi$, that is the automaton for a single timing constraint is connected to the network of timed automata representing the software architecture and the network is checked according to the specified TCTL-formula.

5 Implementation and Evaluation

The transformations were implemented as an independent tool, which uses the automation feature of SystemDesk® to retrieve AUTOSAR model informations. It includes separated components for model conversion and export. The exporting module comprises functionalities to compile an XML file out of the timed automata model, which is compatible to the UPPAAL [13] model checker.

The efficiency of the approach was evaluated by transforming three scenarios while measuring the time for model transformation and model checking via UPPAAL. The measurements were performed on an Intel i7-4810MQ @ 2.8 GHz with 16 GB RAM and Windows 7 Professional. UPPAAL version 4.0.13 was used with BFS search order, conservative state space reduction and DBM state space representation.

Table 1 shows the model sizes and runtime measurements for three different AUTOSAR models, namely a tutorial project, a model of a fueling system and the already mentioned model for direction indication. For each demo project at least one constraint of each type was modeled and verified. In Fig. 7, the runtime results split into transformation and constraint checking time are visualized. The

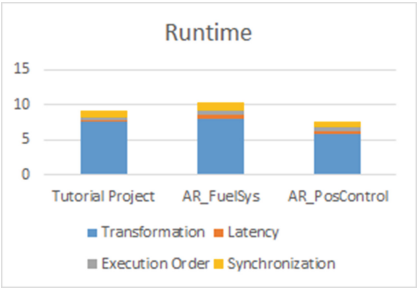

Fig. 7. Transformation and verification runtime

table given below gives exact numbers. Transformation and verification runtime is highest for the *AR_FuelSys* demo although it is not the biggest AUTOSAR model. The reason is that it contains more model elements for which timed automata have to be generated.

Table 1. Model size and runtime

Test system	Tut.Project	AR_FuelSys	AR_PosControl
AUTOSAR elements	748	723	503
Timed automata	26	38	23
Transformation time (s)	7.56	7.96	5.9
Latency constraint (s)	0.33	0.74	0.39
Execution order constraint (s)	0.4	0.53	0.5
Synchronization constraint (s)	0.95	1.13	0.94
Verification time $\sum$	9.24	10.36	7.73

The first results show that for these type of systems timing analysis is promising as the runtime is sufficiently low for real world use. Most of the time is spent in the transformation process. But as the transformation has polynomial runtime in the size of model elements, also larger models should be manageable.

6 Conclusion

In this work, an approach for the verification of timing requirements of AUTOSAR-based software architectures has been presented. Utilizing this method, timing requirements can be checked early and without access to source code. Only timing annotations (best case and worst case execution times) for runnable entites are required. They have to be introduced with the help of expert knowledge in a conservative fashion, or upper bounds for execution have to be figured out by static code analysis methods. For the verification of the AUTOSAR architecture existing tools for the verification of timed automata (like UPPAAL) can then be used.

By transforming AUTOSAR-architectures to timed automata a formal verification of timing requirements gets possible. The modeling of the AUTOSAR architecture and the required model elements for the analysis, however, have to be done manually. For example, timing requirements have to be specified. For this there is currently no tool available, which makes modeling time consuming and error prone. As future work, we will thus investigate how timing requirements can be precisely but easily (graphically) specified. Until now formal verification is only seldomly used in the software development process for automotive systems, because a successive application not only requires sound analysis methods, but also easy integration into existing development processes. Simplification of the formal specification and the quality analysis of timing requirements are thus crucial steps for the acceptance in industry.

References

1. AUTOSAR. http://www.autosar.org
2. Richter, K.: Compositional scheduling analysis using standard event models: the SymTA/S approach. Ph.D. thesis, Braunschweig (2005)
3. Feiertag, N., Richter, K., Nordlander, J., Jonsson, J.: A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In: IEEE Real-Time Systems Symposium 2008, vol. 29 (2008)
4. Perathoner, S., Wandeler, E., Thiele, L., Hamann, A., Schliecker, S., Henia, R., Racu, R., Ernst, R., Harbour, M.G.: Influence of different system abstractions on the performance analysis of distributed real-time systems. J. Des. Autom. Embed. Syst. **13**(1–2), 27–49 (2009)
5. Neumann, S., Kluge, N., Wätzoldt, S.: Automatic transformation of abstract autosar architectures to timed automata. In: Proceedings of the 5th International Workshop on Model Based Architecting and Construction of Embedded Systems, ACES-MB 2012, pp. 55–60. ACM, New York (2012)
6. Gehrke, M., Nawratil, P., Niggemann, O., Schäfer, W., Hirsch, M.: Scenario-based verification of automotive software systems. In: Giese, H., Rumpe, B., Schätz, B. (eds.) Dagstuhl-Workshop MBEES. Daghstuhl-Workshop MBEES, vol. 2, pp. 35–42. TU Braunschweig, Institut für Software Systems Engineering (2006)
7. Scheickl, O., Ainhauser, C., Gliwa, P.: Tool support for seamless system development based on autosar timing extensions. In: Embedded Real-Time Software and Systems 2012 (2012)
8. Heckmann, R., Ferdinand, C.: Worst-case execution time prediction by static program analysis. In: Jacquart, R. (ed.) Building the Information Society. IFIP Advances in Information and Communication Technology, vol. 156, pp. 377–383. Springer, Heidelberg (2004)
9. AUTOSAR: Layered software architecture (2013). http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/general/auxiliary/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf
10. Alur, R., Dill, D.: A theory of timed automata. Theor. Comput. Sci. **126**, 183–235 (1994)
11. Olderog, E.R., Dierks, H.: Real-Time Systems: Formal Specification and Automatic Verification (2008)
12. Milner, R.R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Berlin (1980)
13. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
14. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
15. AUTOSAR: Autosar timing extensions template (2013). http://www.autosar.org/fileadmin/files/releases/4-2/methodology-and-templates/templates/standard/AUTOSAR_TPS_TimingExtensions.pdf