Multi-aspect Virtual Integration approach for Real-Time and Safety Properties

Tayfun Gezgin, Raphael Weber, Markus Oertel
OFFIS Institute for Information Technology
Escherweg 2, 26121 Oldenburg, Germany
{tayfun.gezgin, raphael.weber, markus.oertel}@offis.de

***Abstract*—Verification techniques for analyzing the design or requirements at early development stages are used since the beginning of the model-based design paradigm. Most of these analyses are focused on a single purpose, like safety, real-time, or geometry. This separation of concerns leads to the introduction of so called aspects that describe these properties of a system. Nevertheless, these aspects are not necessarily independent. In this paper we use the fault tolerance time interval, the maximum time to recover from faults, as an example to state the need for a multi-aspect analysis. We present how a virtual integration test can be performed covering safety and real-time properties to prove the correct refinement of requirements. Our requirements formalization approach using contracts, a pattern language and the internal representation as timed automata are described. The presented technique is applied to an automotive lane-keeping-support system.**

I. Introduction & Related Work

For safety-critical systems a safety case is required that proves the absence of unreasonable risks. Therefore, depending on the applicable safety standards, a rather huge set of analysis activities needs to be performed. Typically, this includes safety and timing analyses. These disciplines have been separated for a long time, finally leading to the establishment of dedicated design aspects (views on the system) [1] which are purely concerned with *either* safety *or* timing.

In practice, both aspects safety and real-time typically cannot be regarded in isolation. For example, faults in an automatic light controller of a vehicle need to be detected and mitigated in about 200ms. Therefore, changes in the safety concept may directly influence the timing requirements of the related components. Today's safety standards like the ISO 26262 [2] put an early emphasis on both aspects, leading to a need for combined analyses. As an example for the need of a combined analysis at an early stage of the development process is the concept of fault tolerance time intervals (FTTI) which define the maximum time to mitigate a fault in the system, before it leads to a hazard. These requirements address both aspects at the same time and are relevant from the initial hazard analysis until the final development of software and hardware.

During the design process of a system the overall architecture is typical refined in more fine-grained component structures. To avoid time intensive manual reviews, which have to be performed over and over again in case of changes in the design, automated techniques have been proposed to analyze the correct split-up of requirements. A popular technique is called the virtual integration test (VIT, see Section IV) [3] which is able to prove whether a set of sub-requirements fulfills a top-level requirement. The VIT uses contracts [4], [5] as a paradigm to enable component based black-box integration tests. The contract assertions are stated in a formal language. Simple to use languages have been proposed for timing requirements [6] as well as for safety requirements [7].

Note that the aim of this paper is not to demonstrate a high performance multi-aspect analysis tool but to present a new method to combine both aspects, safety and real-time. Also note that we will not present a satisfaction analysis but a method to check the validity of a virtual integration concerning the specifications of components. We illustrate the usability and feasibility by applying our timed automaton based VIT prototype implementation to an industrial case study with the restriction to permanent fault occurrences.

A. Related Work

Timing analyses were performed late in the development process, after the tasks and software architectures had been defined, to give confidence in the chosen scheduling algorithm and the expected worst case execution times (WCETs). In literature there are many techniques to verify these properties. The classical approach for scheduling analysis is a holistic one, as it was worked out by e. g. Tindell and Clark [8] computing fixed-point equations of the whole architecture to determine the response times of all allocated tasks. Mubeen et al. presented a tighter version of response time analysis with offsets in [9] which is also implemented in the Rubus-ICE [10]. In [11] activation pattern for tasks are described by upper and lower arrival curves realizing a more *compositional* analysis method. Based on this work a compositional scheduling analysis tool, called SymTA/S, was created by SymtaVision [12]. These classical approaches typically yield pessimistic results for systems containing distributed resources. State based techniques as presented in [13], [14] overcome these deficiencies coming along with a worse scalability. Another work using a UPPAAL timed automata representation (like our approach) derived from UML models is presented in [15]. However, none of these approaches addresses safety aspects.

Safety analyses were mainly performed on higher abstraction levels to prove the correctness of the safety concept. A classical approach concentrating on fault propagation are *HipHops* [16] to generate fault trees based on local propa-

gation specifications. In contrast to this approach, which can be used together with development models like *Simulink* [17] also a couple of more theoretical safety analysis techniques exist like "function-structure models" [18] or the commonly used formalism of processes and channels [19]. However, none of the existing approaches is able to deal with real-time requirements in a proper way.

For virtual integration testing also some tooling support exists: A tool for checking the refinement between contracts called *OCRA* was presented in [20]. There, contracts are specified in a pattern based language called *Othello* which are later translated to a temporal logic. In contrast to our work, they use an SMT solver to check such relations. We use a timed automaton approach as we also want to extend our approach by a schedulability check in future work.

B. Outline

In this paper we present an approach to formalize and analyze (in a virtual integration setting) requirements addressing both, safety and timing. In Section III we explain the necessity to combine safety and timing analyses on the example of the fault tolerance time interval. We present an extension of the pattern-based requirements specification language (RSL) to cover sporadic activations (representing sporadically occurring faults) and present a translation to timed automata in Section IV. We use a case-study of an lane-keeping-support system to apply our technique in Section V, and finally give a conclusion and outlook for future work.

II. Fundamentals

This section provides an overview of basic notions and formalisms necessary to understand how we realized the virtual integration test. In our previous work [21] on a new common systems meta-model (CSM), Heterogeneous Rich Components (HRCs), which originate from the European SPEEDS project [22], represent the major modelling artifact. These HRCs will be explained in Subsection II-A. In addition to HRCs a methodology to traverse along the design space was also proposed in [21], a short outline of which is provided in Subsection II-B. HRC requirements are specified in a formalized way in the pattern based requirements specification language (RSL). An excerpt of the patterns relevant for this work is given in Subsection II-C. Finally, Subsection II-D contains a formal description of contracts to which we will refer in our VIT approach (Section IV).

A. Heterogeneous Rich Components

The CSM provides basic building blocks needed to model systems as components with ports and connections (bindings) between them. We refer to these components as Heterogeneous Rich Components (HRCs). HRCs rely on the basic concepts of SysML blocks [23]. The dynamics of an HRC can be specified by behavior, e. g. an external behavior model, or even source code. Furthermore, requirements (or contracts) refer to a *required* behavior whereas the *actual* behavior is specified as stated above. The idea of contracts is inspired by Bertrand Meyer's programming language Eiffel and its *design by contract* paradigm [24].

In HRC, contracts are a pair consisting of an assumption and a guarantee, both of which are specified by some text. Here we assume that assumptions and guarantees are specified in RSL, see Subsection II-C. An assumption specifies how the context of the component, i. e. the environment from the point of view of the component, should behave. Only if the assumption is fulfilled, the component will behave as guaranteed. This enables the verification of virtual system-integration (integrate a more detailed component or a sub-component in a more abstract environment) at an early stage in the design flow, even when there is no implementation yet. Thus, the system decomposition can be verified with respect to contracts. Contracts will be explained in more detail in Subsection II-D. Note that in this work we consider only the HRC semantics where a connection between two ports describes their equality. A deeper insight into HRCs can also be found in [25], [26].

B. Structuring the development process

When developing an embedded system, an architecture is regarded in different *Perspectives* at several *Abstraction Levels* during the design process as mentioned in Section I. On each abstraction level the product architecture is regarded in different perspectives. As an additional concept for the separation of concerns, models in each perspective reflect different aspects. For example, an aspect "Safety" might be regarded in every perspective but an aspect "Real-Time" is not regarded in a geometric perspective and the aspect "Cost" is not regarded when considering operational use cases.

To keep the models in different perspectives and abstraction levels consistent (keep traceability between development steps) we defined a so called *Realization-* and *Allocation-Link*. The basic idea behind both concepts is to relate the observable behavior of components exposed at its ports.

Realizations are relationships between ports of components on different abstraction levels. Intuitively a realization-link states, that a component *c1* has somehow been refined to component *c1'* and is now more concrete in terms of its interface and/or behavior. The refinement cannot always be captured by a pure decomposition approach. Thus, we define the realization of a component by introducing a state-machine that translates the behavior of the refined component *c1'* into according events observable at a port of component *c1*.

Allocations are relationships between ports of components in different perspectives. Intuitively an allocation-link states that the logical behavior of a component *c4* is part of the behavior of a resource *r2*, to which it has been allocated. Here, we consider the same link-semantics as for the realization. For more details, refer to [21]. In Section IV we will introduce a technique to automatically verify the allocation and realize relations of components.

As mentioned above, one basic design step is (de-) composition. With contracts annotated to both the component and its sub-components, we want to check whether the composition

of the sub-components behavioral specifications is a valid refinement of the surrounding component's behavior. The following section will describe a basic set of language patterns that may be used to specify contracts.

C. Requirement Specification Language

Natural language requirements are often accompanied by ambiguity, incompleteness or inconsistency; the reason for this resides in the very nature of a spoken language. These unintended byproducts may not be detected until late phases of the development process and therefore cause the realization to not fulfill the intended goals of the specification or cause incompatibilities to other system parts. A solution to this problem would be the use of formal languages to specify the requirements. But these are often hard to understand and therefore difficult to use. An alternative that bridges this gap is the pattern-based Requirements Specification Language (RSL) [27], [26]. This formal language provides a fixed semantic that enables an automated validation or verification but is still well readable compared to natural language.

Consisting of static text elements and attributes which are filled in by a requirements engineer, patterns have a well-defined semantic so that a consistent interpretation of the system specification between all stakeholders can be ensured. To cope with the needs of the different aspects of a design, various sets of patterns have been defined which are partly described in the following. We thereby focus on the portions relevant for this work.

1) Basic Elements: We will shortly describe the basic elements (like events or time intervals) used in most patterns.

a) Events and Sets of Events: Events are signals that occur at a defined point in time. Examples are messages sent over a bus system, sporadic signals from sensors or even user interaction like pressing a button. While specifying events the names can be chosen by the requirements engineer, but have to be used consistently, i. e. same events have to have the same identifiers. If there is an underlying architecture these names have to be mapped to the corresponding parts in the design model in later design stages.

b) Intervals: Intervals describe the amount of time between two points in time. These points in time can either be an event or a timed value. In case of events the first occurrence of the *startevent* opens the interval and the first occurrence of the *endevent* after *startevent* closes the interval: "`[startevent, endevent]`". In case that timed values are the boundaries for an interval there must be reference point for the timers. The times in the action part refer to the activation point of the pattern through the trigger. In that sense it is not allowed to use *unbounded* times values in the trigger.

c) Open/Closed Intervals: The braces for writing the intervals can be used to express open and closed intervals:

`[a,b]` → closed, closed interval

`]a,b[` → opened, opened interval

But also other combinations are possible:

`[a,b[` → closed, opened interval

`]a,b]` → opened, closed interval

Closed intervals ends indicate that the point in time where the event occurs still belongs to the interval, open interval ends indicate that the point in time at the events representing the border does not belong to the interval.

2) R1 Pattern - Periodic Activation: `event` **`occurs each`** `period [`**`with jitter`** `jitter]`

This pattern describes the periodic occurrence of an event such as the activation of a task. The event can be delayed by an additional jitter. This is a derived pattern from R2 such that *minperiod* equals *maxperiod*. We restrict the jitter to be less or equal than the period.

Example: Natural language requirement: The diagnosis task shall be executed every 10ms and may jitter 1ms. RSL: `diagnosisTaskActivate` **`occurs each`** `10ms` **`with jitter`** `1ms`.

3) R2 Pattern - Sporadic Activation: `event` **`occurs sporadic with minperiod`** `period [`**`and maxperiod`** `period] [`**`and jitter`** `jitter]`.

This pattern describes the sporadic occurrence of an event. The minimal distance of the ideal occurrences is defined by *minperiod*, which can be delayed by an additional jitter. If *maxperiod* is given this specifies the maximal distance between the ideal occurrences. The minimal distance between the real occurrences can be constrained by Pattern R3. As for the $R1$ pattern, we restrict the jitter to be less or equal than the maxperiod.

Example: Natural language requirement: Processing of received data is necessary at most each 10ms and may jitter 1ms. RSL: `dataReceived` **`occurs sporadic with minperiod`** `10ms` **`and jitter`** `1ms`.

4) R3 Pattern - Delay: **`Whenever`** `event` **`occurs,`** `event` **`[does not] occur[s] [during`** `interval`**`]`**.

This pattern describes the dependency between two events that may occur on different subjects. When using the optional interval attribute the pattern claims that the second event occurs in the specified time bounds. If no interval is specified, there is no constraint on when the event has to occur. In such a case the requirement has to be refined later.

Example: Natural language requirement: The response time of the diagnosis task shall be 10ms +/- 1ms. RSL: **`whenever`** `diagTaskStart` **`occurs,`** `diagTaskFinished` **`occurs during`** `[9ms, 11ms]`.

D. Contracts

We use the pattern-based requirements specification language as described in Subsection II-C to specify contracts. For the analysis, the text patterns are transformed to timed automata, as will be discussed in Section IV.

Formally, the semantics of a contract is defined as

$$[\![C]\!] := [\![A]\!]^{Cmpl} \cup [\![G]\!], \tag{1}$$

where $(X)^{Cmpl}$ defines the complement of a set X in some universe $\mathcal{U}$, and $[\![X]\!]$ is defined as the semantic interpretation of X. In our case, $[\![X]\!]$ is given in terms of sets of timed traces. A trace over an alphabet Σ is a sequence of events. Further,

a time sequence τ is a monotonically increasing sequence of real values, such that for each $t \in \mathbb{R}$ there exist some $i \geq 1$ such that $\tau_i > t$. A timed trace is a sequence (ρ, τ) where ρ is a sequence of events and τ a time sequence. The set of all timed traces over Σ is denoted by $Tr(\Sigma)$.

The specification S of a component is given in terms of a set of contracts, i. e. $[\![S]\!] := \bigcap_{i=1}^{n} [\![C_i]\!]$. An implementation I of a component satisfies its specification S, if $[\![I]\!] \subseteq [\![S]\!]$ holds. The refinement relation between two contracts C and C' is defined as follows:

$$C' \text{ refines } C, \text{ if } [\![A]\!] \subseteq [\![A']\!] \text{ and } [\![G']\!] \subseteq [\![G]\!], \tag{2}$$

When a component is decomposed into a set of subcomponents, we have to check whether the overall contract $C = (A, G)$ (which also will be called *global*) and all subcontracts $C_i = (A_i, G_i)$ (also called *local*) for $i \in \{1, ..., n\}$ are consistent. Under the assumption, that for all local assumptions A_i which have a non-empty intersection with the global assumption A the subset relation $A_i \subseteq A$ holds, we have to check the following virtual integration condition:

$$\begin{aligned} i)\ A \wedge G_1 \wedge ... \wedge G_n \ &\Rightarrow\ A_1 \wedge ... \wedge A_n \\ ii)\ A \wedge G_1 \wedge ... \wedge G_n \ &\Rightarrow\ G. \end{aligned} \tag{3}$$

An in-depth discussion about virtual integration can be found in [3]. In [3] contracts were extended by so called *weak* assumptions. Weak assumptions are used to describe a set of possible environments in which the component guarantees different behaviors. This separation is only methodological, and does not affect the semantics of the definition of the original contracts: Let $C = (A_s, A_{w_1}, ..., A_{w_n}, G_1, ..., G_n)$ be a contract consisting of a strong assumption A_s, a set of weak assumptions A_{w_i}, and a set of corresponding guarantees G_i for $i \in \mathbb{N}$. Semantically, we map C to a standard contract of the form $C' = (A_s, G)$, where $G = (G'_1 \wedge ... \wedge G'_n)$ $G'_i = (A_{w_i} \Rightarrow G_i)$.

III. Fault Tolerant Time Interval

The introduction of multiple design aspects was merely based on the distinct types of analyses that have been performed to verify a system. Timing and Safety analyses are the most popular aspects coming along a huge set of analysis techniques. Nevertheless, current safety standards like the ISO 26262 [2] force a coalescence of these aspects. One argument is the existence of the concept of *Fault Tolerance Time Intervals* (FTTI) in both aspects, creating a close connection.

The FTTI specifies the time between the occurrence of a fault and the hazardous event (see Fig. 1). The actual value is given by the intended use of the item and its physical environmental constraints. For example: during night driving the fault "loss of both front lights" has a FTTI from 10ms, i. e. within 10ms a proper reaction has to be performed to bring the system back into a safe state. A possible safe state in this example could be switching on the fog- and side-lights. When the 10ms are exceeded it might not be possible anymore for the driver to keep the car on the road in *any* road situation.

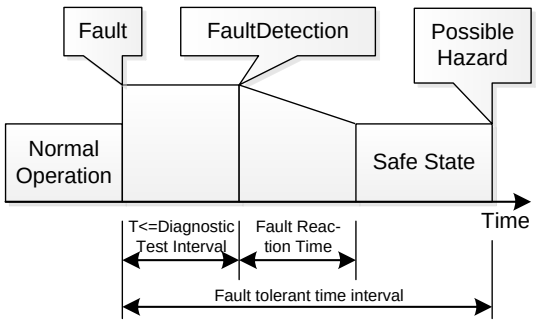

Fig. 1. Safety Related Timing [2]

Typically, the FTTIs are already known for all hazards during the hazard and risk assessment phase of an ISO 26262 process.

Based on the identification of the hazards, a functional safety concept is created. It describes the handling of faults and possible mitigation strategies on a purely functional level. This safety concept is later on refined into a technical safety concept realizing the specified safety properties in terms of hardware and software. During these safety concepts the FTTI is getting separated into the *diagnostic test interval*(DTI) and the *Fault Reaction Time* (FRT).

The refinement of the FTTI has a distinct relation to the timing properties of the System since the diagnostic capabilities of a system are often relying on periodic activation. A typical example is the monitoring of a component using a heartbeat signal. To detect total loss or communication faults a system is periodically asked to respond to a send request. If there is no answer in a defined time interval, the component is assumed to be defect and the proper counter measures can be activated.

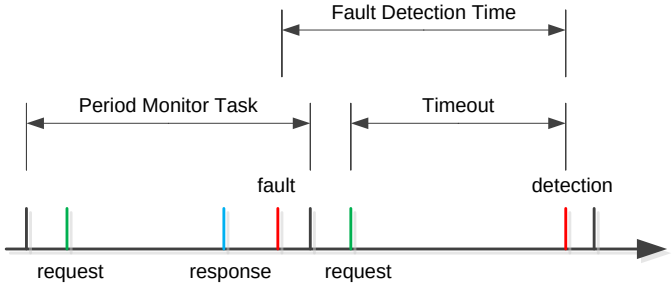

Fig. 2. Detecting a fault using a heartbeat monitor

Fig. 2 displays the relation between the fault detection time and the period/timeout of the monitor task. The figure also illustrates that a refinement step has been performed in which an initial requirement, which indicated the FTTI has been implemented using a watchdog, for one particular fault. The ISO 26262 requires that every refinement step in the safety concept is verified. This step can be automated using the virtual integration test (see Section IV for multi-aspect scenarios, saving time on verification activities while gaining accuracy as well.

IV. Virtual Integration Check using Timed Automata

We will introduce a technique to check the refinement relation of contracts in terms of the virtual integration condition as introduced in Section II-D. For this, we translate all contracts of the corresponding system to UPPAAL timed automata [28], and perform a reachability check [29]. Our refinement check was initially introduced in [30].

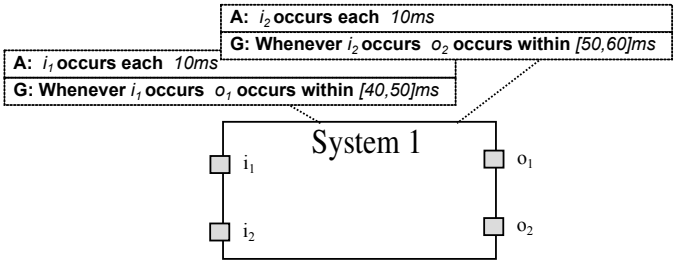

Fig. 3. System with two independent contracts.

For our analysis, we assume that the specification of a system or a component consists of either a single contract or a set of independent contracts that do not affect each other. Let S be a component, $\mathcal{C}_S$ its contract specification, and $\mathcal{P}(C)$ the set of ports of S the contract $C \in \mathcal{C}_S$ talks about. We assume that for the contract specification of a system it holds that $\bigcap_{C \in \mathcal{C}_S} \mathcal{P}(C) = \emptyset$. Consider for example the system specification in Fig. 3, which consists of two contracts. As both contracts talk about different ports, they do not affect each other, and thus can be treated independently from each other. Checking dependent contracts would be much more complicated, as we would have to build the conjugation of all dependent contracts, involving negation parts in the formulae. A discussion for this can be found in [30]. Note that contracts *between* systems can have dependencies.

A. From Contracts to Automata Networks

Let (A, G) be a global contract, and let (A_i, G_i) for $i \in \{1, ..., n\}$ be a set of local contracts. To check the condition of Equation 3, we derive a timed automaton O_{A_i} out of each local assumption A_i for $i \in \{1, ..., n\}$ serving as passive observer. Further, we derive a timed automaton O_G out of the global guarantee G. The transitions of each O_{A_i} for $i \in \{1, ..., n\}$ and O_G are annotated with receiving events and clock constraints in such a way that the observers accept the set of timed traces which are element of $[\![A_i]\!]$ and $[\![G]\!]$. For all traces which are not element of $[\![A_i]\!]$ (or $[\![G]\!]$) the corresponding observer enters a bad state.

Further, we derive an automaton T_A for the global assumption A and automata T_i with $i \in \{1, ..., n\}$ for each local guarantee. These automata serve as trigger for the observer automata. The transitions of T_A are annotated with sending events and timing constraints, such that T_A produces all traces that are element of $[\![A]\!]$. The automata for the local guarantees consist of both receiving and sending events (see next section for details and examples) and are called *transceiver automata* in the following.

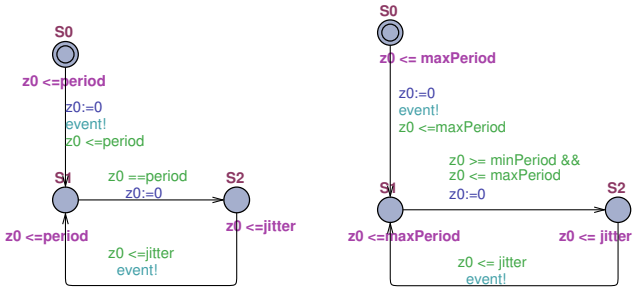

Fig. 4. Left: trigger automaton for periodic *R1* pattern; Right: trigger automaton for sporadic *R2* pattern.

From all automata we build the automaton network $S = T_A \parallel T_1 \parallel \ldots \parallel T_n \parallel O_{A_1} \parallel \ldots \parallel O_{A_n} \parallel O_G$. If one of the trigger automata produces a sequence which is not an element of $[\![A_1 \wedge ... \wedge A_n]\!]$ — and therefore the subset inclusion property of the local assumptions is violated — or not an element of $[\![G]\!]$ — where the subset property of the global guarantee is violated — a corresponding observer will enter a bad state. So, we need to check S against the following Uppaal query:

$$R = A\Box(\neg\, O_{A_1}.bad \wedge ... \wedge \neg O_{A_n}.bad \wedge \neg O_G.bad) \quad (4)$$

This formula states that a bad state of any of the observer is never reached. If the automaton network satisfies Formula 4, i. e. when $S \models R$, we have shown that the system architecture annotated with the local contracts can be virtually integrated in the context of the system with the global contract. In other words, the set of local contracts refines the global contract.

B. From Pattern to Timed Automata

Our prototype implementation transforms the RSL patterns introduced in Section II-C to timed automata. Note that for the sake of readability we will only consider single events instead of sets of events. The treatment of sets of events is straightforward.

The *R1* and *R2* pattern specifying some periodic and sporadic occurrence of some events are typically used as a trigger to the rest of the system. In the left part of Fig. 4 the trigger automaton for the *R1* pattern is illustrated. According to standard event streams as introduced in the real-time calculus [31] the initial event is fired within the interval $[0, period]$, and each successive event is fired every $period$ time unit. State S_2 is needed in order to appropriately handle jitters.

The trigger automaton for the sporadic *R2* pattern is illustrated in the right part of Fig. 4. Note that that the $maxPeriod$ has always to be defined. If not, the automaton could stay in the states S_0 and S_1 forever and we would obtain an unbounded automaton model. Similar to the previous case, the first event is fired within the interval $[0, maxPeriod]$. In contrast to the automaton of the *R1* pattern, each successive event

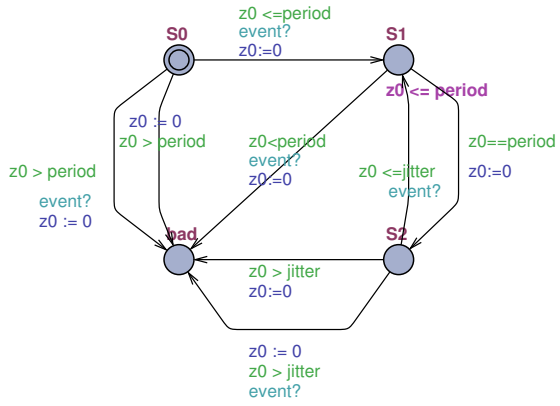

Fig. 5. Observer automaton for *R1* pattern.

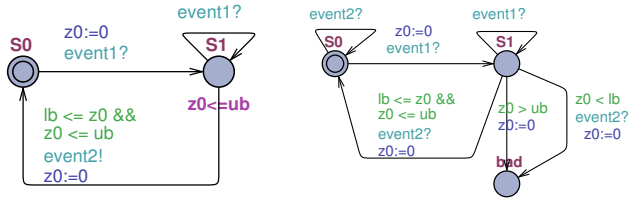

Fig. 6. Left: transceiver automaton for *R3* pattern; Right: observer automaton for *R3* pattern.

is fired within the time interval $[minPeriod, maxPeriod]$, delayed by some jitter.

In the case that an assumption containing an *R1* or *R2* pattern is refined, also observer automata for these pattern are required. In Fig. 5 the observer automaton for the *R1* pattern is depicted. In addition to the corresponding trigger automaton, it has a bad state. This state can be reached from the initial state, if the initial event from the environment arrives too late. From state S_1 the automaton can switch to the bad state, if the event arrives too early, and from state S_2 the bad state is reached, when the jitter exceeds its allowed value. This edge occurs twice, i. e. with and without the annotation of the corresponding event, because we want to prevent the automaton to get stuck in S_2 when an event is received by the time the jitter exceeds its allowed value and the automaton has not yet switched to the bad state. We omit the explicit treatment of the observer automaton for an *R2* pattern, as it is built analogously.

The *R3* pattern is used for both a transceiver or as a pure observer. The transceiver automaton for the *R3* pattern is illustrated in the left part of Fig. 6: Whenever an event from the environment is received, it sends an output event after a delay interval $[lb, ub]$ where $lb, ub \in \mathbb{N}$. Note that we only consider a *iterative* activation of our automata, i. e. we do not consider parallel activations so far, as we ignore each further activation in state S_1 with the aid of the self loop annotated by a corresponding receiving event. Our approach will be extended to deal with multiple activations. For that, we must allocate explicit indexes to the observable events. Also, we need to know the number of maximal parallel activations a priori, such that a corresponding number of automata can be instantiated.

The corresponding observer automaton for the *R3* pattern is depicted in the right part of Fig. 6: Whenever we receive an event from the environment, the automaton checks whether an event $event2$ is sent as a response. If such an event is send too early or too late, we switch to the bad state of the automaton. The transition $S_1 \rightarrow_{z0 \leq lb} bad$ is skipped, if $lb = 0$.

Note that we have skipped the discussion of how events are reproduced, i. e. events which are received by a set of automata instead of only a single one. This is actually realized by some *glue logic* consisting of small automata that multiply events.

V. Case Study

We evaluated our virtual integration test for multi-aspect (real-time and safety) designs using an industrial case study of a lane-keeping-support system (LKS). The LKS system is a driver assistant function that helps the driver to keep his car on the lane during the trip. A scenario in which the LKS is of particular importance is a tired or distracted driver loosening the grip of the steering wheel which leads to the car bearing away from the current lane. Since a car unintentionally leaving its lane may lead to a hazardous situation the LKS aims to prevent this from happening. So in order to keep the car on the lane the system evaluates (using a camera and driver monitoring) whether the driver changes the lane unintentionally and decides to intervene by steering the car back to the lane via a correction of the steering angle or via a short braking intervention on either side of the car. These two intervention options do not mutually exclude each other, they are more like variants depending on the actual variant configuration of the car that the LKS is integrated in.

For safety relevant automotive systems the ISO 26262 has to be applied as the current safety standard. Following the ISO 26262 process, timing requirements are already present at a very early stage of the development. A risk analysis is performed at the beginning of the system design, to identify the potential risks resulting from item malfunctions. The LKS uses an electronic power steering [32] to control the steering angle of the car. Therefore, the potential risk exists that the system performs a change in the steering angle unintentionally, leading to a crash of the vehicle. In order to evaluate the risk of this hazard the ISO 26262 considers three parameters: *Severity*, *Probability of Exposure*, and *Controllability*.

Considering controllability: In Fig. 7 the results of a user study regarding the occurrences of (permanent) faults in an electronic power steering are depicted. Depending on the force that is applied from the electronic motor to the steering axis the time varies in which the user realizes that there is a deviation and perceives the deviation as disturbing or even looses control over the car. To build a safe system safety goals are derived from such case studies that shall guarantee that faults do not lead to situations where the user cannot control the vehicle anymore. These safety goals are in this

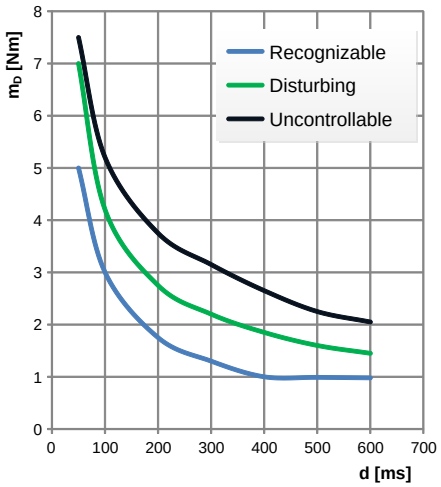

Fig. 7. Controllability of deviations in power steering systems [33].

case specifying the maximum time a fault may be present in the system unless the hazard occurs, the Fault Tolerance Time Interval (FTTI). This high level timing requirement is further refined during the development process. The ISO 26262 requires an analysis assuring this refinement has been performed correctly, making the virtual integration check a valuable tool for safety engineers.

We modeled the LKS using the CSM. The LKS-model introduced in [34] thereby served as a basis. The overall functional structure of the LKS system is displayed in Fig. 8. The functional *chain* starts at the video sensing unit which produces an uncompressed 2 mega-pixel 15 frames per second video stream. This video stream is used to extract line information which may indicate lanes of a street, so the data workload between video sensing and line detection is quite high. From there, rather small data amounts are passed down the chain. Line-to-lane-fusion extracts the actual data about the own and the neighboring lanes relevant for the car from the line data passed down from line detection. As the name suggests, the situation evaluation assesses the current situation based on additional sensor data (i. e. the current steering angle and other human machine interface data) and outputs the (un)intended trajectory as indicated by the sensor data. Additionally, situation evaluation forwards data about whether the trajectory is intended or unintended based on certain thresholds in the sensor data analysis. The trajectory planning function is the controlling part that compares the trajectory from the situation evaluation with the current trajectory of the car (analyzed using a yaw-rate sensor as source). Based on the offset between these two trajectories and the information about the intention of the driver it decides when and how to intervene. After the trajectory planning there is a breakdown in different options, depending on which hardware is actually used in the system (either ESP or EPS; neither ESP nor EPS; both ESP and EPS). In our case study we considered the latter.

Starting with this functional structure of the LKS system we refined our model by adding communication signals between the functions and deriving a logical structure from the

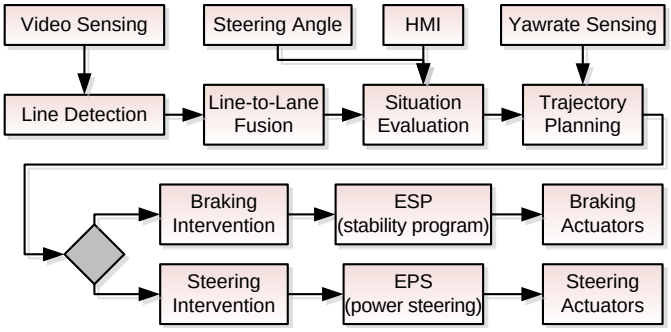

Fig. 8. Functional structure of the LKS case study.

functional one. Note that some signals are forwarded (i. e. connected) to multiple logical tasks (logical units performing calculations). The distinction between tasks and signals is a necessary refinement step to emphasize that communication induces behavior and thus takes time. This is of particular importance for the virtual integration test since the composition of signals and tasks along with their allocation to buses and processing units impacts the overall timing behavior. Note that we do not analyze the schedulability for allocated tasks and signals to buses and processing units in this paper. We only consider the system's specifications and its subsequent refinements in this work. In the following two subsections we describe the safety and timing specifications for the *Situation Evaluation* task in both the black-box and the white-box view.

A. Black-Box View of Situation Evaluation

The Situation Evaluation (SE) is concerned with assessing the current car situation. It calculates the currently set trajectory of the car based on the steering angle sensor input and the lane information. Additionally, it receives data from the human machine interface upon which it decides whether the driver steered intentionally or not.

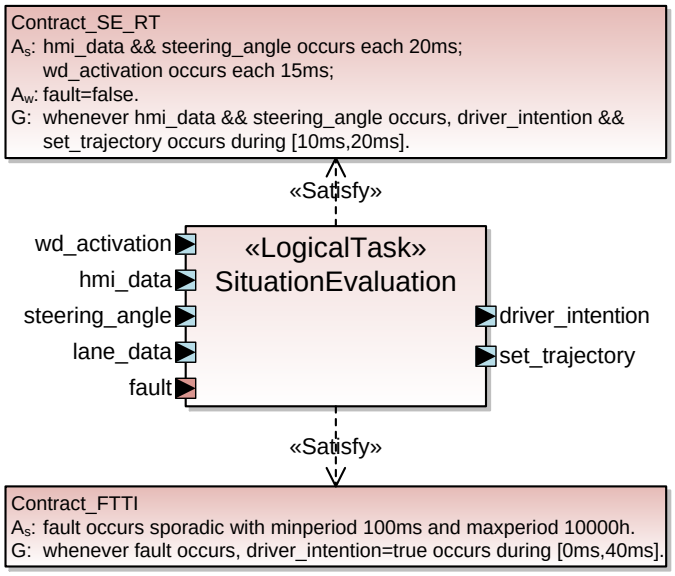

Fig. 9. Black-box view of the situation evaluation task.

Fig. 9 shows the black-box view of the SE task which satisfies the contract *Contract_FTTI* specified as follows: Assuming that a fault (re)occurs within certain time bounds (between 100 milliseconds and 10,000 hours) SE shall guarantee that within 40ms SE returns to a safe state (in this case: make sure that the LKS does not override the driver, i. e. it is the driver's intention to steer with the given steering angle). The guarantee part reflects the FTTI, derived from a previously performed safety analysis. The other contract *Contract_SE_RT* specifies the real-time behavior of SE. Here we assume that *hmi_data* and *steering_angle* are events that always occur together and trigger the SE task every 20ms. Furthermore, we assume that the internal fault detection mechanism is triggered every 15ms. With the weak assumption we ensure that we only have to ensure the guarantee if there is no fault. The guarantee of that contract states that the delay between triggering and answering shall be between 10ms and 20ms.

B. White-Box View of Situation Evaluation

The white-box view of the SE task gives more detail on the single sub-tasks and reveals the internal safety mechanisms. In Fig. 10 we see the sub-tasks and their requirements. The safety relevance of the system has already been discussed, leading to a need for architectures able to detect and mitigate faults. To reduce the complexity, we just focus on one single fault in the SE component: a permanent crash of the SECalculation sub-task. Faults are modeled as additional ports, where a signal may arrive sporadically between 100ms and 10,000h, representing the occurrence of the fault. In the implementation we currently make sure that all faults are permanent, i. e. once they occurred they cannot disappear on their own. To be able to detect this fault a watchdog is used, which is activated every 15ms. Right after its activation it sends a request to the SECalculation unit and waits for a response. If the SECalculation is not responding the component is considered broken and the faulDetected port is set to *true*.

The SEFaultReact sub-task is a typical override component that replaces the signals coming from the SECalculation sub-task by the safe-state if the component is faulty. Considering that we want to avoid that the whole system causes a steering angle correction without need, the safe-state is given by setting the *driver_intention* signal to *true*. The following components will not try to attempt a correction. In addition, the driver needs to be informed about the reduced functionality until it may be resolved by a reset. Driver warning and reset mechanisms are not modeled here to keep the example readable.

There are two contracts for the calculation routine in the bottom of the figure specifying the real-time property of a deadline in *Contract_SECalculation_RT* and the watchdog behavior in the fault-free case in *Contract_SECalculation_NoFault*. The calculation task offers two additional ports *req* and *ack* to allow for the watchdog-based fault recognition.

In the upper part of the figure there are two contracts each for both SEWatchDog and SEFaultReact. *Contract_FautlDetection* and *Contract_AllOK* specify how the fault is detected by the watchdog and which corresponding

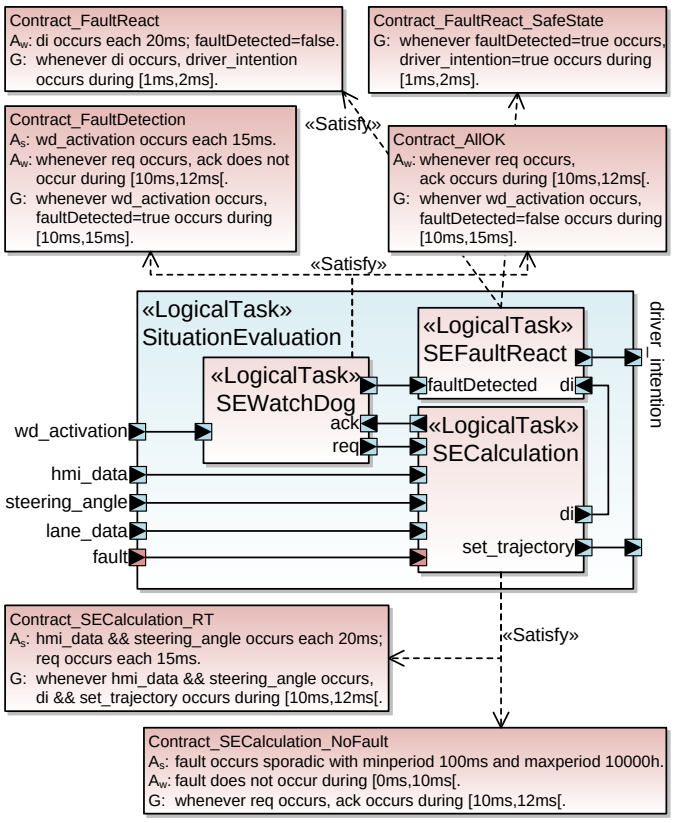

Fig. 10. White-box view of the situation evaluation task.

output events are sent to the fault reaction component. Note that we assumed here that the *wd_activation* and the *req* are the same signal (i. e. a *req* is sent by the watchdog immediately after it has been activated).

Contract_FaultReact specifies the required real-time delay between the triggering of SEFaultReact by the *di* event and the *driver_intention* event. *Contract_FaultReact_SafeState* states that if a fault is detected (indicated by the signal received from the watchdog) the safe state shall be reached within 1ms to 2ms. As mentioned above safe state means that we make sure that the LKS does not interfere with the driver's commands.

C. Verification

To check the decomposition of our case study we apply and adapt the transformation of the considered RSL patterns illustrated in Section IV and our corresponding prototype implementation. To obtain independent contracts as discussed in Section IV for the use case scenario, we consider that the evaluations of the ports *driver_intention=false*, *driver_intention=true*, *faultDetect=true*, and *faultDetect=false* are all different events, and rename *driver_intention=true* to *safeState*, *driver_intention=false* to *ok*, *faultDetect=true* to *falutDetection*, and *faultDetect=false* to *noFault*. Our system model assumes that sporadically there could occur a fault. We are restricted to use 15-bit integers through the usage of UPPAAL. In the use case scenario, the upper bound of the sporadic activation of the fault was specified as 10,000h. For the generated automaton we use the maximum allowed value

32767. This makes no difference, as the value is still much larger than the defined delay times.

Our prototype implementation builds a trigger automaton out of all assumptions of the overall *SituationEvaluation* component as illustrated in Section IV, specifying that events *hmi_data* and *steering_angle* occur periodically with the period value 20, and that event *fault* occurs sporadically within the interval $[100, 32767]$. Further, the tool derives a pure observer automaton $O_{G_{SE\_RT}}$ out of the guarantee of the *SituationEvaluation* component, specifying a global deadline between the occurrence of the events *hmi_data*, *driver_intention* and *steering_angle*, *set_trajectory*. Further, the FTTI is specified to be between $[0, 40]$ time units leading to the observer automaton $O_{G_{FTTI}}$.

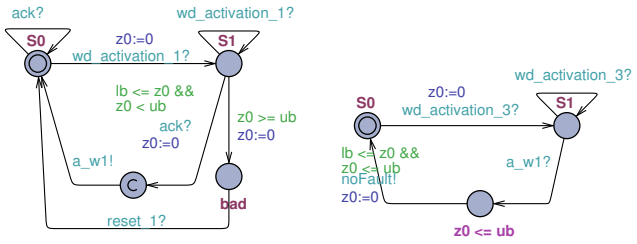

Fig. 11. Weak assumption (left) and corresponding guarantee (right) of *Contract_AllOk*.

In the next step our tool builds the automata obtained from the contracts of the decomposition structure. In detail, for the *SEWatchDog* component we get an *R1* observer $O_{FaultDetection}$ for its local strong assumption from *Contract_FaultDetection*. Further, we get a set of automata for each weak assumption guarantee pair. For this, we have to adapt the automata transformation in such a manner, that an automaton obtained from the weak assumption can interact with its corresponding guarantee automaton. For this, consider both automata of Fig. 11 obtained from *Contract_AllOk*: The weak assumption is specified with an *R3* pattern. If the ack signal from the system is received within the specified interval the weak assumption holds and its corresponding automaton depicted in the left part of Fig. 11 triggers the corresponding guarantee automaton. To realize this triggering, we adapted the transformation in such a way that an intermediate committed state is entered whenever this message is received in time and immediately left again triggering the automaton of the guarantee. When the event *ack* is not received in time, the automaton switches to its *bad* state which represents an inactive state. That is, the weak assumption does not hold in such a case and its guarantee is not activated. The guarantee automaton illustrated in the right part of Fig. 11 is also extended through an intermediate state, such that it can receive events from the automaton of the weak assumption and can send its *ok* signal within the specified time bound, i. e. $lb = 10$ and $ub = 15$.

In Fig. 12 the automata obtained from *Contract_FaultDetection* are illustrated. The left part of the figure illustrates the automaton obtained from the weak assumption. The transformation is adapted in such a manner that if the event *ack* from the system is received in time the automaton switches to its *bad* state (again representing its inactive state where the weak assumption does not hold). If the event *ack* is not received in time, its guarantee automaton is triggered which then sends a fault detection event within the values $lb = 10$ and $ub = 15$. All other automata of the decomposition structure are build in the same manner.

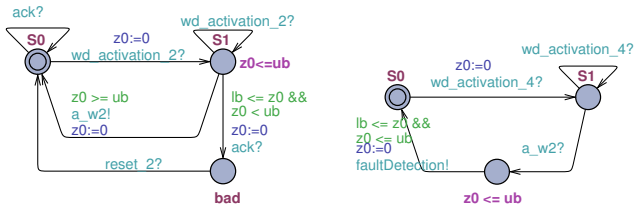

Fig. 12. Weak assumption (left) and corresponding guarantee (right) of *Contract_FaultDetection*.

The UPPAAL query asks for the reachability of all observer automata. For reasons of clarity, we split the overall query in the following parts:

- Observer of the guarantees of the overall component *SituationEvaluation*:

$$A_1 = \neg(O_{G_{SE\_RT}}.bad \ \vee \ O_{G_{FTTI}}.bad),$$

- Observer of assumptions of the component *SECalculation*:

$$A_2 = \neg \ (O_{SECalculation\_RT}.bad \ \vee \ O_{SECalculation\_NoFault}.bad),$$

- Observer of assumptions of the components *SEWatchDog* and *SEFaultReact*:

$$A_3 = \neg(O_{FaultDetection}.bad \ \vee \ O_{FaultReact}.bad).$$

We get the following UPPAAL query:

$$A\Box \ (A_1 \wedge A_2 \wedge A_3) \ \wedge \ \neg deadlock.$$

The deadlock property is needed since the time intervals for the transceiver automata could be defined in such a manner that the event propagation process cannot be completed. Consider for example that the delay interval between the events *req* and *ack* would be specified in such a manner that its lower bound is larger than the delay interval between the events *wd_activation* and *ok*. With this, the guarantee automaton of Fig. 11 would remain in state $S1$. The analysis of this property with UPPAAL took about 30ms of verification time.

VI. Conclusion & Outlook

We discussed that independence of system analyses on different aspects is typically not given on the example of the fault tolerance time interval. We presented our timed automata based virtual integration test to prove the correct refinement of contract-based safety and real-time requirements. This technique was applied to an automotive lane-keeping-support system leading to a complex timed automaton network. We

still need to integrate more concepts from the safety perspective: We are currently limited to permanent faults, leaving transient ones aside. Additionally, we focused on *crash* as a single fault type, which leads to a situation in which the affected component may not react at all anymore. This scope needs to be extended to also include other fault types. A delay, causing a violation of timing requirements is a straight forward extension, as well as the introduction of value faults, causing calculations to produce wrong results. The combined specification of safety and timing requirements may be used for other analyses than virtual integration. Depending on the overall load of the controller, on which the presented watch dog of the system runs, the fault detection time may vary. To verify these times, we will apply our scheduling analysis technique [35].

ACKNOWLEDGMENT

This work was partly supported by the European Commission funding the Large-scale integrating project (IP) proposal under ICT Call 7 (FP7-ICT-2011-7) "Designing for Adaptability and evolutioN in System of systems Engineering (DANSE)" (No. 287716), the German Federal Ministry of Education and Research under grant number 01IS12005M, the ARTEMIS Joint Undertaking within the European project MBAT under grant agreement No. 269335, the German Federal Ministry of Education and Research (BMBF) under grant number 01IS11003L. The responsibility for the content of this publication lies with the authors.

REFERENCES

[1] I.-S. S. B. IEEE Std 1471-2000, "Ieee recommended practice for architectural description of software-intensive systems," September 2000.

[2] ISO26262, "Road vehicles – functional safety," 2011.

[3] W. Damm, H. Hungar, B. Josko, T. Peikenkamp, and I. Stierand, "Using contract-based component specifications for virtual integration testing and architecture design," in *DATE Exhibition*, 2011, pp. 1–6.

[4] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple viewpoint contract-based specification and design," in *Formal Methods for Components and Objects*. Springer Berlin Heidelberg, 2008, vol. 5382, pp. 200–225.

[5] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen, "Contracts for systems design," in *Inria Research*, 2012.

[6] P. Reinkemeier, I. Stierand, and P. R. S. Henkler, "A pattern-based requirement specification language: Mapping automotive specific timing requirements," in *Software Engineering 2011 Workshopband*, R. Reussner, A. Pretschner, and S. Jhnichen, Eds. Gesellschaft für Informatik e.V. (GI), 2011, pp. 99–108.

[7] M. Oertel, A. Mahdi, E. Böde, and A. Rettberg, "Contract-based safety: Specification and application guidelines," in *Proceedings of the 1st International Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC 2014)*, 2014.

[8] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," *Microprocess. Microprogram.*, vol. 40, pp. 117–134, April 1994.

[9] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study," *Comput. Sci. Inf. Syst.*, vol. 10, no. 1, pp. 453–482, 2013.

[10] J. Maki-Turja and M. Nolin, "Efficient implementation of tight response-times fortasks with offsets," *Real-Time Systems*, vol. 40, no. 1, pp. 77–116, 2008.

[11] L. Thiele, S. Chakraborty, M. Gries, A. Maxiaguine, and J. Greutert, "Embedded software in network processors - models and algorithms." Springer Verlag London, UK, 2001, pp. 416 –434.

[12] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *Computers and Digital Techniques, IEE Proceedings* -, vol. 152, no. 2, pp. 148–166, Mar 2005.

[13] E. Fersman, P. Pettersson, and W. Yi, "Timed automata with asynchronous processes: schedulability and decidability," in *Proceedings of TACAS*. Springer, 2002.

[14] A. David, J. Illum, K. G. Larsen, and A. Skou, "Model-based framework for schedulability analysis using uppaal 4.1," in *Model-Based Design for Embedded Systems*, G. Nicolescu and P. Mosterman, Eds., 2009.

[15] A. Muniz, A. Andrade, and G. Lima, "Integrating uml and uppaal for designing, specifying and verifying component-based real-time systems," *Innovations in Systems and Software Engineering*, vol. 6, no. 1-2, pp. 29–37, 2010.

[16] A. Pasquini, Y. Papadopoulos, and J. McDermid, "Hierarchically performed hazard origin and propagation studies," in *Computer Safety, Reliability and Security*, ser. Lecture Notes in Computer Science, K. Kanoun, Ed. Springer Berlin / Heidelberg, 1999, vol. 1698, pp. 688–688, 10.1007/3-540-48249-0_13.

[17] Y. Papadopoulos and M. Maruhn, "Model-based synthesis of fault trees from matlab-simulink models," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, 2001, pp. 77–82.

[18] K. Echtle, *Fehlertoleranzverfahren*. Springer-Verlag, 1990.

[19] L. Lamport and S. Merz, "Specifying and verifying fault-tolerant systems," in *Formal Techniques in real-time and fault-tolerant systems*. Springer, 1994, pp. 41–76.

[20] A. Cimatti, M. Dorigatti, and S. Tonetta, "Ocra: A tool for checking the refinement of temporal contracts," in *ASE IEEE*, 2013, pp. 702–705.

[21] A. Baumgart, P. Reinkemeier, A. Rettberg, I. Stierand, E. Thaden, and R. Weber, "A model-based design methodology with contracts to enhance the development process of safety-critical systems," in *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, ser. SEUS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 59–70.

[22] The SPEEDS Consortium, "SPEEDS Project," http://speeds.eu.com.

[23] *OMG Systems Modeling Language (OMG SysML ^{TM})*, Object Management Group, November 2008, version 1.1.

[24] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[25] B. Josko, Q. Ma, and A. Metzner, "Designing Embedded Systems using Heterogeneous Rich Components," *Proceedings of the INCOSE*, 2008.

[26] Project SPEEDS: WP.2.1 Partners, "SPEEDS Meta-model Behavioural Semantics — Complement do D.2.1.c," The SPEEDS consortium, Tech. Rep., 2007.

[27] P. Reinkemeier, I. Stierand, P. Rehkop, and S. Henkler, "A pattern-based requirement specification language: Mapping automotive specific timing requirements," in *Software Engineering 2011 - Workshopband*, ser. LNI. GI, 2011.

[28] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer STTT*, vol. 1, no. 1-2, pp. 134–152, 1997.

[29] L. Aceto, A. Burgueño, and K. Larsen, "Model checking via reachability testing for timed automata," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, B. Steffen, Ed. Springer Berlin / Heidelberg, 1998, vol. 1384, pp. 263–280, 10.1007/BFb0054177.

[30] T. Gezgin, R. Weber, and M. Girod, "A refinement checking technique for contract–based architecture designs," in *Fourth International Workshop on Model Based Architecting and Construction of Embedded Systems*, 10 2011.

[31] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 4, 2000, pp. 101 –104 vol.4.

[32] G. Reimann, P. Brenner, and H. Büring, *Handbuch Fahrerassistenzsysteme*. ViewegTeubner Verl, 2012, ch. Lenkstellsysteme, pp. 287 – 311.

[33] M. Suerken and T. Peikenkamp, "Model-based application of iso 26262: The hazard and risk assessment," SAE, Tech. Rep., April 2013.

[34] R. Weber, E. Thaden, S. Henkler, J. Höfflinger, and S. Prochnow, "Design space exploration for an industrial lane-keeping-support case study," in *Proceedings of DATE Conference – University Booth*, 24 - 28 Mar. 2014.

[35] T. Gezgin, I. Stierand, S. Henkler, and A. Rettberg, "State-based scheduling analysis for distributed real-time systems," *Design Automation for Emb.Sys.*, pp. 1–18, 2013.