Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Automatic modelling and verification of AUTOSAR architectures[☆]

Miaomiao Zhang[a], Yu Teng[a], Hui Kong[b], John Baugh[c], Yu Su[a], Junri Mi[d], Bowen Du[a,*]

[a] School of Software Engineering, Tongji University, 200092, China
[b] Huawei Corporation, 200000, China
[c] Department of Civil, Construction, and Environmental Engineering, North Carolina State University, NC 27695, USA
[d] Alibaba Corporation, 310000, China

ARTICLE INFO

Article history:
Received 4 September 2022
Received in revised form 4 January 2023
Accepted 7 March 2023
Available online 11 March 2023

Keywords:
Vehicle electronic system
AUTOSAR
Formal modelling
Timed automata
Verification

ABSTRACT

AUTOSAR (AUTomotive Open System ARchitecture) is a development partnership whose primary goal is the standardization of basic system functions and functional interfaces for electronic control units in automobiles. As an open specification, its layered software architecture promotes the interoperability of real-time embedded vehicle systems and components. It also opens up the possibility of formal modelling and verification approaches, centred around the specification, that can be used to support analysis in the early stages of design. In this paper, we describe a methodology and associated tool, called A2A, that automatically models systems defined by the AUTOSAR specifications as timed automata, and then verifies their timing properties using UPPAAL. It contains 22 groups of timed automata templates, together with two auxiliary test templates, that model the AUTOSAR architecture and timing properties, allowing time-related behaviours to be extracted from the three-layer architecture, i.e., the AUTOSAR Software, AUTOSAR Runtime Environment, and Basic Software layers, and templates to be automatically instantiated. The timing properties are specified using timed computation tree logic (TCTL) in UPPAAL to verify the system model. We demonstrate the capabilities of the methodology by applying it to an AUTOSAR architecture that describes an internal vehicle light control system, thereby showing its effectiveness.

1. Introduction

Automobile manufacturers continue to incorporate new safety, performance, and entertainment features in modern vehicle electronics—with thousands of chips, millions of lines of code, and more than a hundred electronic control units (ECUs) found in many of today's cars (Sheng et al., 2015; Charette, 2021; Gu et al., 2016; Alam et al., 2019; Alladi et al., 2020). Writing real-time software components to control ECUs, and ensuring that they work together cooperatively and reliably, is therefore a growing challenge. To address this concern, the AUTOSAR specification defines a standard architecture for interoperability that includes application interfaces and basic software modules (Yamili and Kathiresh, 2021; Fürst et al., 2009; Fennel et al., 2006; Fürst and Bechter, 2016), which can be used to develop real-time embedded vehicle systems. As a result, software developed according to the AUTOSAR standard can operate across a range of different vehicles and ECUs, saving resources and improving the reusability of the software (Menard et al., 2020; Kotur et al., 2020; Jelecevic and Minh, 2019; Garcia and Olmedo, 2020).

Although AUTOSAR defines a standard software architecture for programming ECUs, its use alone cannot guarantee that software developed within its framework is safe and free of defects (Nasser and Ma, 2019; Piper et al., 2015; Tucci-Piergiovanni et al., 2011). Attention to the correctness concerns raised by increasingly complex and interrelated functions in vehicle systems is still necessary to ensure personal safety and minimize the likelihood of property loss (Luong et al., 2017; Zhang et al., 2019). As a result, the behaviours of AUTOSAR software must be analysed and verified to guarantee that vehicle ECUs perform their intended functions safely and correctly.

Analysis tools like SystemDesk provide simulation capabilities to test AUTOSAR software and promote correctness (Neumann et al., 2012). However, the tool does not support the analysis of non-functional properties like timing, and because it is based on simulation, it is unsuitable for finding corner cases or errors early in the development process (Beringer and Wehrheim, 2016; Sarikhada and Shah, 2020). In principle, verification during the development phase can nevertheless incorporate timing attributes as well, potentially reducing costs and improving safety. These benefits can be achieved through formal modelling and analysis of AUTOSAR components in the early phases of development. Examples of doing so include the coarse-grained modelling of tasks and so-called *RunnableEntities*, which we later describe,

[☆] Editor: Earl Barr.
[*] Corresponding author.
E-mail address: bowendu@tongji.edu.cn (B. Du).

https://doi.org/10.1016/j.jss.2023.111675
0164-1212/© 2023 Elsevier Inc. All rights reserved.

in the Basic Software layer and the AUTOSAR Software layer to verify the timing attributes (Neumann et al., 2012; Beringer and Wehrheim, 2016). Other aspects, however, including communication, execution order, different task types (periodic and sporadic), and different scheduling strategies (preemptive and non preemptive) are as yet unincorporated in our study, and remain the subject of future work.

In this paper, we propose a methodology that can automatically convert the AUTOSAR architecture, from top to bottom in a fine-grained manner, into a series of timed automata and verify whether the timing constraints are satisfied using UPPAAL. Due to the complexity of the AUTOSAR architecture, modelling it manually is both cumbersome and error-prone. As a result, we have developed a tool called *A2A* that can automatically perform the modelling process and verify the timing properties using UPPAAL. Therefore, developers of AUTOSAR software can make use of this methodology to ensure that their implementations of ECU software meet the required timing constraints that are specified. Finally, as a case study, we show how the methodology can be used to verify the timing constraints of an internal vehicle light control system.

The main contributions we report in the paper are as follows:

- First, a tool A2A is developed to automatically extract the configuration information (scenarios and parameters, detailed in Section 4) of time-related behaviours from the AUTOSAR architecture conforming to an arXML description.
- Second, an algorithm is proposed to automatically construct time-related behaviours of the AUTOSAR architecture into an interconnected series of timed automata by A2A. That is, we provide a set of timed automata templates[1] to characterize behaviours and add them to the library of A2A. Then, the A2A tool instantiates the timed automata templates based on the extracted configuration information to automate the entire modelling process. The generated timed automata can work collaboratively to realize the dynamic interactive behaviours of the AUTOSAR architecture following the AUTOSAR specification.
- Third, a verification approach based on the generated interconnected timed automata is presented. To verify the timing constraints more conveniently, we first construct a series of auxiliary timed automata test templates[1] according to the definitions of the timing constraints in the AUTOSAR specification. All templates are added to the tool library of A2A. The A2A tool then collects timing constraint information from the user and instantiates timed automata test templates according to the specified constraint information automatically. According to the specified constraint information, the timing constraints will be described by the timed computation tree (TCTL) language and verified in UPPAAL.
- Finally, the AUTOSAR architecture of an internal vehicle light control system is modelled and verified to demonstrate performance evaluations and show the effectiveness of the proposed methodology.

The remainder of the paper is organized as follows. In Section 2, the system architecture of AUTOSAR and the verification tool UPPAAL (Behrmann et al., 2004, 2006) are described. We characterize related work in Section 3. The overall methodology undertaken in this study is introduced in Section 4. The extraction of configuration information from the AUTOSAR architecture is described in Section 5. The method of automatically modelling the AUTOSAR architecture is presented in Section 6, and Section 7 describes the verification of timing constraints. The performance of the proposed methodology is demonstrated in Section 8. Finally, we conclude the paper in Section 9.

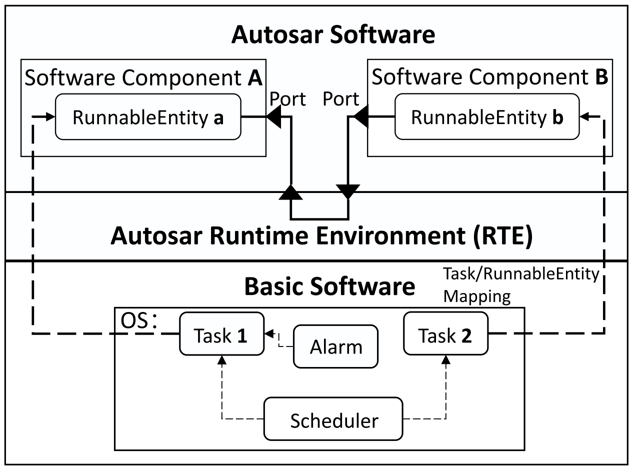


Fig. 1. The AUTOSAR layered architecture.

2. Background

2.1. AUTOSAR

AUTOSAR is an open and standardized software architecture for automotive ECUs. As shown in Fig. 1, it consists of three layers, from top to bottom: AUTOSAR Software, AUTOSAR Runtime Environment (RTE), and Basic Software (Anon, 2021).

The AUTOSAR Software layer contains the actual control software, which is constructed in the form of software components. AUTOSAR software components can be further divided into the atomic software component, the composition software component, and the parameter software component (Anon, 2021). Of these, the atomic software component cannot be structurally decomposed but can integrate the internal behaviours of the software. The composition software component, in contrast, can aggregate existing software components by composing them in a connector. In addition, the parameter software component provides calibration parameters in other software components, which does not influence the time-related behaviours and is not considered in this work.

Within an atomic software component, the *RunnableEntity* associated with the implementation realizes the behaviours of the component. In addition, each software component communicates with other software components through its ports. Each pair of ports that communicates is connected through a composition component. The communication paradigms between software components include the following:

- Sender–receiver communication is used to realize data transmission among the software components.
- Client–server communication is when a *client* RunnableEntity invokes a *server* RunnableEntity to execute an operation.
- Trigger event communication is when one RunnableEntity triggers another RunnableEntity.

The RTE controls communication between the ports of two software components in the AUTOSAR Software layer. In addition, the RTE also manages the cooperation between the AUTOSAR Software and Basic Software layers (Anon, 2021). For example, the RTE is responsible for mapping the RunnableEntities to tasks.

The fundamental role of the Basic Software layer is its responsibility for managing the hardware resources and the provision of common sources for the application software. The Basic Software layer provides services, e.g., in the form of drivers, for accessing

[1] https://github.com/Tongji-lab/Autosar/tree/main/AutosarTemplates

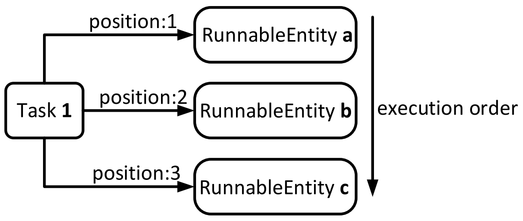


Fig. 2. Three RunnableEntities mapped onto one task.

the hardware as well as the operating system (OS) functionalities (Yamili and Kathiresh, 2021; Anon, 2021). Here, the OS mainly contains the alarm, the task, and the scheduler (Anon, 2021). Among them, the alarm is responsible for periodically activating tasks. In addition, the scheduler controls the running of the task according to the scheduling strategy. For the task, it implements the RunnableEntities mapped onto it. Fig. 2 shows a case where three RunnableEntities map onto one task, Task **1**. Each RunnableEntity mapped onto Task **1** has a specific position. If more than one RunnableEntity needs to be executed, they will be executed in the order of their positions in Task **1**.

A high-level overview of the collaborative work between the AUTOSAR Software and Basic Software layers can also be seen in Fig. 1. There is a RunnableEntity **a** in software component **A**, and another RunnableEntity **b** in software component **B**. The RunnableEntity **a** is mapped onto a periodic task, Task **1**, and the RunnableEntity **b** is mapped onto a sporadic task, Task **2**. To realize the periodic invocation from **a** to **b**, the entire process consists of the following steps:

1. The alarm records the time and activates Task **1** periodically. After Task **1** is activated, it waits to be scheduled by the scheduler.
2. When the scheduler schedules Task **1**, Task **1** starts to run.
3. Task **1** starts the RunnableEntity **a** according to the position of the RunnableEntity **a** mapped on it.
4. The RunnableEntity **a** can inform the RTE to invoke the RunnableEntity **b**.
5. The RTE receives the information and then activates the RunnableEntity **b**.
6. Task **2**, containing the RunnableEntity **b**, is activated.
7. When Task **2** is scheduled by the scheduler, it starts to run.
8. Task **2** starts the RunnableEntity **b** according to the position of the RunnableEntity **b** mapped on it.

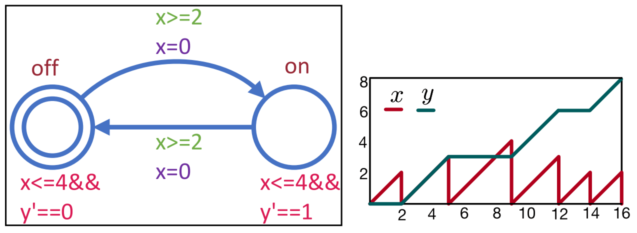
The configuration information from the AUTOSAR Software layer, the RTE layer, and the Basic Software layer is saved in an arXML source file. Therefore, this arXML source file is the starting point of our work, and so we make use of the AUTOSAR Tool Platform (Artop) (Voget, 2010) to parse arXML source files of the AUTOSAR architecture.

2.2. UPPAAL

We model and analyse the time-related behaviours of the AUTOSAR architecture using UPPAAL, a toolbox for verifying real-time systems modelled as networks of timed automata.

A timed automaton is a finite automaton—a graph containing a finite set of nodes or locations and a finite set of labelled edges—extended with real-valued variables. Formally, a timed automaton (Bengtsson and Yi, 2003; Alur and Dill, 1994) is a tuple $\mathbf{A} = (L, l_0, X, \Sigma, E, I)$, where

- L is a set of locations,
- $l_0 \in L$ is the initial location,
- X is a set of clocks and $\Phi(X)$ (like $x \leq 10$) represents the set of clock constraints,
- Σ is a set of actions,
- $E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset, and
- $I : L \to \Phi(X)$ assigns invariants to locations.



(a) (b)

Fig. 3. A timed automaton with stopwatches (Yan et al., 2018). (a) Timed automaton with stopwatches. (b) Simulation.

In addition, we also employ stopwatch automata (SWA), an extension that adds stopwatches to timed automata (Cassez and Larsen, 2000). In timed automata with stopwatches, clocks can be assigned to a rate of zero or one in a location. When a clock is assigned to rate *zero*, it freezes and keeps time from advancing. When the clock is assigned to rate *one*, it resumes from the last frozen point. The model in Fig. 3(a) is a timed automaton using a clock x to enforce the switch between two locations **off** and **on**, where **off** is the initial location. A stopwatch y, which is running only in location **on**, is introduced to measure the accumulated time in **on**. Therefore, the value of y is left unchanged in location **off**, as shown in Fig. 3(b) (Yan et al., 2018).

The UPPAAL modelling language extends timed automata with some additional features. For example, the automata synchronize using channels declared as c. In this case, an edge labelled *c!* is a sender, and another edge labelled *c?* is a receiver. If multiple receivers can respond to a sender in the current state, one receiver is chosen non-deterministically to communicate with the sender. Asynchronous communication is achieved using broadcast channels. In this case, a sender can communicate with zero, one, or multiple receivers. Therefore, broadcast sending is never blocked, and any receiver that can respond to the sender in the current state must communicate. In addition, for locations, UPPAAL supports urgent locations and committed locations, in which no delay is allowed. Furthermore, if a timed automaton is in a committed location, represented by Ⓒ in UPPAAL, the only transitions permitted are those from the committed location.

To verify that a model satisfies a property, a requirement specification is expressed in a formally well-defined and machine-readable language. UPPAAL uses a simplified version of timed computation tree logic (TCTL) as the query language. In TCTL, the query language consists of path formulae and state formulae. State formulae describe individual states, whereas path formulae quantify over paths or traces of the model (Behrmann et al., 2004).

3. Related work

Research on AUTOSAR properties and timing characteristics broadly falls within two categories: (1) modelling and informal analysis of time-related behaviours, and (2) formal verification of specific properties.

In the first category, numerous projects have focused on the modelling of time-related behaviours and the analysis of execution time and schedulability. Such studies have included, for instance, the use of modelling languages like Simulink (Chen et al., 2018), as well as discrete-event system specifications (Denil et al., 2017), where a simulation model is constructed for Autosar-based electronic control units that are connected by a communication bus. In other work (Klobedanz et al., 2010), the Timing Augmented Description Language (TADL) is applied to timing modelling and analysis in a case study on speed-adaptive steer-by-wire systems. A substantial amount of work has addressed task schedulability. One such study (Anssi et al., 2011) describes basic features of an analyzable Autosar model required by scheduling analysis. Zhao et al. (2017) present design optimization approaches for Autosar models with preemption thresholds and mixed-criticality scheduling. Other studies (Copic et al., 2020; Stegmeier et al., 2017) describe the mapping mode of RunnableEntities and the parallelism of RunnableEntities needed to shorten worst-case execution times. Yan and Guo (2019) use transition graphs to model schedule tables and propose an algorithm to analyse schedulability by checking all schedule scenarios. Safety and progress guarantees, such as deadlock checking, the satisfaction of latency time constraints (LTCs), and periodic event triggering constraints (PETCs) (Anon, 2021), are outside of the scope of these studies.

In the second category, formal verification, several methods have been proposed to verify timing constraints and guarantee safety. Zhu et al. (2013) focus on the timing properties of Autosar and propose an automatic verification framework based on rewriting logic to analyse the timing behaviours, though it only concerns about the reduced OS. One study Choi (2018) suggests a pattern-based framework that can be used to generate configurable formal OS and test models. However, it also only considers OS. Another study Yan et al. (2018) presents a method for checking safety-critical autonomous control systems that can tolerate transient faults, where a three-layer system model is given using timed automata, so that system behaviours can be evaluated. Nevertheless, it does not consider the RTE layer, and it simplifies the diverse and complex communication mechanisms in Autosar software instead of modelling message passing between tasks. Neumann et al. (2012) and Beringer and Wehrheim (2016) model timed automata for three layers of the Autosar architecture and determine best-case and worst-case execution times (BCET and WCET), though they do not discuss fine-grained modelling of the communication behaviours between RunnableEntities and preemptive scheduling strategy, potentially affecting the verification results. Not addressed in these studies is the automatic transformation from a standard arXML document to formal models, though Neumann et al. (2012) mention it without offering details.

4. Methodology

4.1. Time-related behaviours considered

In this work, we consider various time-related behaviours in Autosar architectures. Because different scenarios lead to different time-related behaviours, we consider the time-related behaviours within each scenario. For instance, there are different communication scenarios between software components in the Autosar Software, such as different ways to invoke the server, and different scheduling scenarios in the Basic Software, such as different scheduling strategies.

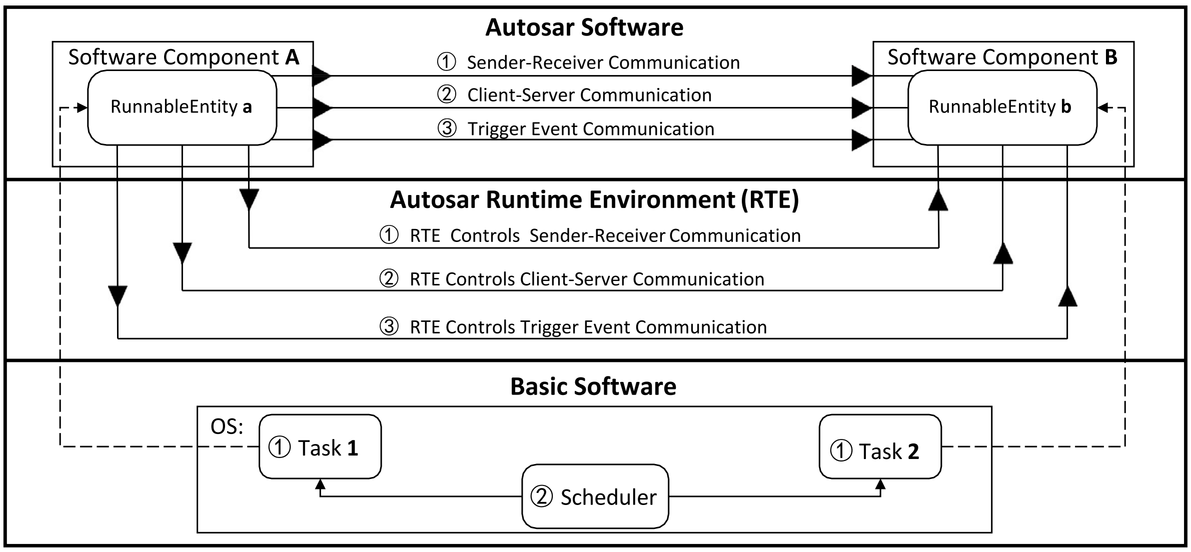
As shown in Fig. 4, we consider 22 scenarios in total from the Autosar Software, Autosar Runtime Environment (RTE), and Basic Software layers. In the Autosar Software layer, eight scenarios in three communication paradigms are considered, including four scenarios that involve client–server communication. For instance, Scenario (5) is the case in which a server RunnableEntity runs non-concurrently, and a client RunnableEntity synchronously invokes the server RunnableEntity running non-concurrently with a timeout monitoring mechanism. In addition, there are ten scenarios for the RTE to control the communication. For instance, to maintain sender–receiver communication, seven scenarios are distinguished that handle three different data transmission methods. In the Basic Software layer, additional scenarios are considered, including preemptive and non preemptive scheduling strategies, and both periodic and sporadic tasks.

4.2. Process of the proposed methodology

Our overall workflow of the methodology, which automatically models and then verifies the Autosar architecture, is shown in Fig. 5. The steps are as follows:

1. Because the scenarios contained in the Autosar architecture are defined in an arXML source file, a tool we developed called A2A, written in Java, extracts the scenarios and parameters from a given arXML source file of the Autosar architecture automatically. Within each scenario, parameters are extracted that characterize, quantitatively, the various time-related behaviours and dependencies, such as the length of the queue storing data, the period of each task and the timeout value. The arXML source file, before it is input to the A2A tool, is parsed by a software development toolkit (SDK) in the Artop.
2. An arXML source file normally consists of a number of scenarios, whose main behaviours are illustrated in the Autosar specification (Anon, 2021). Therefore, for each scenario, according to the specification, we need to construct several timed automata templates to describe the scenario. The number of templates for each scenario is given in Fig. 4. All of the constructed timed automata templates are added to the tool library of A2A.
3. The A2A tool selects the timed automata templates built for the extracted scenarios. Then, the execution time information specified by the user is collected. Finally, A2A instantiates the selected timed automata templates according to the extracted parameters and execution time information.
4. Based on explanations of the timing constraints in the Autosar specification, auxiliary timed automata test templates are constructed for verifying the timing constraints and also added to the tool library.
5. According to the extracted scenarios and parameters, A2A collects timing constraints from the user. A2A automatically instantiates timed automata test templates according to the collected timing constraints information.
6. Finally, users can perform TCTL queries in Uppaal to determine whether timing constraints are satisfied by the timed automata of the Autosar architecture.

When a preemptive strategy or wait point appears in arXML files, we employ stopwatch automata to model the corresponding Autosar architecture, which may lead to inconclusive results in some situations: for verification, an over-approximation technique is chosen to check safety properties, which state that something bad never happens (Behrmann et al., 2004). Therefore, if the result of performing a check is 'yes', we can be certain the architecture meets its timing constraints. However, if the result is 'no', the result is inconclusive, since the over-approximation itself may have introduced extra states that do not actually exist in the real architecture. In other words, a check showing that timing requirements are met is *always* conclusive, but for a negative result, in cases where stopwatch automata are employed and timing


Autosar Software
Software Component A
RunnableEntity a
① Sender-Receiver Communication
② Client-Server Communication
③ Trigger Event Communication
Software Component B
RunnableEntity b
Autosar Runtime Environment (RTE)
① RTE Controls Sender-Receiver Communication
② RTE Controls Client-Server Communication
③ RTE Controls Trigger Event Communication
Basic Software
OS:
① Task 1
① Task 2
② Scheduler

Layer	Categary	Scenarios	#Templates
Autosar Software	①	(1) Sender and receiver access data implicitly	2
		(2) Sender and receiver access data explicitly	2
		(3) Receiver is blocked until data is received	2
	②	(4) Client invokes concurrent server	2
		(5) Client invokes non-concurrent server synchronously with timeout monitoring	2
		(6) Client invokes non-concurrent server synchronously without timeout monitoring	2
		(7) Client invokes non-concurrent server asynchronously	2
	③	(8) Trigger event communication	2
RTE	①	(9) RTE stores data in a queue	1
		(10) RTE stores data in a queue and activates a RunnableEntity when data is sent	1
		(11) RTE stores data in a queue and activates a RunnableEntity when data is received	1
		(12) RTE stores data in a queue and wakeups receiver when data is received	1
		(13) RTE stores data in a single data set	1
		(14) RTE stores data in a single data set and activates a RunnableEntity when data is sent	1
		(15) RTE stores data in a single data set and activates a RunnableEntity when data is received	1
	②	(16) Client and server are in the same partition	3
		(17) Client and server are in the different partitions	4
	③	(18) RTE stores triggers in a queue	1
Basic Software	①	(19) Task is periodic	1
		(20) Task is non-periodic	1
	②	(21) OS adopts the preemptive scheduling strategy	1
		(22) OS adopts the non preemptive scheduling strategy	1

Fig. 4. Considered scenarios of time-related behaviours in Autosar architecture.

requirements *appear* not to have been met, further scrutiny may be required to determine whether the counterexample produced is valid.

In experiments we have performed, such as those described in Section 8, the frequency of such inconclusive results varies with timing constraints and parameters in the Autosar architecture, such as the execution time of RunnableEntities. When the result is inconclusive, one should carefully examine the counterexample given by the model-checking tool or adjust the scenarios or parameters of the architecture to get a 'yes' result. Alternatively, developers and researchers may work together to extend data structures of difference bounded matrices (DBMs) to encode exactly a stopwatch automaton and to perform an exact reachability analysis (Cassez and Larsen, 2000). Here, DBMs offer canonical representations for constraint systems and a DBM representation is in fact a weighted directed graph where the vertices correspond

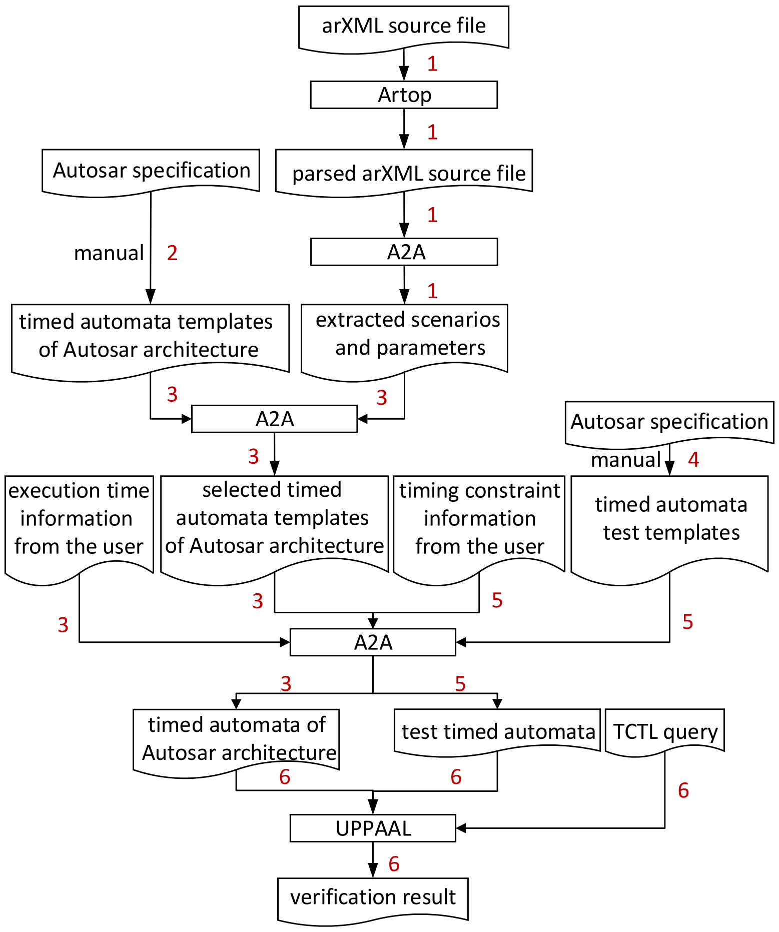

arXML source file
1
Artop
1
Autosar specification
parsed arXML source file
manual
2
1
A2A
1
timed automata templates of Autosar architecture
extracted scenarios and parameters
3
3
A2A
Autosar specification
3
manual
4
execution time information from the user
selected timed automata templates of Autosar architecture
timing constraint information from the user
timed automata test templates
3
5
3
5
A2A
3
5
timed automata of Autosar architecture
test timed automata
TCTL query
6
6
6
UPPAAL
6
verification result

Fig. 5. The process of the proposed methodology.

to clocks and the weights on the edges stand for the bounds on the differences between pairs of clocks (Larsen et al., 1997).

To put into context the role of over-approximation techniques, it is worth noting that they are used only when stopwatch automata are present. That is, if the scheduling strategy is non-preemptive and no wait points are involved in an AUTOSAR architecture, the verification results are always conclusive, since no stopwatch automata are used.

5. Extracting scenarios and parameters

In this step, according to the names and meanings of labels in the arXML source file defined by the AUTOSAR specification, pertinent labels for scenarios and parameters have been obtained and encoded in the A2A tool.

```
...
<SENSOR-ACTUATOR-SW-COMPONENT-TYPE>
  <SHORT-NAME>ClientSWC</SHORT-NAME>
  <PORTS>
    <R-PORT-PROTOTYPE>
      <SHORT-NAME>ClientPort</SHORT-NAME>
      <REQUIRED-COM-SPECS>
        <CLIENT-COM-SPEC>
          <OPERATION-REF DEST="CLIENT-SERVER-OPERATION">.../
                Operation
          </OPERATION-REF>
        </CLIENT-COM-SPEC>
      </REQUIRED-COM-SPECS>
      <REQUIRED-INTERFACE-TREF
      DEST="CLIENT-SERVER-INTERFACE">.../CSInterface</REQUIRED-
            INTERFACE-TREF>
```

```
...
        <RUNNABLE-ENTITY>
          <SHORT-NAME>ClientRunnableEntity</SHORT-NAME>
          <SERVER-CALL-POINTS>
            <SYNCHRONOUS-SERVER-CALL-POINT>
              <SHORT-NAME>ClientCallPoint</SHORT-NAME>
              <OPERATION-IREF>
                <CONTEXT-R-PORT-REF DEST="R-PORT-PROTOTYPE">
                  .../ClientPort</CONTEXT-R-PORT-REF>
                <TARGET-REQUIRED-OPERATION-REF
                DEST="CLIENT-SERVER-OPERATION">
                  .../Operation</TARGET-REQUIRED-OPERATION-REF>
              </OPERATION-IREF>
              <TIMEOUT>0.001</TIMEOUT>
...
<COMPOSITION-SW-COMPONENT-TYPE>
  <SHORT-NAME>Composition</SHORT-NAME>
  <COMPONENTS>
    <SW-COMPONENT-PROTOTYPE>
      <SHORT-NAME>ClientSWCinComposition</SHORT-NAME>
      <TYPE-TREF DEST="SENSOR-ACTUATOR-SW-COMPONENT-TYPE">.../
           ClientSWC
      </TYPE-TREF>
    </SW-COMPONENT-PROTOTYPE>
    <SW-COMPONENT-PROTOTYPE>
      <SHORT-NAME>ServerSWCinComposition</SHORT-NAME>
      <TYPE-TREF DEST="SERVICE-SW-COMPONENT-TYPE">.../ServerSWC</
           TYPE-TREF>
    </SW-COMPONENT-PROTOTYPE>
  </COMPONENTS>
  <CONNECTORS>
    <ASSEMBLY-SW-CONNECTOR>
      <SHORT-NAME>Connector1</SHORT-NAME>
      <PROVIDER-IREF>
        <CONTEXT-COMPONENT-REF DEST="SW-COMPONENT-PROTOTYPE">
          .../ServerSWCinComposition</CONTEXT-COMPONENT-REF>
        <TARGET-P-PORT-REF DEST="P-PORT-PROTOTYPE">.../ServerPort
        </TARGET-P-PORT-REF>
      </PROVIDER-IREF>
      <REQUESTER-IREF>
        <CONTEXT-COMPONENT-REF DEST="SW-COMPONENT-PROTOTYPE">
          .../ClientSWCinComposition</CONTEXT-COMPONENT-REF>
        <TARGET-R-PORT-REF DEST="R-PORT-PROTOTYPE">.../ClientPort
        </TARGET-R-PORT-REF>
...
<SERVICE-SW-COMPONENT-TYPE>
  <SHORT-NAME>ServerSWC</SHORT-NAME>
  <PORTS>
    <P-PORT-PROTOTYPE>
      <SHORT-NAME>ServerPort</SHORT-NAME>
      <PROVIDED-COM-SPECS>
        <SERVER-COM-SPEC>
          <OPERATION-REF DEST="CLIENT-SERVER-OPERATION">.../
               Operation
          </OPERATION-REF>
          <QUEUE-LENGTH>1</QUEUE-LENGTH>
        </SERVER-COM-SPEC>
      </PROVIDED-COM-SPECS>
      <PROVIDED-INTERFACE-TREF DEST="CLIENT-SERVER-INTERFACE">
      .../CSInterface</PROVIDED-INTERFACE-TREF>
...
        <OPERATION-INVOKED-EVENT>
          <SHORT-NAME>OIEvent</SHORT-NAME>
          <START-ON-EVENT-REF DEST="RUNNABLE-ENTITY">
            .../ServerRunnableEntity</START-ON-EVENT-REF>
          <OPERATION-IREF>
            <CONTEXT-P-PORT-REF DEST="P-PORT-PROTOTYPE">
            .../ServerPort</CONTEXT-P-PORT-REF>
            <TARGET-PROVIDED-OPERATION-REF
            DEST="CLIENT-SERVER-OPERATION">
              .../Operation</TARGET-PROVIDED-OPERATION-REF>
...
        <RUNNABLE-ENTITY>
          <SHORT-NAME>ServerRunnableEntity</SHORT-NAME>
          <CAN-BE-INVOKED-CONCURRENTLY>false
...
<CLIENT-SERVER-INTERFACE>
  <SHORT-NAME>CSInterface</SHORT-NAME>
  <OPERATIONS>
    <CLIENT-SERVER-OPERATION>
      <SHORT-NAME>Operation</SHORT-NAME>
...
```

Listing 1: The AUTOSAR Software layer of an arXML source file.

A2A takes the arXML source file as input, and uses Artop's SDK to parse and identify the arXML source file. Listing 1 shows a fragment, that is, the AUTOSAR Software layer of an arXML source file, which includes three layers from the AUTOSAR architecture, and Fig. 6 shows a portion of the result of the file after being parsed.

After being parsed, the values of selected labels are extracted. Fig. 7 shows a flow extraction procedure starting from the *AssemblySwConnector*, which signifies a connector, and allows a pair of ports communicating with each other to be extracted. For each port, the *RequiredInterface* and *ProvidedInterface* labels are extracted to determine the communication paradigm. Then, labels like *SynchronousServerCallPoint* and *CanBeInvokedConcurrently* are extracted to determine the scenario of the invocation in client–server communication. Parameters that determine quantitative behaviours are also extracted, such as *Timeout*, indicating a time limit on the communication.

Fig. 8 shows the flow of extracting scenarios and parameters from the AUTOSAR Software layer of Fig. 6. According to **Connector1**, a pair of ports **ClientPort** and **ServerPort** communicating with each other is found. Then, the **CSInterface** determines the communication paradigm between the two ports to be client–server communication. In addition, **ClientCallPoint** signifies a synchronous invocation method and a configured timeout value of **0.001** seconds. In other words, the classification is determined to be Scenario (5), as previously shown in Fig. 4.

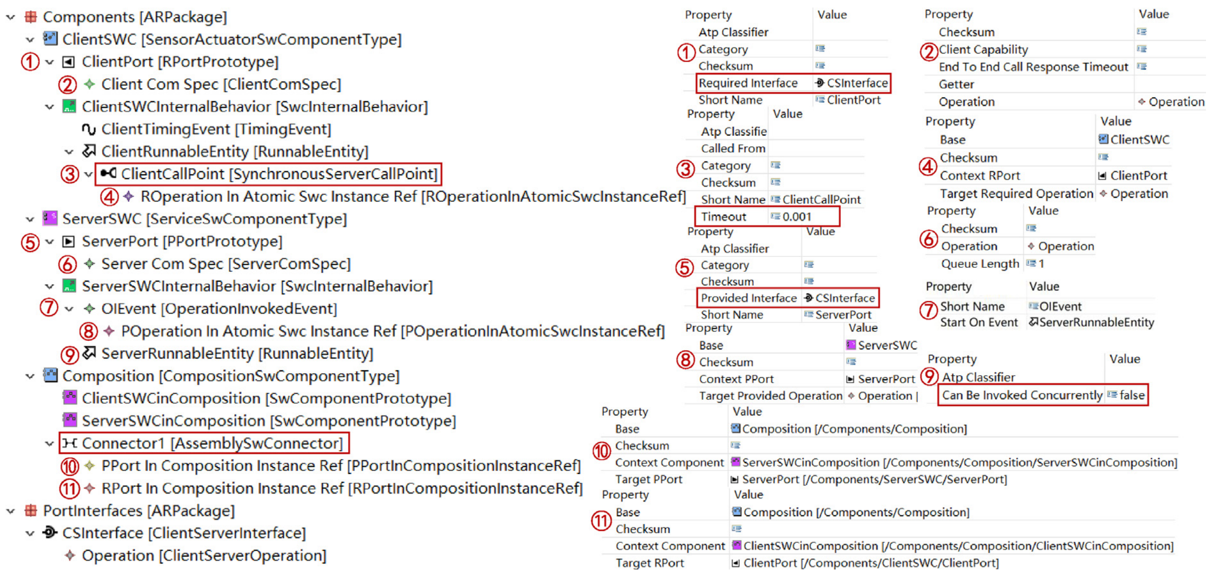
From there, further information extracted from the RTE and Basic Software layers includes the mapping relationship between RunnableEntities and tasks from the *RteEventToTaskMapping* label. In the Basic Software layer, the *OsAlarm* label indicates which alarm activates the task, and the *OsTaskSchedule* label specifies the scheduling scenario, preemptive or non preemptive. Parameters like *OsTaskPriority*, representing the priority of the task, are also acquired.

6. Modelling the AUTOSAR Architecture

In this section, we describe our approach for automatically constructing time-related behaviours in the AUTOSAR architecture as a timed automata network. We begin by explaining the timed automata templates built for each scenario, and then show how timed automata templates are automatically selected and instantiated for each scenario according to the extracted scenarios, parameters, and the specified execution time information.

6.1. Timed automata templates

The AUTOSAR specification clearly defines the main behaviours of the different layers of the AUTOSAR architecture in different scenarios. However, some detailed aspects can be customized according to the various needs of the manufacturers. Based on



```
⌄ Components [ARPackage]
  ⌄ ClientSWC [SensorActuatorSwComponentType]
①   ⌄ ClientPort [RPortPrototype]
②       Client Com Spec [ClientComSpec]
    ⌄ ClientSWCInternalBehavior [SwcInternalBehavior]
        ClientTimingEvent [TimingEvent]
      ⌄ ClientRunnableEntity [RunnableEntity]
③       ⌄ ClientCallPoint [SynchronousServerCallPoint]
④           ROperation In Atomic Swc Instance Ref [ROperationInAtomicSwcInstanceRef]
  ⌄ ServerSWC [ServiceSwComponentType]
⑤   ⌄ ServerPort [PPortPrototype]
⑥       Server Com Spec [ServerComSpec]
    ⌄ ServerSWCInternalBehavior [SwcInternalBehavior]
⑦     ⌄ OIEvent [OperationInvokedEvent]
⑧         POperation In Atomic Swc Instance Ref [POperationInAtomicSwcInstanceRef]
⑨       ServerRunnableEntity [RunnableEntity]
  ⌄ Composition [CompositionSwComponentType]
      ClientSWCinComposition [SwComponentPrototype]
      ServerSWCinComposition [SwComponentPrototype]
    ⌄ Connector1 [AssemblySwConnector]
⑩       PPort In Composition Instance Ref [PPortInCompositionInstanceRef]
⑪       RPort In Composition Instance Ref [RPortInCompositionInstanceRef]
⌄ PortInterfaces [ARPackage]
  ⌄ CSInterface [ClientServerInterface]
      Operation [ClientServerOperation]
```

①

Property	Value
Atp Classifier	
Category	
Checksum	
Required Interface	CSInterface
Short Name	ClientPort

②

Property	Value
Checksum	
Client Capability	
End To End Call Response Timeout	
Getter	
Operation	Operation

③

Property	Value
Atp Classifie	
Called From	
Category	
Checksum	
Short Name	ClientCallPoint
Timeout	0.001

④

Property	Value
Base	ClientSWC
Checksum	
Context RPort	ClientPort
Target Required Operation	Operation

⑤

Property	Value
Atp Classifier	
Category	
Checksum	
Provided Interface	CSInterface
Short Name	ServerPort

⑥

Property	Value
Checksum	
Operation	Operation
Queue Length	1

⑦

Property	Value
Short Name	OIEvent
Start On Event	ServerRunnableEntity

⑧

Property	Value
Base	ServerSWC
Checksum	
Context PPort	ServerPort
Target Provided Operation	Operation |

⑨

Property	Value
Atp Classifier	
Can Be Invoked Concurrently	false

⑩

Property	Value
Base	Composition [/Components/Composition]
Checksum	
Context Component	ServerSWCinComposition [/Components/Composition/ServerSWCinComposition]
Target PPort	ServerPort [/Components/ServerSWC/ServerPort]

⑪

Property	Value
Base	Composition [/Components/Composition]
Checksum	
Context Component	ClientSWCinComposition [/Components/Composition/ClientSWCinComposition]
Target RPort	ClientPort [/Components/ClientSWC/ClientPort]

Fig. 6. The result of the arXML source file (Listing 1) parsed by Artop.

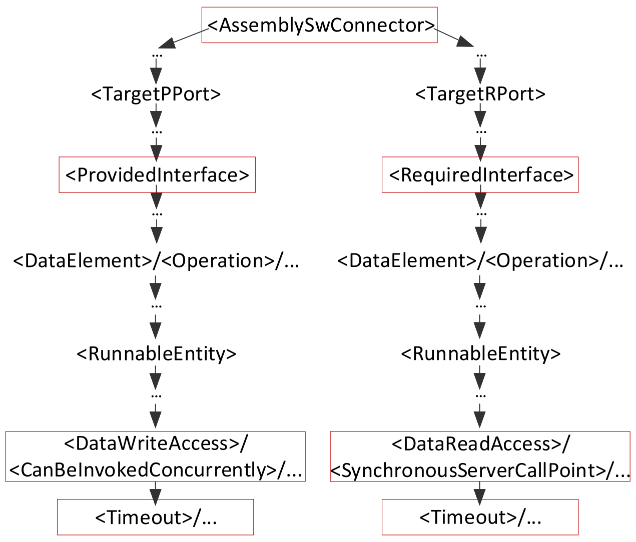


Fig. 7. Flow of scenarios and parameters extraction from a parsed arXML source file.

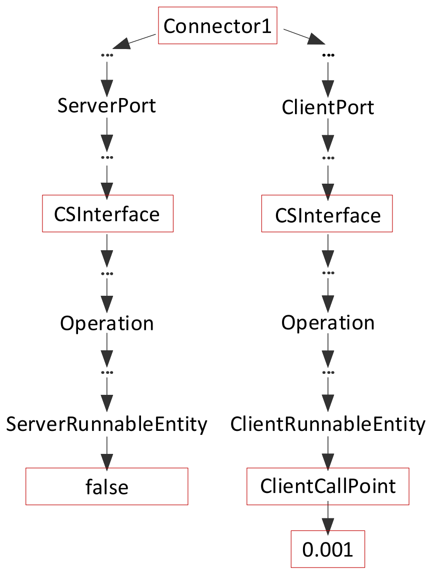


Fig. 8. Flow of scenarios and parameters extraction from the parsed arXML source file (Fig. 6).

the AUTOSAR specification, timed automata templates for the AUTOSAR Software, RTE, and Basic Software layers are defined separately for the various scenarios. As shown in Fig. 4, a group of timed automata templates is constructed for each scenario to characterize the time-related behaviours in the scenario. In addition, in some scenarios, particular behaviours are left undefined in the AUTOSAR specification, so we address them in a manner consistent with industrial practice, as described below. After the timed automata templates are instantiated, they exhibit dynamic interactive behaviours according to the provisions in the AUTOSAR specification. As a conversion, all braces (**{}**) that surround, e.g., instance names, are later on replaced with the concrete instance values. For instance, **{cid}** is subsequently replaced with a number marking the invocation of a client RunnableEntity.

6.1.1. Underspecified behaviours

The AUTOSAR architecture standard is silent on some implementation details, thereby leaving some behaviours *underspecified*. In this study, we supplement the standard with interpretations in three situations:

1. an invocation from a client RunnableEntity when the related RTE queue is full
2. repeated invocations to the server RunnableEntity from the same client RunnableEntity after a timeout occurs in synchronous client–server communication
3. the number of times for accessing a piece of data, invoking a server, or raising a trigger during the execution of a RunnableEntity

How we address each of these is described below, in turn.

In the first situation, an invocation from a client RunnableEntity when the related RTE queue is full, we choose to ignore the invocation outright. The rationale for doing so is that, if the related RTE queue is full, no invocation can be added to the queue, so no further invocations will be executed. This behavioural detail is modelled in timed automaton templates for the RTE in Scenario (16) and for the client RTE in Scenario (17).

In the second situation, when there are repeated invocations to the server RunnableEntity from the same client RunnableEntity after a timeout occurs, we make a decision to either allow or disallow the invocation based on whether the client and server are in the same partition. In the AUTOSAR architecture, partitions are used to decompose an ECU into functional units: there is one RTE per partition controlling the RunnableEntities in the partition. If the client and server are located in the same partition, the RTE shall ensure that the server is not invoked again by the same client until the server has terminated. Otherwise, the RTE shall ensure that the server can be invoked again by the same client after the timeout notification is passed. This behavioural detail is modelled in timed automaton templates, again, for the RTE in Scenario (16) and for the client RTE in Scenario (17).

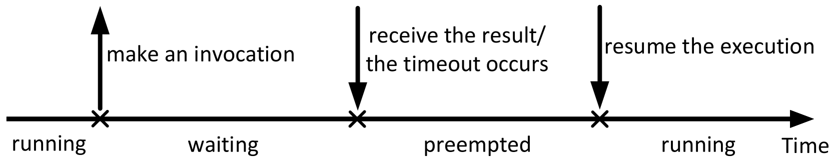
In the third situation, we specify the number of times for accessing a piece of data, invoking a server, or raising a trigger during the execution of a RunnableEntity simply to be one. This particular number appears in the templates of Scenario (1) to Scenario (3), the templates for the client RunnableEntity in Scenario (4) to Scenario (7), and the template for the trigger source RunnableEntity in Scenario (8).

If developers prefer a different interpretation or additional behaviours, of course, they can include them by changing the variables and transitions in the templates as appropriate.

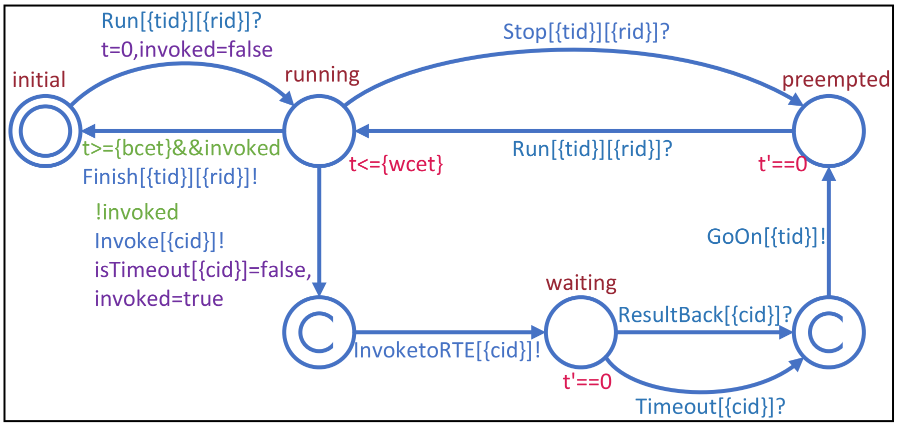
6.1.2. Templates for the AUTOSAR software layer

In the software layer, as shown in Fig. 4, there are eight groups of timed automata templates established to describe the behaviours of the RunnableEntities in eight scenarios covering sender–receiver communication, client–server communication, and trigger event communication. For example, in sender–receiver communication, there are three scenarios to transfer data, namely, "Sender and receiver access data implicitly", "Sender and receiver access data explicitly", and "Receiver is blocked until data is received". Each scenario needs two templates for the description.

For brevity, only the timed automata templates in Scenario (5) of client–server communication, e.g., "Client invokes non-concurrent server synchronously with timeout monitoring" from Fig. 4, are introduced in detail. Although two timed automata templates, client and server, are constructed for this scenario, we focus on the client template here as the other is relatively simple. So first, the behaviours of the client RunnableEntity in this scenario are shown according to the provisions of the



(a)



(b)

Fig. 9. Behaviours of the client RunnableEntity and timed automaton template of the client RunnableEntity in Scenario (5). (a) Behaviours of the client RunnableEntity in Scenario (5). (b) Timed automaton template of the client RunnableEntity in Scenario (5).

AUTOSAR specification. Then the timed automaton template of the client RunnableEntity for this scenario is introduced.

Client–server communication occurs when a client RunnableEntity invokes a server RunnableEntity. Here, considering Scenario (5), the execution method of the server is non-concurrent and the client initiates synchronous invocations with the timeout monitoring mechanism. The behaviours of the client in this scenario are shown in Fig. 9(a) and in accordance with the AUTOSAR specification (Anon, 2021), which states the following:

"...This means that the RunnableEntity is supposed to perform a blocking wait for a response from the server..."

"... The ServerCallPoint allows the specification of a timeout so the client can be notified that the server is not responding and can react accordingly. If the client invokes the server synchronously, the RTE API call to invoke the server reports the timeout..."

"... The task that contains a runnable waiting at a wait point changes from waiting to preempted..."

"...A task that is preempted from executing the ExecutableEntity execution-instance changes state from preempted to running..."

So, in this scenario, the client RunnableEntity first invokes the server RunnableEntity and then waits for a response. The timeout monitoring policy is that if no response is received within the configured timeout, the client RunnableEntity will be awakened by the RTE.

The timed automaton template for the client RunnableEntity in this case is shown in Fig. 9(b). Because there can be multiple client RunnableEntities invoking the same server RunnableEntity, the invocation of each client RunnableEntity is marked by a number **cid**. When the client RunnableEntity makes an invocation, it must send the signal **Invoke[{cid}]** to the scheduler and the signal **InvoketoRTE[{cid}]** to the RTE. It then migrates to the **waiting** location and waits for the result signal **ResultBack[{cid}]**. Here, the behaviour for recording time is modelled in the timed automaton template of the RTE. Therefore, the client RunnableEntity can know the occurrence of the timeout by receiving the signal **Timeout[{cid}]** from the RTE. In addition, a local clock variable **t** is used to control this RunnableEntity and determine whether it executes within the BCET and the WCET.

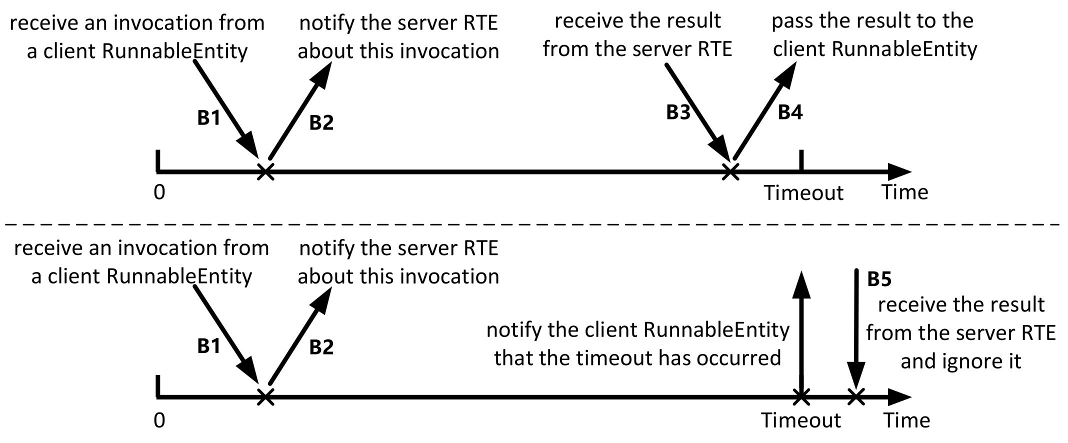
6.1.3. Templates for the RTE layer

From the ten scenarios of the RTE layer, we choose a more complicated control scenario for the detailed description. As shown in number (17) in Fig. 4, the RTE controls client–server communication if a client RunnableEntity and server RunnableEntity are located in different partitions.

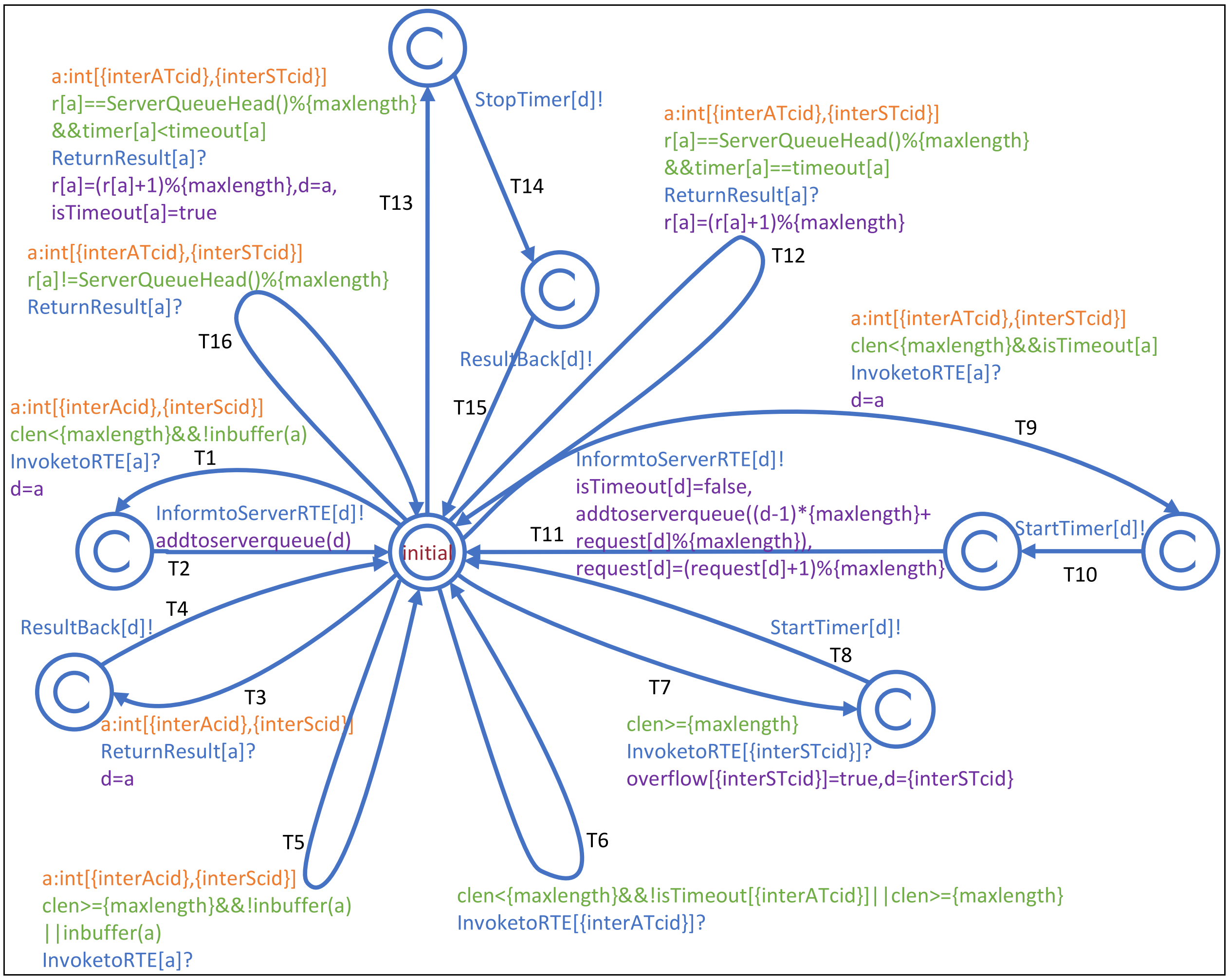
If the client and server are located in different partitions, they are controlled by two RTEs: a client RTE and a server RTE. The critical behaviours of the client RTE are shown in Fig. 10(a). When the client RTE receives an invocation from some client RunnableEntity, it notifies the corresponding server RTE, which activates the server RunnableEntity to execute the invocation and outputs the result to the client RTE. Upon receiving the result, the client RTE passes it to the client RunnableEntity. In case a timeout occurs, that is, the client RTE does not receive the result within a specified deadline, it notifies the client RunnableEntity of the timeout and does not pass the later result to the RunnableEntity, as shown at the bottom of Fig. 10(a). The timed automaton template for the client RTE is shown in Fig. 10(b). As shown in Table 1, the behaviours in Fig. 10(a) are separately realized in Fig. 10(b).

In the function of the template, the client RTE and the server RTE maintain a first-in-first-out queue on the invoked server side to store outstanding invocations for the invoked server RunnableEntity, as defined in the AUTOSAR specification (Anon, 2021):

"... The RTE shall buffer a request on the server side in a first-in-first-out queue..."


receive an invocation from a client RunnableEntity
notify the server RTE about this invocation
receive the result from the server RTE
pass the result to the client RunnableEntity
B1
B2
B3
B4
0
Timeout
Time
receive an invocation from a client RunnableEntity
notify the server RTE about this invocation
B1
B2
notify the client RunnableEntity that the timeout has occurred
B5
receive the result from the server RTE and ignore it
0
Timeout
Time

(a)


a:int[{interATcid},{interSTcid}]
r[a]==ServerQueueHead()%{maxlength}
&&timer[a]<timeout[a]
ReturnResult[a]?
r[a]=(r[a]+1)%{maxlength},d=a,
isTimeout[a]=true
T13
StopTimer[d]!
T14
a:int[{interATcid},{interSTcid}]
r[a]==ServerQueueHead()%{maxlength}
&&timer[a]==timeout[a]
ReturnResult[a]?
r[a]=(r[a]+1)%{maxlength}
T12
a:int[{interATcid},{interSTcid}]
r[a]!=ServerQueueHead()%{maxlength}
ReturnResult[a]?
T16
ResultBack[d]!
T15
a:int[{interATcid},{interSTcid}]
clen<{maxlength}&&isTimeout[a]
InvoketoRTE[a]?
d=a
T9
a:int[{interAcid},{interScid}]
clen<{maxlength}&&!inbuffer(a)
InvoketoRTE[a]?
d=a
T1
InformtoServerRTE[d]!
addtoserverqueue(d)
T2
initial
T11
InformtoServerRTE[d]!
isTimeout[d]=false,
addtoserverqueue((d-1)*{maxlength}+
request[d]%{maxlength}),
request[d]=(request[d]+1)%{maxlength}
StartTimer[d]!
T10
T4
ResultBack[d]!
T3
a:int[{interAcid},{interScid}]
ReturnResult[a]?
d=a
StartTimer[d]!
T8
T7
clen>={maxlength}
InvoketoRTE[{interSTcid}]?
overflow[{interSTcid}]=true,d={interSTcid}
T5
a:int[{interAcid},{interScid}]
clen>={maxlength}&&!inbuffer(a)
||inbuffer(a)
InvoketoRTE[a]?
T6
clen<{maxlength}&&!isTimeout[{interATcid}]||clen>={maxlength}
InvoketoRTE[{interATcid}]?

(b)

Fig. 10. Critical behaviours of the client RTE and timed automaton template of the client RTE in Scenario (17). (a) Critical behaviours of the client RTE in Scenario (17). (b) Timed automaton template of the client RTE in Scenario (17).

Table 1
Mapping from Fig. 10(a) to Fig. 10(b).

Behaviours in Fig. 10(a)	Transitions in Fig. 10(b)
B1	T1, T5, T6, T7, T9
B2	T2, T11
B3	T3, T13
B4	T4, T15
B5	T12, T16

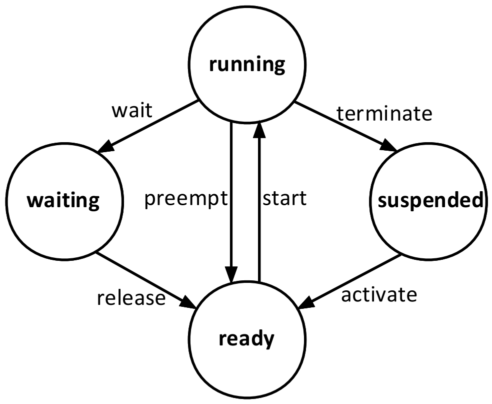


Fig. 11. Task state model in Scenario (19).

The above fragment from the AUTOSAR specification is realized by the transitions labelled with T2 and T11 in Fig. 10(b).

In addition, the template models a timeout monitoring mechanism, conforming to the AUTOSAR specification (Anon, 2021), which states the following:

"...If a timeout was detected in asynchronous inter-ECU or inter-partition client–server communication, the RTE shall ensure that the server can be invoked again by the same client after the timeout notification was passed to the client..."

"...A buffer overflow of the server is not reported to the client. The client will receive a time out..."

"... When the RTE_E_TIMEOUT error occurs, the RTE shall discard any subsequent responses to that request..."

The above three fragments from the AUTOSAR specification are separately realized by the transitions labelled with T9, T7, T12, and T16 in Fig. 10(b).

6.1.4. Templates for the basic software layer

Because the task and scheduler significantly impact time-related behaviours in the AUTOSAR architecture, they are considered here. There are four scenarios in this layer, as shown in Fig. 4, numbered from Scenario (19) to Scenario (22), taking account of the task types, e.g., periodic and sporadic tasks, as well as whether or not the scheduling is preemptive. We introduce Scenario (19) in this part, where the task is periodic, and one timed automaton template is needed to express the entirety of task behaviours.

As shown in Fig. 11, AUTOSAR specifies the states and transitions for the task (Anon, 2005), and all of them are modelled in the proposed timed automaton template, given in Fig. 12. The automaton contains all the states of the task, and the transitions between states are realized through the signals **Activate**, **Start**, **Preempt**, **Wait**, **Release**, and **Terminate**. Here, when the timed automaton template receives the signal **Start** at the location **ready**, or **ready2**, or **ready3**, or **ready4**, it uses the function **HeadTask()** to determine whether the scheduled task is itself. In addition, to realize the time-related behaviours of the periodic task, a local clock **Alarm** is used to activate the task periodically and check whether the execution time of the task meets its deadline **DeadLine[{tid}]**, where the variable **tid** is the id of the task. The timed automaton template of the task can also match the RunnableEntities to itself, for the sake of executing these RunnableEntities in the order of their positions in this task. For this reason, an incremental integer variable **rid** is used to record the positions of these RunnableEntities. When it is the time of the RunnableEntity **rid** to be executed, the task sends the signal **Run[{tid}][rid]** to start this RunnableEntity and the signal **Stop[{tid}][rid]** can be also used to stop the execution of this RunnableEntity if the task **tid** is preempted.

6.2. Mapping and instantiating templates

After scenarios and parameters are extracted from the arXML source file and templates for each scenario built, the required templates are selected according to the extracted scenarios by the A2A tool. A user interface is provided that allows a user to specify the BCET and the WCET for each RunnableEntity. Then, according to the extracted parameters values and execution times, the selected templates are automatically instantiated as described in Section 4, by the A2A tool.

For example, consider generating a timed automaton for the client RunnableEntity in Listing 1. Because the scenario extracted is number (5), the timed automaton template of Fig. 9(b) is selected for instantiation. The information needed to instantiate it comes from the values of the extracted parameters and execution times of the client RunnableEntity. For the RunnableEntity, the parameters include the task containing this particular client RunnableEntity, and the position of the client RunnableEntity in the corresponding task. Then the A2A tool sets the variable **tid** of the corresponding task. Thus, after extraction, **tid** is set to *two*, and its position in the task is *one*. Inputs of the BCET is one second, and the WCET is two seconds. So, by using the A2A tool, the timed automaton template is instantiated, as shown in Fig. 13.

7. Verification of timing constraints

Through the method presented, the AUTOSAR architecture is modelled, automatically, as a timed automata network. Timing constraints on the software must then be verified, as dictated by the AUTOSAR specification. In this study, four types of timing constraints are considered and included.

7.1. Deadlock checking

In UPPAAL, a system is deadlocked if it reaches a state in which there are no outgoing action transitions from the state itself or its delay successors (Behrmann et al., 2004). Whether a deadlock occurs in the model can be directly verified by the TCTL-formula: $A[\,]$ *not deadlock*.

7.2. Latency Timing Constraints (LTCs)

The amount of time that elapses between the occurrence of two events can be constrained (Anon, 2021). Here, the occurrence of the two events has a function-related causal order. Between them, one event is responsible for stimulating and the other is responsible for responding. Therefore, an LTC specifies the minimum and/or maximum duration from the event's occurrence that initiates the stimulus to the occurrence of the response event.

A test automaton template called **Tester** is constructed to verify an LTC, as shown in Fig. 14. To instantiate this timed automaton template, the user needs to input the maximum value of the latency time (MLT) in the user interface. Then the A2A tool

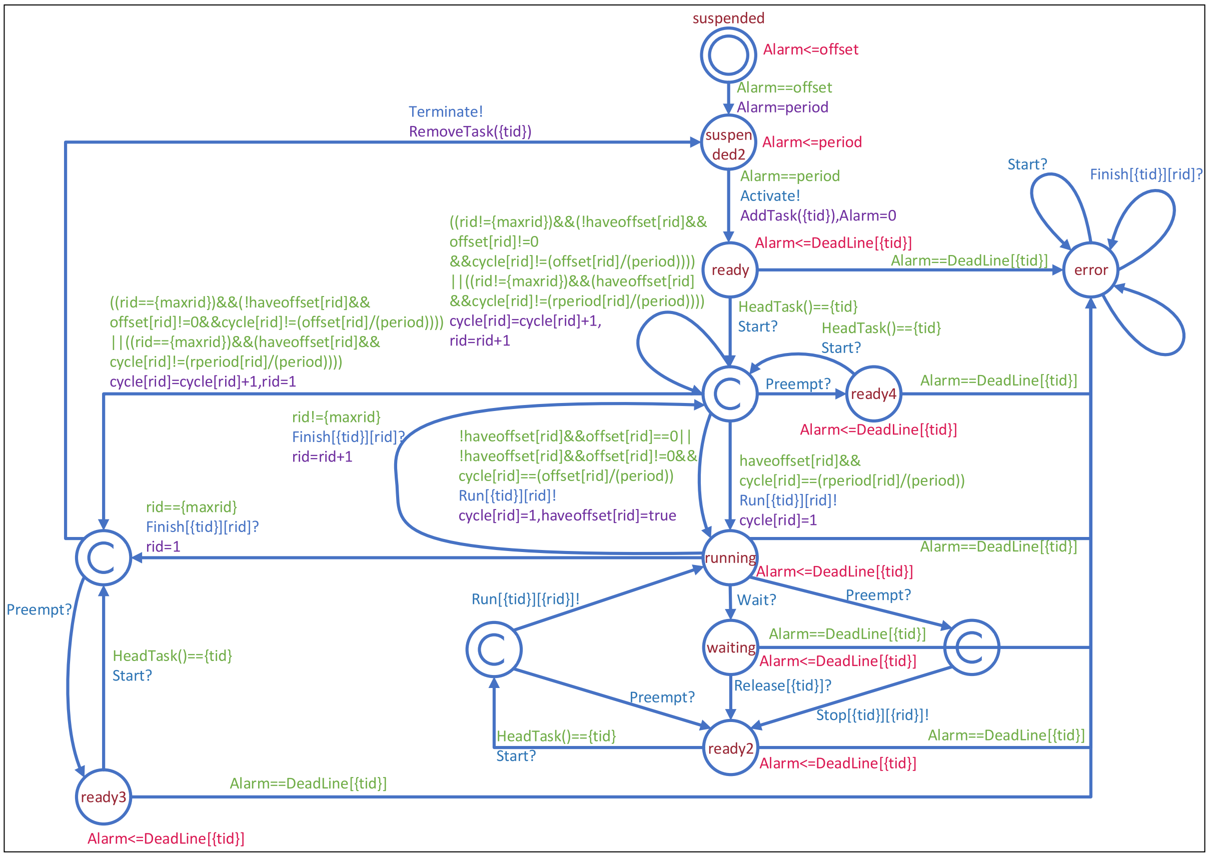


Fig. 12. Timed automaton template of the task in Scenario (19).

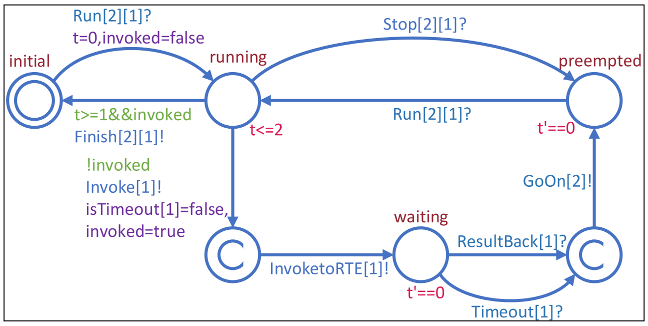


Fig. 13. Timed automaton of the client RunnableEntity in Listing 1.

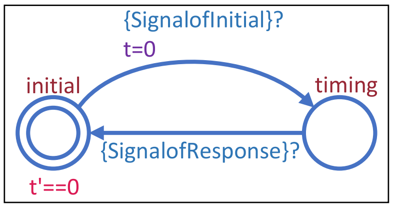


Fig. 14. **Tester** template for verifying an LTC.

instantiates the test automaton template automatically according to the input of the user. When the event initiating the stimulus or the response event occurs, it is sent as a broadcast signal to the test automaton. The test automaton waits for the signal of the event initiating the stimulus at the initial location, and returns to the initial location when receiving the signal of the response event. A local clock **t** is used to record the latency time. Finally, the user can use a TCTL-query to verify an LTC: $A[]\ Tester.t \le MLT$.

7.3. Periodic Event Triggering Constraints (PETCs)

This type of constraint specifies the occurrence of a periodic event (Anon, 2021). Such an event occurs strictly according to a set period, but a jitter value may also be used to relax this strict periodicity requirement. Normally, we hope that a periodic event should occur within an allowable jitter range. Let t_n be the nth occurrence of the event. A periodic event triggering constraint is satisfied if and only if for every occurrence of the event at t_n, the following holds true, where p is the period:

$$(n-1)p \le t_n \le (n-1)p + jitter$$

However, this is not guaranteed because of the multiple tasks and RunnableEntities, as well as different scheduling strategies adopted in the Basic Software layer.

So checking whether the constraint holds is necessary. In the industrial community, the method is one of using testing or simulation with a bounded time interval for observation. Nevertheless, such an approach only considers finite occurrences of the event and cannot cover unbounded intervals. We build an auxiliary automaton template **PTester**, given in Fig. 15, for this purpose. The value of jitter for some event is input by the user interface.

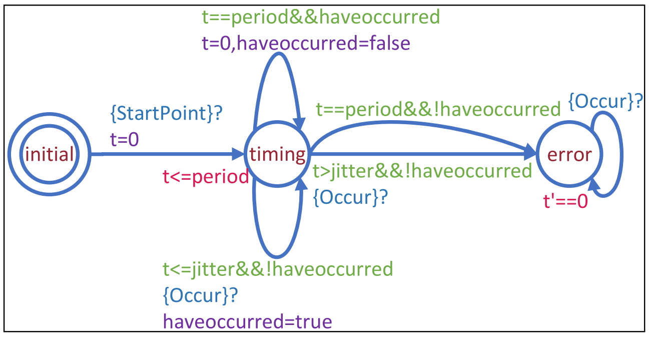


Fig. 15. **PTester** template for verifying a PETC.

In terms of the input, **PTester** is instantiated automatically to determine whether the occurrences of the event meet the periodic event triggering constraint. If 'yes', the 'error' location is not reached. Thus, the property is of the form: *A*[] *not PTester.error*.

7.4. Schedulability

If all tasks always meet their respective deadlines, then schedulability is satisfied. Whether the architecture meets schedulability requirements can be directly verified by using the TCTL-query: *A*[] *not {TaskName}.error* for all tasks, where 'error' is the location denoting the deadline violation of some task.

Note that using stopwatches in Uppaal creates an over-approximation of the state space, meaning that if the above query is satisfiable, it is guaranteed that the architecture is schedulable under all circumstances. However, if the query is not satisfiable, this means that a counterexample has been established in the over-approximation. However, the architecture may still be schedulable since the counterexample may not necessarily correspond to a feasible run of the original architecture (David et al., 2009).

8. Experiment

To evaluate the proposed methodology, we apply it to an actual vehicle interior lighting control system (Anon, 2018). We use the A2A tool to model the time-related behaviours of the system as timed automata, and further verify timing constraints of the model, such as deadlock, LTC, PETC, and schedulability. The Autosar architecture of the vehicle interior lighting control system is first described, and then the verification of the model is presented and explained, showing the effectiveness of the proposed methodology.

8.1. Vehicle interior lighting control system

The vehicle interior lighting control system turns lights inside the car on or off depending on whether the doors are open or closed. The system consists of two door sensors, an actuator, a manager, and a service component; it works as follows:

- The sensors invoke functions in the service component to read the digital signals from the doors and send them to the manager.
- The manager sends the status of the lights to the actuator according to the signals received from the doors.
- The actuator invokes the service component to set the status of the lights, whether to be on or off.

Table 2
The verification results of deadlock.

BCETs (ms)	WCETs (ms)	Deadlock Free
1	2	Yes
2	3	Yes
3	4	Yes
4	5	Yes
5	6	Yes
6	7	Yes
7	8	Yes

Fig. 16 illustrates the Autosar architecture of the lighting control system. Within the Autosar Software layer, in the upper left of the figure, the software component *ioHWAb* provides two services to the sensors through the RunnableEntities *RDigitalServiceReadRight* and *RDigitalServiceReadLeft*, respectively, in the lower left, within the Autosar Software layer. It also includes the RunnableEntity *RDigitalServiceWrite*, which sets the status of the lights.

The software components *LeftDoorSensor* and *RightDoorSensor*, in the centre of the figure, use the RunnableEntities *RLeftDoorSensor* and *RRightDoorSensor* to invoke operations in *ioHWAb* with a period of 100 ms to read the digital signals from the doors. In addition, within the door sensor software components, the RunnableEntities *RGetResultDigitalServiceReadLeft* and *RGetResultDigitalServiceReadRight* are responsible for acquiring digital signals from the corresponding door and sending them to *LightManager*, respectively.

LightManager, in the upper right of the figure, receives digital signals from the doors and sends the status of the lights to *FrontLightActuator* through the RunnableEntities *RLightManagerReceiveLeftDoor* and *RLightManagerReceiveRightDoor*. When the status of the doors is sent to *LightManager*, above centre, the RunnableEntity *RFrontLightActuator* is activated by *LightManager* to read the status of lights, and then it invokes an operation to write the status to the lights.

In the Basic Software layer, at the bottom of the figure, for each RunnableEntity, there is one task to match it. Each task has a defined priority and is preemptive. Some tasks, *TaskLeftDoorSensor* and *TaskRightDoorSensor*, with the alarm indicators, are periodic tasks whose period and deadline are 50 ms.

8.2. Modelling process and verification result

The vehicle interior lighting control system contains eight sporadic tasks, two periodic tasks, and ten RunnableEntities, and includes four sender–receiver communication protocols and three client–server communication protocols. Using the A2A tool, the user first inputs BCETs and WCETs for all RunnableEntities, and the desired timing constraints, and then 34 parallel timed automata are generated from the arXML source file of the system.[2] Four kinds of timing constraints are verified based on the automata:

8.2.1. Deadlock

Using Uppaal to check for deadlock violations, the results in Table 2 show that the model is free of deadlock.

8.2.2. Checking latency timing constraints

Two LTCs are examined:

1. The amount of time from a valid invocation of *RLeftDoorSensor* to the start of an *RGetResultDigitalServiceReadLeft* execution should be less than a specific value of MLT.

[2] https://github.com/Tongji-lab/Autosar/tree/main/AutosarExperiment

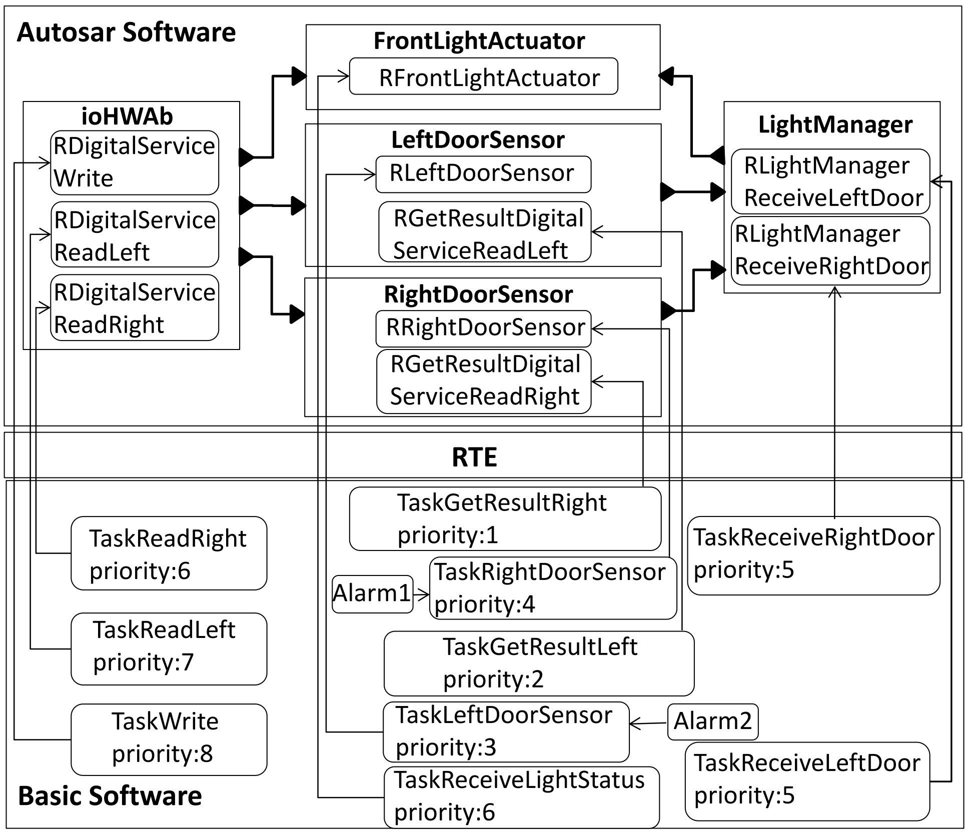

Autosar Software
FrontLightActuator
RFrontLightActuator
ioHWAb
RDigitalService Write
RDigitalService ReadLeft
RDigitalService ReadRight
LeftDoorSensor
RLeftDoorSensor
RGetResultDigital ServiceReadLeft
RightDoorSensor
RRightDoorSensor
RGetResultDigital ServiceReadRight
LightManager
RLightManager ReceiveLeftDoor
RLightManager ReceiveRightDoor
RTE
TaskGetResultRight priority:1
TaskReadRight priority:6
Alarm1
TaskRightDoorSensor priority:4
TaskReceiveRightDoor priority:5
TaskReadLeft priority:7
TaskGetResultLeft priority:2
TaskWrite priority:8
TaskLeftDoorSensor priority:3
Alarm2
TaskReceiveLeftDoor priority:5
TaskReceiveLightStatus priority:6
Basic Software

Fig. 16. AUTOSAR architecture of the vehicle interior lighting control system.

2. The amount of time from a valid invocation of *RRightDoorSensor* to the start of an *RGetResultDigitalServiceReadRight* execution should be less than a specific value of MLT.

There are four RunnableEntities considered in the two properties, namely, *RLeftDoorSensor*, *RGetResultDigitalServiceReadLeft*, *RRightDoorSensor*, and *RGetResultDigitalServiceReadRight*. The user adds as input MLT to verify the LTC, and the results are shown in Table 3. For each pair of BCETs and WCETs values, we see that, compared with LTC (a), LTC (b) requires a larger value of MLT to satisfy the constraint. This property is due to different priorities of the tasks that correspond to the four RunnableEntities, resulting in the executions of more additional RunnableEntities between the two Entities of LTC (b). In addition, for each LTC, following the increase of the BCETs and WCETs, the value of MLT satisfying the LTC increases accordingly because of longer executions of the related RunnableEntities.

8.2.3. *Checking periodic event triggering constraints*

Two PETCs are considered:

1. The occurrence of *RLeftDoorSensor* triggered by a periodic event should occur within an allowable jitter range.
2. The occurrence of *RRightDoorSensor* triggered by a periodic event should occur within an allowable jitter range.

In Table 4, the user adds as input the expected jitter to verify the PETCs. To satisfy the constraints, a larger jitter value is needed for PETC (a) than PETC (b). This is because the task *TaskLeftDoorSensor* containing *RLeftDoorSensor* has a lower priority than the task *TaskRightDoorSensor* containing *RRightDoorSensor*, causing *RLeftDoorSensor* to wait for more RunnableEntities to complete before its execution. In addition, we see that the value of jitter satisfying each PETC grows with the increase of BCETs and WCETs. This is also due to the fact that *RLeftDoorSensor* and *RRightDoorSensor* should spend more time waiting for other RunnableEntities to complete.

Table 3
The verification results of LTCs.

BCETs (ms)	WCETs (ms)	MLT (ms)	LTC (a)	LTC (b)
1	2	8	No	No
		11	Yes	No
		18	Yes	No
		20	Yes	Yes
1	3	14	No	No
		17	Yes	No
		29	Yes	No
		31	Yes	Yes
2	4	19	No	No
		20	Yes	No
		37	Yes	No
		42	Yes	Yes

8.2.4. *Schedulability*

The results of the schedulability verification are shown in Table 5. We find that as BCETs and WCETs increase, the schedulability cannot be satisfied. This is due to the fact that the RunnableEntities in each task will take longer to execute, causing the task to fail to meet its deadline.

Through these experiments modelling and working with the AUTOSAR architecture, we find the A2A tool to be robust in its ability to extract configuration information from arXML source

Table 4
The verification results of PETCs.

BCETs (ms)	WCETs (ms)	Jitter (ms)	PETC (a)	PETC (b)
1	2	0	No	No
		2	No	Yes
		4	No	Yes
		6	Yes	Yes
1	3	1	No	No
		4	No	Yes
		8	No	Yes
		10	Yes	Yes
2	4	3	No	No
		6	No	Yes
		11	No	Yes
		12	Yes	Yes

Table 5
The verification results of the schedulability.

BCETs (ms)	WCETs (ms)	Schedulability
1	2	Yes
2	3	Yes
3	4	Yes
4	5	Yes
5	6	Yes
6	7	No
7	8	No

files and to construct corresponding timed automata for both the architecture and the auxiliary tests. For testing larger systems, as the number of clocks in the AUTOSAR architecture grows, we are aware of the possibility that state explosion could lead to longer verification times. In addition, the timing performance of systems is sensitive to modest changes in parameter values, so exploring the methodology in configuring and optimizing those parameters could be a fruitful direction for further study.

9. Conclusion

In this study, we have demonstrated the viability of automatically extracting time-related behaviours from AUTOSAR architecture description files, and turning them into a network of timed automata in UPPAAL that can be checked against timing properties of interest, including deadlocks, latencies, periodic event triggering, and schedulability. Additional insight might be gained by formulating and checking timing properties beyond those we have considered. In addition, future advancements addressing the state space explosion problem may enable the verification of even larger and more complex AUTOSAR architectures.

In terms of scope, our work is still preliminary—a proof of concept. The AUTOSAR specification defines an immense number of scenarios, rules, and behaviours, and we have modelled just a subset of those that are time-related. These time-related scenarios directly determine the verification result, and other aspects are simplified in order to highlight the effects of sequence and timing constraints. More specifically, it is time-related scenarios in the AUTOSAR Software layer and the RTE layer that are considered. For the Basic Software layer, scenarios with interrupting processing and mixed preemptive scheduling, which have preemptable and non-preemptable tasks mixed on the same architecture, are unaddressed, though these will be explored in future work. In terms of timing constraints, we have considered four broad categories of them, but the work could be extended to address additional ones that appear in the AUTOSAR specification, namely, constraints on execution time, execution order, synchronization timing, offset timing, and age.

For future work, we plan to augment our modelling and verification of AUTOSAR architectures by extending the capabilities of the A2A tool. In particular, new timed automata templates will be constructed to characterize behaviours in scenarios that include interrupting processing and a mixed preemptive scheduling, and these will be added to the A2A library. In addition, we plan to support new categories of timing constraints by building the necessary timed automata test templates. To address the state space explosion problem, we hope to refine our models by reducing the number of states, variables, and clocks needed, and by making use of assume guarantee reasoning (Clarke et al., 1989; Grumberg and Long, 1994).

More far-reaching goals include the use of model learning algorithms to further improve and streamline the process. In the present work, timed automata templates must be constructed, and scenarios and parameters extracted, in advance, from a given arXML file. In an effort to reduce the tedium of this step, we hope to develop algorithms that observe the external behaviours of a given AUTOSAR architecture, and then learn corresponding timed automata that are able to express them.

CRediT authorship contribution statement

Miaomiao Zhang: Conceptualization, Methodology, Formal analysis, Writing – review & editing. **Yu Teng:** Conceptualization, Methodology, Validation, Formal analysis, Writing – original draft, Writing – review & editing. **Hui Kong:** Conceptualization, Methodology, Formal analysis. **John Baugh:** Writing – review & editing. **Yu Su:** Software. **Junri Mi:** Methodology, Software. **Bowen Du:** Conceptualization, Methodology, Formal analysis, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work is supported in part by the project of Huawei Corporation, the National Natural Science Foundation of China (61972284, 62032019).

References

Alam, M.S.U., Iqbal, S., Zulkernine, M., Liem, C., 2019. Securing vehicle ECU communications and stored data. In: 2019 IEEE International Conference on Communications. ICC, IEEE, pp. 1–6.
Alladi, T., Chakravarty, S., Chamola, V., Guizani, M., 2020. A lightweight authentication and attestation scheme for in-transit vehicles in IOV scenario. IEEE Trans. Veh. Technol. 69 (12), 14188–14197.
Alur, R., Dill, D.L., 1994. A theory of timed automata. Theoret. Comput. Sci. 126 (2), 183–235.
Anon, 2005. OSEK/VDK operating system. [Online]. Available: http://www.openosek.org/tikiwiki/tiki-index.php.
Anon, 2018. AUTOSAR application design. [Online]. Available: https://www.cnblogs.com/snddman/p/10138552.html.
Anon, 2021. AUTOSAR: Document search. [Online]. Available: https://www.autosar.org/nc/document-search.
Anssi, S., Tucci-Piergiovanni, S., Kuntz, S., Gérard, S., Terrier, F., 2011. Enabling scheduling analysis for AUTOSAR systems. In: 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. IEEE, pp. 152–159.
Behrmann, G., David, A., Larsen, K.G., 2004. A tutorial on UPPAAL. In: Formal Methods for the Design of Real-Time Systems. Springer, pp. 200–236.
Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M., 2006. UPPAAL 4.0. In: Quantitative Evaluation of Systems. IEEE Computer Society, Los Alamitos, CA, pp. 125–126.
Bengtsson, J., Yi, W., 2003. Timed automata: Semantics, algorithms and tools. In: Advanced Course on Petri Nets. Springer, pp. 87–124.

Beringer, S., Wehrheim, H., 2016. Verification of Autosar software architectures with timed automata. In: Critical Systems: Formal Methods and Automated Verification. Springer, pp. 189–204.

Cassez, F., Larsen, K., 2000. The impressive power of stopwatches. In: International Conference on Concurrency Theory. Springer, pp. 138–152.

Charette, R.N., 2021. How software is eating the car. [Online]. Available: https://spectrum.ieee.org/software-eating-car.

Chen, J., Alalfi, M.H., Dean, T.R., Ramesh, S., 2018. Modeling Autosar implementations in simulink. In: European Conference on Modelling Foundations and Applications. Springer, pp. 279–292.

Choi, Y., 2018. A configurable V&V framework using formal behavioral patterns for OSEK/VDX operating systems. J. Syst. Softw. 137, 563–579.

Clarke, E.M., Long, D.E., McMillan, K.L., 1989. Compositional model checking.

Copic, M., Leupers, R., Ascheid, G., 2020. Modelling machine learning components for mapping and scheduling of Autosar runnables. In: 2020 IEEE 31st International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 127–137.

David, A., Illum, J., Larsen, K.G., Skou, A., 2009. Model-based framework for schedulability analysis using Uppaal 4.1. In: Model-Based Design for Embedded Systems. Citeseer, pp. 121–143.

Denil, J., De Meulenaere, P., Demeyer, S., Vangheluwe, H., 2017. DEVS for Autosar-based system deployment modeling and simulation. Simulation 93 (6), 489–513.

Fennel, H., Bunzel, S., Heinecke, H., Bielefeld, J., Fürst, S., Schnelle, K.-P., Grote, W., Maldener, N., Weber, T., Wohlgemuth, F., et al., 2006. Achievements and exploitation of the Autosar development partnership. Convergence 2006, 10.

Fürst, S., Bechter, M., 2016. Autosar for connected and autonomous vehicles: The Autosar adaptive platform. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W). IEEE, pp. 215–217.

Fürst, S., Mössinger, J., Bunzel, S., Weber, T., Kirschke-Biller, F., Heitkämper, P., Kinkelin, G., Nishikawa, K., Lange, K., 2009. Autosar–a worldwide standard is on the road. In: 14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden. Vol. 62, p. 5.

Garcia, J.L.M., Olmedo, I.S., 2020. Introducing a deferrable server into Autosar. In: 2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications. RTCSA, IEEE, pp. 1–6.

Grumberg, O., Long, D.E., 1994. Model checking and modular verification. ACM Trans. Progr. Lang. Syst. (TOPLAS) 16 (3), 843–871.

Gu, Z., Han, G., Zeng, H., Zhao, Q., 2016. Security-aware mapping and scheduling with hardware co-processors for flexray-based distributed embedded systems. IEEE Trans. Parallel Distrib. Syst. 27 (10), 3044–3057.

Jelecevic, E., Minh, T.N., 2019. Visualize real-time data using Autosar.

Klobedanz, K., Kuznik, C., Thuy, A., Mueller, W., 2010. Timing modeling and analysis for Autosar-based software development-a case study. In: 2010 Design, Automation & Test in Europe Conference & Exhibition. DATE, IEEE, pp. 642–645.

Kotur, M., Lukić, N., Krunić, M., Velikić, G., 2020. Utilization of design patterns in Autosar adaptive standard. In: 2020 IEEE 10th International Conference on Consumer Electronics (ICCE-Berlin). IEEE, pp. 1–6.

Larsen, K.G., Larsson, F., Pettersson, P., Yi, W., 1997. Efficient verification of real-time systems: Compact data structure and state-space reduction. In: Proceedings Real-Time Systems Symposium. IEEE, pp. 14–24.

Luong, H.P., Panda, M., Vu, H.L., Vo, B.Q., 2017. Beacon rate optimization for vehicular safety applications in highway scenarios. IEEE Trans. Veh. Technol. 67 (1), 524–536.

Menard, C., Goens, A., Lohstroh, M., Castrillon, J., 2020. Achieving determinism in adaptive Autosar. In: 2020 Design, Automation & Test in Europe Conference & Exhibition. DATE, IEEE, pp. 822–827.

Nasser, A., Ma, D., 2019. Defending Autosar safety critical systems against code reuse attacks. In: Proceedings of the ACM Workshop on Automotive Cybersecurity. pp. 15–18.

Neumann, S., Kluge, N., Wätzoldt, S., 2012. 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. pp. 55–60.

Piper, T., Winter, S., Suri, N., Fuhrman, T.E., 2015. On the effective use of fault injection for the assessment of Autosar safety mechanisms. In: 2015 11th European Dependable Computing Conference. EDCC, IEEE, pp. 85–96.

Sarikhada, R.M., Shah, P.K., 2020. Speed up the validation process by formal verification method. In: 2020 IEEE International Conference for Innovation in Technology. INOCON, IEEE, pp. 1–4.

Sheng, Z., Kenarsari-Anhari, A., Taherinejad, N., Leung, V.C., 2015. A multichannel medium access control protocol for vehicular power line communication systems. IEEE Trans. Veh. Technol. 65 (2), 542–554.

Stegmeier, A., Kehr, S., George, D., Bradatsch, C., Panic, M., Bödekker, B., Ungerer, T., 2017. Evaluation of fine-grained parallelism in Autosar applications. In: 2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation. SAMOS, IEEE, pp. 121–128.

Tucci-Piergiovanni, S., Mraidha, C., Wozniak, E., Lanusse, A., Gerard, S., 2011. A UML model-based approach for replication assessment of Autosar safety-critical applications. In: 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications. IEEE, pp. 1176–1187.

Voget, S., 2010. Autosar and the automotive tool chain. In: 2010 Design, Automation & Test in Europe Conference & Exhibition. DATE, IEEE Computer Society, pp. 259–262.

Yamili, Y.C., Kathiresh, M., 2021. Autosar and MISRA coding standards. In: Automotive Embedded Systems. Springer, pp. 37–70.

Yan, R., Guo, J., 2019. Timing modeling and analysis for Autosar schedule tables. In: 2019 IEEE 19th International Symposium on High Assurance Systems Engineering. HASE, IEEE, pp. 123–130.

Yan, R., Yang, J., Zhu, D., Huang, K., 2018. Design verification and validation for reliable safety-critical autonomous control systems. In: 2018 23rd International Conference on Engineering of Complex Computer Systems. ICECCS, IEEE, pp. 170–179.

Zhang, M., Chong, P.H.J., Seet, B.-C., 2019. Performance analysis and boost for a MAC protocol in vehicular networks. IEEE Trans. Veh. Technol. 68 (9), 8721–8728.

Zhao, Q., Gu, Z., Zeng, H., 2017. Design optimization for Autosar models with preemption thresholds and mixed-criticality scheduling. J. Syst. Archit. 72, 61–68.

Zhu, L., Liu, P., Shi, J., Wang, Z., Zhu, H., 2013. A timing verification framework for Autosar OS component development based on real-time maude. In: 2013 International Symposium on Theoretical Aspects of Software Engineering. IEEE, pp. 29–36.

Miaomiao Zhang is a full professor at School of Software Engineering, Tongji University, Shanghai, China. Her research interests include formal methods and software engineering. Contact her at miaomiao@tongji.edu.cn.

Yu Teng is currently a Ph.D. student at School of Software Engineering, Tongji University, Shanghai, China. Her research interests include formal methods and software engineering. Contact her at tengyu@tongji.edu.cn.

Hui Kong is a technical expert in formal verification at Huawei Technologies Co., Ltd in Shanghai, China. His research is mainly focused on software and hybrid system model checking. Contact his at kongh08@gmail.com.

John Baugh is a professor in the Department of Civil, Construction, and Environmental Engineering at North Carolina State University in Raleigh, North Carolina. His research interests include cyber-physical systems, scientific computing, and formal methods. Contact him at jwb@ncsu.edu.

Yu Su is currently a master student at School of Software Engineering, Tongji University, Shanghai, China. Contact him at lemon@tongji.edu.cn.

Junri Mi is currently an engineer at Alibaba Corporation, Hangzhou, China. Contact him at 18mjr@tongji.edu.cn.

Bowen Du is an assistant professor at School of Software Engineering, Tongji University, Shanghai, China. His research interests include cyber-physical system, artificial intelligence and software engineering. Contact him at bowendu@tongji.edu.cn.