Scalable Analysis of Real-Time Requirements

Vincent Langenfeld, Daniel Dietsch, Bernd Westphal, Jochen Hoenicke
Department of Computer Science, University of Freiburg
Email: {langenfv,dietsch,westphal,hoenicke}@informatik.uni-freiburg.de
Amalinda Post
Robert Bosch GmbH, Email: Amalinda.Post@de.bosch.com

***Abstract*—Detecting issues in real-time requirements is usually a trade-off between flexibility and cost: the effort expended depends on how expensive it is to fix a defect introduced by faulty, ambiguous or incomplete requirements. The most rigorous techniques for real-time requirement analysis depend on the formalisation of these requirements. Completely formalised real-time requirements allow the detection of issues that are hard to find through other means, like real-time inconsistency (i.e., "do the requirements lead to deadlocks and starvation of the system?") or vacuity (i.e., "are some requirements trivially satisfied"). Current analysis techniques for real-time requirements suffer from scalability issues – larger sets of such requirements are usually intractable. We present a new technique to analyse formalised real-time requirements for various properties. Our technique leverages recent advances in software model checking and automatic theorem proving by converting the analysis problem for real-time requirements to a program analysis task. We also report preliminary results from an ongoing, large scale application of our technique in the automotive domain at BOSCH.**

I. INTRODUCTION

The specification of requirements is a critical activity in software and system development because the set of requirements is supposed to partition the set of possible systems into acceptable and non-acceptable (or correct and incorrect) systems. A defect in a requirements specification can lead to a situation where a software or system is delivered that is *formally correct*, i.e., that satisfies the (erroneous) requirements but does not satisfy the customer's needs. To mitigate the risk of defects in requirements specification, the analysis of requirements specifications for generic properties like consistency or unambiguity is recommended [12]. An established approach to the analysis of requirements for generic properties is the review. Yet reviews, as a human activity, are error-prone and do not scale well for large sets of requirements.

Recent works have proposed automated formal analyses of requirements for generic properties (e.g., [5], [9], [16], [17]). In order to apply automated formal analyses, two problems need to be solved. Firstly, requirements need to be specified in a precise, formal, machine-readable language. There are efforts to bridge the gap between the classical formal behaviour specification languages, like different temporal logics, and the practice of requirements engineers in industry contexts. For example, Crapo et al. [16] propose the ASSERT language, Teige et al. [5] advocate the so-called Universal Pattern [22], and Post et al. [20] have developed a restricted English grammar based on the specification pattern system by Konrad et al. [13]. Secondly, scalable algorithms for the actual analysis for generic properties are necessary.

In this work, we address the second problem of analysing requirements for generic properties. We consider *real-time* requirements, that is, requirements that specify the behaviour of systems which have to react to inputs within given time bounds in continuous time. We assume that the requirements to be analysed are specified using restricted English grammar [20] with the underlying Duration Calculus [6] semantics. The generic properties of requirements that we consider are partly specific to real-time requirements. The most prominent considered property is *real-time-consistency* [18], a property that implies but is not equivalent to general consistency. An rt-inconsistency in a requirements specification is particularly hard to spot in large sets of requirements and may only manifest during implementation and hence incur unnecessary effort and cost. It has been shown [17] that rt-inconsistency does occur in industrial requirements specifications and that it is often the source of difficult-to-detect defects in later stages of system development. Post et al. [18] have shown rt-inconsistency to be decidable by a reduction to real-time model-checking [2]. The reduction inherits the theoretical exponential worst-case complexity and does not scale well for larger sets of requirements.

In this work, we present a new algorithm for the analysis of a given set of real-time requirements for, among others, rt-inconsistency. The new algorithm is based on a reduction of the rt-inconsitency problem to a program analysis problem and also supports analyses for the generic requirements properties *vacuity* and *consistency* (both in their elaboration for real-time requirements). We have implemented the new algorithm in the open-source program analysis framework ULTIMATE [1] in the form of the tool ULTIMATE REQANALYZER. We evaluated the performance of REQANALYZER on a selection of real-time requirements specifications from two sources. One source is the benchmarks provided in previous works regarding analysing real-time requirements [18], [19]. The other source is an ongoing collaboration with an industry partner from the automotive domain. The new tool REQANALYZER has three major improvements over previous approaches. Firstly, it is able to analyse significantly larger sets of real-time requirements than the original approach [18]. Secondly, the new

The third author was supported by the DFG, reference no. WE 6198/1-1.

[1] https://github.com/ultimate-pa/ultimate

algorithm provides gradual results (in contrast to the original approach that terminates with an inconclusive result if time or memory resources are exceeded) and thereby promises to be effectively applicable to industry scale real-time requirements specifications. Thirdly, the algorithm allows us to pinpoint the source of an issue, i.e., it does not only yield *whether* an issue is present, but also where.

The paper is organised as follows. Section II recalls the property rt-inconsistency and motivates the need for scalable property analysis tools. Section III provides preliminaries for the reduction of real-time requirements analysis to a program analysis problem in Sections IV to VI. In Section VII, we present and discuss preliminary results from an ongoing, extensive, industrial scale case-study with the new real-time requirements analysis tool in the automotive domain. Section VIII discusses related work and Section IX concludes.

II. Problem Analysis

In this section we give an overview over the formalisation of requirements using restricted English grammar, the formal semantics of patterns from this grammar in terms of phase event automata (PEA), and properties to analyse a requirements specification for [17]–[20]. We motivate the need for a new scalable automated analysis of the considered properties for a set of requirements.

A. Requirements Formalisation

Following [17], [20], requirements are formalised using sentence patterns that are parameterised by expressions over system observables and durations. In the following examples we limit the discussion to boolean observables for simplicity, although our overall approach supports a much richer set of types of observables. For example, consider a system with boolean observables `A`, `B`, and `C`. In order to require that the value of `A` is not *true* at all times on all computations of the system we would write

req_1: *Globally, it is* ***never*** *the case that* '**`A`**' *holds.*

The requirement that `A` evaluates to *true* at all times on all computations of the system would be formalised as

req_2: *Globally, it is* ***always*** *the case that* '**`A`**' *holds.*

Timing requirements can be specified, e.g., with the following two patterns. The bounded response requirement

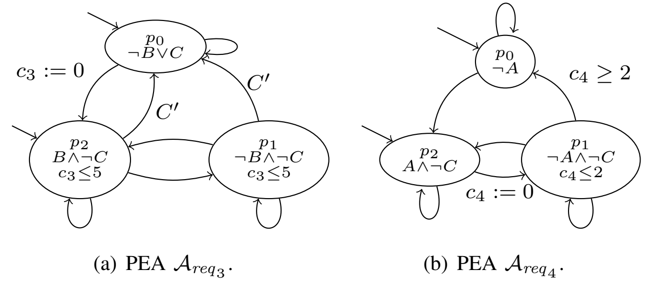
req_3: *Globally, it is always the case that if* '**`B`**' *holds then* '**`C`**' *holds after* ***at most*** '**`5`**' *time units.*

is satisfied by systems where, on each computation, each phase where `B` evaluates to *true* for a positive duration is followed by a phase where `C` evaluates to *true* after 5 time units the latest. The duration of phases can be constrained as follows:

req_4: *Globally, it is always the case that if* '**`A`**' *holds then* '**`!C`**' *holds for* ***at least*** '**`2`**' *time units.*

This requirement is satisfied by systems where each phase with positive `A` is followed by a phase of `C` evaluating to *false* for at least 2 time units.

The formal semantics of the patterns shown above is given by a phase event automaton (PEA). A PEA is a timed



(a) PEA $\mathcal{A}_{req_3}$. (b) PEA $\mathcal{A}_{req_4}$.

Fig. 1. Formal semantics of req_3 and req_4.

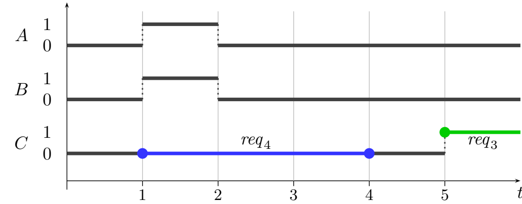


Fig. 2. `A` and `B` occur at the same point in time for one time unit, then `!C` for two time units satisfies req_4, and `C` occurring at time 5 satisfies req_3.

automaton and uses clocks to measure passing of time. A clock is a special variable that is automatically incremented when time passes and can be reset to zero by the transitions of the automaton. Figure 1 shows the PEAs for requirements req_3 and req_4 introduced above. The PEA $\mathcal{A}_{req_3}$ uses the clock c_3 to measure the five time units specified in requirement req_3. The clock variable is reset to zero, when B changes from false to true while C is false. The invariants $c_3 \leq 5$ in the bottom two locations ensure that these locations must be left at the latest when five time units have passed. The transitions leaving these locations check that C holds.

The runs of a PEA are modelled as timed sequences of observables over dense real-time, i.e., each element of the sequence gives a valuation of the considered observables and a duration. A set of pattern instances is modelled by the parallel composition of the PEAs of the requirements in the set (cf. Section III for details). For the full set of patterns the reader is referred to [17], [20] since the technical discussion in the subsequent sections directly works on the PEAs that represent a set of requirements.

B. Properties of Real-Time Requirements

Based on this definition of the semantics of a set of requirements specified using patterns, a number of generic properties for requirements are defined. The most natural generic property of a set of requirements may be *consistency*. A set of requirements is consistent if the specified set of interpretations is not empty, that is, if there is at least one interpretation satisfying all requirements in the set.

Requirements req_1 and req_2 shown above are obviously not consistent: there is no interpretation where observable `A` evaluates to *true* and to *false* at each point in time. The set $\{req_3, req_4\}$ is an example for consistent requirements.

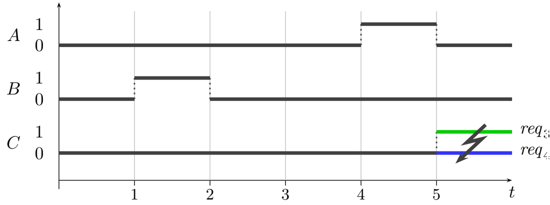


Fig. 3. This execution leads to an inconsistency at time five: Requirement req_3 allows the system to delay the reaction to B until time 5, yet since A occurs at time four, requirement req_4 needs !C at time 5. From time four (and the occurrence of A) on, the system steers toward inevitable rt-inconsistency.

Figure 2 gives (in form of a timing diagram) an example of an interpretation of A, B, and C that satisfies both requirements.

Yet the set $\{req_3, req_4\}$ is not rt-consistent. The property of rt-(in)consistency has been introduced in [18]. A set of requirements is *rt-inconsistent* if and only if there exists a finite prefix of an interpretation (modelling a system computation) such that there does not exist any extension of this prefix which satisfies all requirements in the set. The rt-inconsistency of $\{req_3, req_4\}$ can be seen by the prefix on the interval $[0, 5]$ of the interpretation shown in Figure 3. If A and B change values as shown in the figure, then at time 5, requirement req_4 would only be satisfied if C remained *false* while requirement req_3 would only be satisfied if C changed to *true*. Thus it is not possible to extend the prefix of the interpretation beyond time 5 without violating either req_3 or req_4.

In general, rt-consistency implies consistency but rt-consistency is a stronger property. Note that the rt-inconsistency of the particular set $\{req_3, req_4\}$ can be resolved by adding the following requirement (that disallows behaviour that leads to the identified rt-inconsistency):

> *Globally, it is always the case that if '**B**' holds then '**!A**' holds for **at least** '**5**' time units.*

For the industrial practice it is relevant to detect rt-inconsistencies already during requirements analysis, because rt-inconsistent requirements may lead to erroneous system behaviour that is hard to detect. From the perspective of pure consistency, the set of requirements $\{req_3, req_4\}$ could be read as the request to the developers to develop a system that has only satisfying computations. Such computations exist (as shown above), yet it is in general hard to see which computations these are: If requirements req_3 and req_4 are, e.g., realised by different parts of a program, these parts may just write a boolean variable C. One part with the intention to satisfy req_4 and one part with the intention to satisfy req_3, the actual value of C will depend on the scheduling of these parts. The resulting system would clearly be *incorrect* since it does not deal correctly with the situation shown in Figure 3. Yet the resulting system has *some* computations that perfectly satisfy both requirements. Uncovering such a defect depends on the timing of changes on A and B. Hence, there is a significant risk of not detecting the defect during quality assurance.

Next to consistency and rt-consistency, we consider vacuity of real-time requirements. Intuitively, a requirement is vacuous in a set of requirements if the behaviour it specifies cannot be triggered in a system satisfying all requirements. A vacuous requirement is similar to dead code in an implementation in the sense that is could be removed without changing the meaning. An example of a vacuous set of requirements is $\{req_1, req_4\}$, where req_4 is vacuous. In order to satisfy req_1, the value of A will continuously be *false* in all correct computations. If A is continuously *false* in all computations, then the premise of req_4 is never satisfied and thus req_4 is trivially satisfied. Detecting vacuous requirements is practically relevant since the vacuous requirement may indeed be unnecessary (then it should be removed from the set of requirements to simplify the overall specification) or it may be the case that other requirements erroneously cause the vacuity (then there is a defect in the requirements specification, the specification may not validly reflect customer needs).

C. Analysis of Real-Time Requirements for Properties

Note that it is not sufficient to check whether the pre-conditions of two real-time requirements with inconsistent effects are independent, as this would neither uncover all rt-inconsistencies (rt-inconsistencies may as well involve, e.g., an invariant) nor would all uncovered problems actually be rt-inconsistencies (as other requirements often prevent the rt-inconsistency from happening). Post et al. [18] present an analysis for rt-inconsistency by reducing the rt-inconsistency problem to a real-time model-checking problem. Their approach has shown that it is principally feasible to detect rt-inconsistency for industrial requirements as demonstrated in the case-study reported on in [17].

Yet the case-study [17] already points out two issues with this particular approach. Firstly, the analysis of [18] does not scale well. The case-study article does not report individual time consumption figures but reports that the longest execution time took over 90 minutes. Secondly, some analysis are reported to have terminated prematurely due to memory exhaustion. In this case, the analysis of [18] yields an inconclusive result. We can then neither conclude rt-consistency nor rt-inconsistency.

In this work we present a technique to overcome these two problems. We present an automated method that allows us to handle much larger sets of requirements while at the same time being able to pin-point which requirements of a large set are rt-inconsistent and why. Our method leverages recent advances in automated theorem proving as well as automatic software model checking by reducing the task of checking the requirements to a program analysis task.

The new algorithm furthermore allows us to compute partial results by analysing all subsets consisting of real-time requirements together with all untimed requirements of a set of requirements R of a given size k. For example, for $k = 2$, all pairs of real-time requirements from R would be considered and one would thus detect all rt-inconsistencies between two real-time requirements, even if their observables are only related through untimed requirements in R. Because checking each subset can be performed with a separate resource limit,

we can provide partial results for a given set of requirements. While these results are weaker than a complete analysis, they allow us to find and fix defects and thus increase confidence in the quality of the considered requirements. In practice, it can also be expected that most rt-inconsistencies can be found with small values of k.

III. Preliminaries

Our approach leverages the power of state-of-the-art program verifiers by translating the requirements given as a set of PEAs into a program. In this section we will give the formal definitions needed to describe this translation, namely the definition of programs and their executions, as well as the definition of PEAs and their runs.

We represent a program as a control flow graph whose edges are labelled with statements defined by the following grammar.

```
s ::= assume bexpr | v := expr | havoc v | s;s
```

These statements represent a small subset of the intermediate verification language **Boogie** [15], which uses these statements to express assumptions, assignments, non-deterministic assignments and sequential composition. Given a set of typed variables Var, $\texttt{v} \in Var$ is a variable, `expr` is an expression over Var, and `bexpr` is a Boolean expression over Var. For brevity, we do not define the complete expression syntax but rather assume the availability of Boolean and Real expressions and the usual logical and arithmetic operators.

A **program** is a labelled graph $\mathcal{P} = (Loc, \delta, \ell_0)$ with

- Loc being a set of nodes, called locations,
- $\delta \subseteq Loc \times Stmt \times Loc$ being a finite set of edges labelled with statements, and
- $\ell_0 \in Loc$ being an initial location.

Note that control-flow statements like `while(bexpr)` and `if(bexpr)` are represented by the graph structure, i. e., by loops and branching in the graph guarded by assume statements on the Boolean expression `bexpr`.

The behaviour of the program is given by traces and executions. A **trace** of a program $\mathcal{P}$ is a sequence of statements $\tau = st_0, st_1, st_2, ...$ so that there is a path in $\mathcal{P}$ labelled with τ. A **program state** $\sigma : Var \rightarrow \mathcal{D}$ is a function from the set of variables Var into the set of value domains $\mathcal{D}$ of the variables. The update of a program state $\sigma' = \sigma[v \mapsto d]$ with $v \in V$ and $d \in \mathcal{D}$ is the valuation where $\sigma'(v) = d$ and $\sigma'(v') = \sigma(v')$ for $v' \neq v$. With $\sigma(\texttt{expr})$ we denote the result of the evaluation of `expr` in the program state σ.

In order to describe the behaviour of a program, we define for each statement $st \in Stmt$ a binary **successor relation** $\rho_{st} \subseteq S \times S$ over the set of all program states S as follows.

$$\rho_{st} = \begin{cases} \{(\sigma,\sigma') \mid \sigma(\texttt{expr}) = true \text{ and } \sigma = \sigma'\} & \text{if } st \equiv \texttt{assume expr} \\ \{(\sigma,\sigma') \mid \sigma' = \sigma[v \mapsto \sigma(\texttt{expr})]\} & \text{if } st \equiv \texttt{v := expr} \\ \{(\sigma,\sigma') \mid \exists \sigma'' \bullet (\sigma,\sigma'') \in \rho_{st_1} \text{ and } (\sigma'',\sigma') \in \rho_{st_2}\} & \text{if } st \equiv st_1; st_2 \\ \{(\sigma,\sigma') \mid \exists x \in \mathcal{D} \bullet \sigma' = \sigma[v \mapsto x]\} & \text{if } st \equiv \texttt{havoc v} \end{cases}$$

A sequence of program states $\sigma_0, \sigma_1, \ldots$ is an **execution** of a trace $\tau = st_0, st_1, \ldots$ iff $(\sigma_i, \sigma_{i+1}) \in \rho_{st_i}$ for $i \in \{0, 1, \ldots\}$.

A **Phase Event Automaton** (PEA) is defined as a tuple $\mathcal{A} = (P, V, C, E, s, I, P^0)$ with

- P, V, C being sets of locations, clocks, and variables,
- $E \subseteq P \times \texttt{expr} \times 2^C \times P$ being a set of edges of the form (p, g, X, p') where p is the source location, p' is the target location, g is a Boolean expression over unprimed clock variables in C and primed and unprimed variables in V, and $X \subseteq C$ is a set of clocks to be reset,
- $s : P \rightarrow \texttt{expr}$ assigning a state invariant to each location where $s(p)$ is a Boolean expression over the variables V,
- $I : P \rightarrow \texttt{expr}$ assigning a clock invariant to each location where $I(p)$ is of the form $\bigwedge c_i \leq t_i$ with $c_i \in C$ and $t_i \in \mathbb{Q}^+$, and
- $P^0 \subseteq P$ being a set of initial locations.

A **configuration** of a PEA is a sequence of tuples (p, β, γ, t) with a location p, a valuation of the variables β, a valuation of clocks $\gamma : C \rightarrow \mathbb{R}_0^+$, and a non-zero duration t.

A **run** of a PEA $\mathcal{A}$ is a sequence of configurations $r = (p_0, \beta_0, \gamma_0, t_0), (p_1, \beta_1, \gamma_1, t_1), ...$ with $p_0 \in P^0$ such that for each configuration

- the valuation fulfils the location invariant ($\beta_i \models s(p_i)$),
- the clock valuation (uniformly increased by the duration) fulfils the location's clock invariant ($\gamma_i + t_i \models I(p_i)$),

and for every pair of consecutive configurations there is an edge $(p_i, g, X, p_{i+1}) \in E$ such that

- guard g is fulfilled by the valuations β_i for unprimed variables, by β'_{i+1} for primed variables, and by $\gamma_i + t$ for the clocks ($\beta_i, \beta'_{i+1}, \gamma_i + t \models g$), and
- the next clock valuation resets all clock in X and increments all others by t_i: $\gamma_{i+1}(c) = 0$ for $c \in X$ and $\gamma_{i+1}(c) = \gamma_i(c) + t_i$ for $c \notin X$.

A run $(p_0, \beta_0, \gamma_0, t_0), (p_1, \beta_1, \gamma_1, t_1) \ldots$ is **non-Zeno** if it is infinite and $\sum_{i=0}^{\infty} t_i = \infty$.

The **parallel composition** [11] of two PEAs $\mathcal{A}_1 \parallel \mathcal{A}_2$ is defined as $(P_1 \times P_2, V_1 \cup V_2, C_1 \cup C_2, E, s_1 \wedge s_2, I_1 \wedge I_2, P_1^0 \times P_2^0)$ with $((p_1, p_2), g_1 \wedge g_2, X_1 \cup X_2, (p'_1, p'_2)) \in E$ for every pair of edges in E_1 and E_2. Note that in the parallel composition of a set of PEAs, a transition can only be made if all automata can make a transition.

IV. Translation of PEAs to Boogie

Our analysis of the requirements is based on a translation of the requirements into a program encoding the requirements. In this paper we assume that the requirements are already given in the form of phase event automata (PEAs). An algorithm that translates requirements from pattern language into PEAs was given by Post et al. [18]. In this section we present the translation of the set of PEAs into a program that encodes the parallel composition.

The construction of a program encoding the parallel composition of PEAs exploits the power of using a programming language that supports real-valued variables and non-deterministic assignments. Then, (1) real-time can be encoded

by a real-valued variable that represents the current time and is non-deterministically chosen, and (2) the parallel composition can be encoded by first guessing a new state and then sequentially checking for each automaton that it has a transition to this new configuration.

The overall encoding of the parallel composition is realised by a loop where each iteration corresponds to one configuration and subsequent transition. In the beginning of the loop, all clocks are incremented by a non-deterministically chosen duration. Then, for each automaton, the invariant of the current location is checked. A variable valuation for the next configuration is guessed by non-deterministically assigning values to primed variables. For each automaton, it is checked that there is a transition that leads from the current unprimed to the chosen primed variables. Finally the new values of the primed variables are copied to the unprimed variables. Thus the translation of PEAs to a program keeps track of two configurations at once and verifies that they are related by an enabled edge.

A. PEA to Boogie Translation

Let $\mathcal{A} = \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ be the parallel composition of the PEAs $\mathcal{A}_i = (P_i, V_i, C_i, E_i, s_i, I_i, P_i^0)$. A Boogie program $\mathcal{P}(\mathcal{A}) = (Loc, \delta, \ell_0)$ encoding the parallel composition $\mathcal{A}$ is defined as follows. The variable set of the program statements is defined as $Var = V_{\mathcal{P}} \cup V'_{\mathcal{P}} \cup C_{\mathcal{P}} \cup P_{\mathcal{P}} \cup \{\texttt{delta}\}$ with: $V'_{\mathcal{P}} = \bigcup V'_i$ being the set of primed (accordingly typed) variables, $V_{\mathcal{P}} = \bigcup V_i$ being the set of un-primed (accordingly typed) variables, $C_{\mathcal{P}} = \bigcup C_i$ being the set of real-typed clock variables, $P_{\mathcal{P}} = \{\texttt{pc}_1, ..., \texttt{pc}_n\}$ being a set of auxiliary variables representing the current location of each PEA in $\mathcal{A}$, and `delta` being a real-valued auxiliary variable that represents the duration t_i of the current configuration.

We illustrate our encoding with our running example, the requirements set $\{req_3, req_4\}$. The encoding of the requirements set is shown in Figures 4 to 6. Figure 4 shows the initialization, the main loop, the various locations that represent requirement checks, and references Figure 5 for the encoding of state invariants as well as Figure 6 for the edge relation of the parallel composition.

Initially we restrict the possible locations of each PEA to its initial locations and set all clock variables to zero. The program has one edge with ℓ_0 as its source location,

$$(\ell_0, \; st_{P^0}; st_C \;, \ell_{loop})$$

with st_{P^0} being the concatenation of the assume statements $\texttt{assume} \bigvee_{p \in P_i^0} \texttt{pc}_i = p$ restricting the value of each $\texttt{pc}_i$ to correspond to an element in P_i^0, and st_C being the concatenation of the statements $c := 0$ for each $c \in C_{\mathcal{P}}$. This initialisation edge is executed only once in the beginning of the program. Note that the initialisation edge does not check if the invariant of the initial location is satisfied. This is checked at the beginning of each execution of the main loop (cf. Fig. 5).

The remaining edges of $\mathcal{P}(\mathcal{A})$ form the main loop that is executed in correspondence to each transition between two configurations of the set of PEAs. The loop begins with an edge non-deterministically assigning a positive value to the variable `delta` that represents the duration the automata stay in the current location. This value corresponds to t of a PEA configuration (p, β, γ, t). The clock variables are updated by adding the newly chosen duration to compute $\gamma + t$. The edge has the following form:

$$(\ell_{loop}, \; \texttt{havoc delta; assume delta} > 0\texttt{;} \; st_u \;, \ell_{inv}^0)$$

with st_u being the concatenation of $\texttt{c}_i \texttt{ := } \texttt{c}_i \texttt{ + delta}$ for all $\texttt{c}_i \in C_{\mathcal{P}}$ updating all clock variables.

The edges connecting locations $\ell_{inv}^0, \ldots, \ell_{inv}^n$ sequentially check for each PEA $\mathcal{A}_i$ the state and clock invariants. For each PEA $\mathcal{A}_i$ and location $p \in P_i$ there is an edge originating in location ℓ_{inv}^{i-1} that checks that the location invariant and the clock invariant are fulfilled. It has the following form:

$$(\ell_{inv}^{i-1}, \; \texttt{assume pc}_i \texttt{ = } p\texttt{; assume } st_i(p) \wedge I_i(p) \;, \ell_{inv}^i)$$

A configuration extends a previous run if it fulfils all invariants and thus program location ℓ_{inv}^n is reached. A valuation for the successor configuration β_{i+1} is guessed by non-deterministically assigning new values to the primed variables:

$$(\ell_{inv}^n, \; st_{guess} \;, \ell_{step}^0)$$

where st_{guess} is the concatenation of $\texttt{havoc x'}$, $x' \in V'_{\mathcal{P}}$.

Figure 6 shows how we encode the edge relation of the parallel composition (locations $\ell_{step}^0, \ldots, \ell_{step}^n$). This encoding ensures that for each automaton there is an edge leading to a successor location such that the guard of the edge is fulfilled. The variable $\texttt{pc}_i$ and the clock variables of the corresponding automaton are updated accordingly. For each PEA $\mathcal{A}_i$ and each location $p \in P_i$ there is an instance of the edge

$$(\ell_{step}^{i-1}, \; \texttt{assume pc}_i \texttt{ == } p \;, \ell_{step}^{i,p})$$

and every edge $(p, g, X, p') \in P_i$ is encoded by an instance of the following edge in the program

$$(\ell_{step}^{i,p}, \; \texttt{assume } g\texttt{; } st_{reset}(X)\texttt{; pc}_i \texttt{ := } p' \;, \ell_{step}^i)$$

with $st_{reset}(X)$ being the concatenation of the assignments $c := 0$ for each $c \in X$.

Finally, the variable valuation of the successor configuration is copied into the current variable valuation and the main loop is closed by returning to ℓ_{loop} (cf. Figure 4):

$$(\ell_{step}^n, \; st_{copy} \;, \ell_{loop})$$

with st_{copy} being the concatenation of $v := v'$, $\texttt{v} \in V_{\mathcal{P}}$.

This construction avoids the overhead of constructing the explicit parallel composition $\mathcal{A}$. Nonetheless, the resulting program $\mathcal{P}(\mathcal{A})$ encodes $\mathcal{A} = (P, V, C, E, s, I, P^0)$ with $P = P_1 \times \cdots \times P_n$, i. e., there is an equivalence relation between runs of the PEAs and executions of the program. This is formalised by the following theorem.

Theorem 1: For each run $r = (p_0, \beta_0, \gamma_0, t_0), \ldots$ of PEA $\mathcal{A}$ there is an execution $\sigma_0, \sigma_1, \ldots$ of the program $\mathcal{P}(\mathcal{A})$ and vice versa, such that for each configuration $(p_i, \beta_i, \gamma_i, t_i)$ and

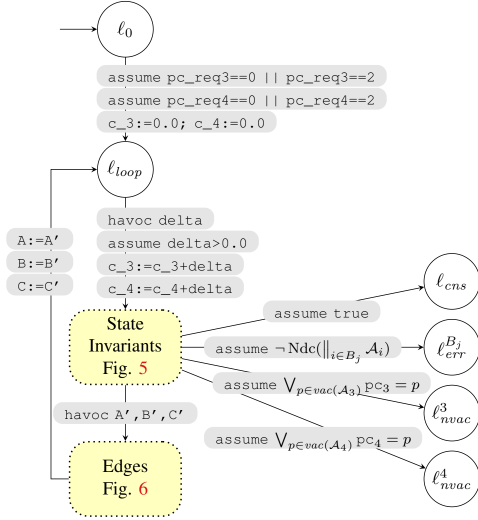


Fig. 4. Program $\mathcal{P}(\mathcal{A})$ of the parallel composition $\mathcal{A} = \mathcal{A}_{req_3} \parallel \mathcal{A}_{req_4}$.

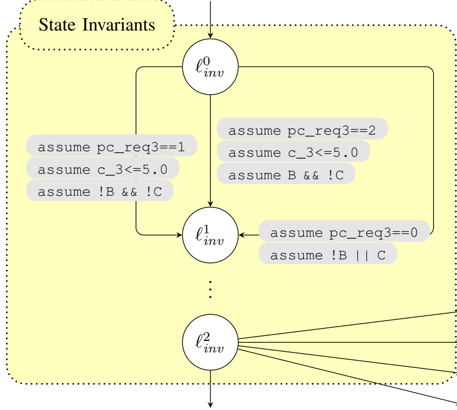


Fig. 5. The encoding of state invariants in $\mathcal{P}(\mathcal{A})$.

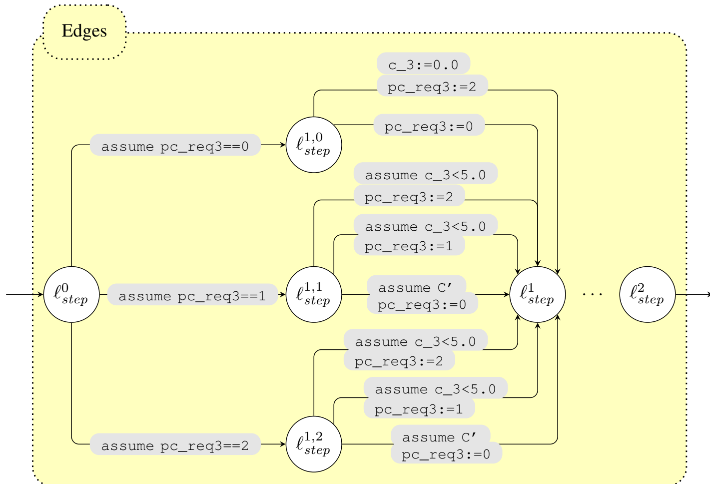


Fig. 6. The encoding of the set of edges of $\mathcal{A}$ in $\mathcal{P}(\mathcal{A})$.

the corresponding valuation $\sigma_{i'}$ where ℓ_{inv}^n is reached the i-th time (i' is $(3n+3)i + n + 2$) the following holds:

$$\begin{aligned} p_i &= (\sigma_{i'}(\mathtt{pc}_1), \dots, \sigma_{i'}(\mathtt{pc}_n)) \\ \beta_i(v) &= \sigma_{i'}(v) \text{ for all } v \in V \\ \gamma_i(c) + t_i &= \sigma_{i'}(c) \text{ for all } c \in C \\ t_i &= \sigma_{i'}(\mathtt{delta}) \end{aligned}$$

V. RT-Inconsistency Check

A set of requirements is rt-inconsistent if it is possible to reach a configuration which will inevitably be followed by the violation of a requirement in the future.

By definition of Post [18] a PEA is **rt-inconsistent** iff it has a finite run, that is not a prefix of a non-Zeno run. Post et al. proved that rt-inconsistence can be expressed equivalently as the reachability a deadlock, i. e., there is a finite run of that cannot be extended by any configuration $(p', \beta', \gamma', t')$. For each configuration in the PEA we can effectively construct a formula that expresses the existence of a successor configuration. A configuration is a valid successor, if there is an edge in each component automaton from the corresponding component, such that the guard g of each edge is satisfied and the state and clock invariant of the destination location of the edge is satisfied. By the definition of PEA, there must be a non-zero time spent in the new location. Thus, the clock invariant has to hold strictly on entry. The following definition gives the formula for the non-deadlock condition, i. e., that the current location has a successor location. The problem of detecting a deadlock thus is reduced to the reachability of a configuration in which the non-deadlock condition is violated.

Definition 1 (Non-Deadlock-Condition): Let $\mathcal{P}(\mathcal{A})$ be the program encoding the parallel composition $\mathcal{A} = \mathcal{A}_1 \parallel \cdots \parallel \mathcal{A}_n$ where $\mathcal{A}_i = (P_i, V_i, C_i, E_i, s_i, I_i, P_i^0)$, the non-deadlock-condition for $\mathcal{P}(\mathcal{A})$ is defined as:

$$\mathrm{Ndc}(\mathcal{A}) = \bigwedge_{(p_1,\dots,p_n)\in P_1\times\cdots\times P_n} \mathtt{pc}_1 = p_1 \wedge \cdots \wedge \mathtt{pc}_n = p_n \\ \to \exists \vec{v'}.\ \mathrm{Ndc}(\mathcal{A}_1, p_1) \wedge \cdots \wedge \mathrm{Ndc}(\mathcal{A}_n, p_n)$$

with

$$\mathrm{Ndc}(\mathcal{A}_i, p) = \bigvee_{(p,g,X,p')\in E_i} g \wedge s'(p') \wedge \mathrm{strict}(I(p'))[X/0]$$

The non-deadlock condition $\mathrm{Ndc}(\mathcal{A})$ is a formula over the variables V encoding the current variable valuation β, V' encoding the variable valuation of the successor configuration β', clock variables C representing the clocks updated by the duration of the current phase, and the program counter variables $\mathtt{pc}_i$ encoding the current program location.

The formula states for the i-th automaton that there is an edge $(p, g, X, p') \in E_i$ from location p to some successor location p', such that the guard of the edge is satisfied and the state and clock invariant hold for the next configuration. Here $s'(p')$ stands for the state invariant $s(p')$ where all variables are replaced by their primed counterpart, which encodes the fact that $s(p')$ should hold in the successor configuration. The

clock invariant $I(p')$ of the successor state must hold strictly, which is expressed by $\mathrm{strict}(I(p'))$. This formula is obtained from $I(p')$ by replacing all occurrences of $c \le t$ by $c < t$. It guarantees that there is some time left to stay in the successor location p'_i after it has been entered, which is required by our definition of runs. Finally by $\mathrm{strict}(I(p'))[X/0]$ we denote the formula where all variables in the reset set X are replaced by the constant 0, in other words, where the constraints $c < t$ are replaced by true for $c \in X$.

The formula $\mathrm{Ndc}(\mathcal{A})$ encodes the non-deadlock condition of $\mathcal{A}$. For each location $(p_1, \ldots, p_n)$ in the parallel composition $\mathcal{A}$, there is a conjunct in $\mathrm{Ndc}(\mathcal{A})$ in Definition 1 that encodes the existence of a successor location in case the current location is $(p_1, \ldots, p_n)$. Here we existentially quantify over the primed variables in the formula $\mathrm{Ndc}(\mathcal{A}_1, p_1) \wedge \cdots \wedge \mathrm{Ndc}(\mathcal{A}_n, p_n)$. Often all existential quantifiers can be eliminated to simplify the non-deadlock condition.

A. Non-Deadlock-Condition example

We return to our example from Figure 1. The formulas $\mathrm{Ndc}(\mathcal{A}_i, p)$ for each component automaton $\mathcal{A}_i$ and each of its location p can be simplified as follows

$$\begin{aligned}
&\mathrm{Ndc}(\mathcal{A}_{req_3}, 0) = \mathit{true}\\
&\mathrm{Ndc}(\mathcal{A}_{req_3}, 1) = \mathtt{C}' \vee c_3 < 5\\
&\mathrm{Ndc}(\mathcal{A}_{req_3}, 2) = \mathtt{C}' \vee c_3 < 5\\
&\mathrm{Ndc}(\mathcal{A}_{req_4}, 0) = \neg\mathtt{A}' \vee (\mathtt{A}' \wedge \neg\mathtt{C}')\\
&\mathrm{Ndc}(\mathcal{A}_{req_4}, 1) = \neg\mathtt{C}' \vee (\neg\mathtt{A}' \wedge c_4 \ge 2)\\
&\mathrm{Ndc}(\mathcal{A}_{req_4}, 2) = \neg\mathtt{C}'
\end{aligned}$$

For example p_2 in $\mathcal{A}_{req_3}$ has three outgoing edges, one to p_0 that can be taken if $\mathtt{C}' \wedge (\neg\mathtt{B}' \vee \mathtt{C}')$ holds, one to p_1 if $\neg\mathtt{B}' \wedge \neg\mathtt{C}' \wedge c_3 < 5$ holds, and one to itself if $\mathtt{B}' \wedge \neg\mathtt{C}' \wedge c_3 < 5$ holds. The disjunction simplifies to $\mathtt{C}' \vee c_3 < 5$. The non-deadlock condition $\mathrm{Ndc}(\mathcal{A}_{req_3} \| \mathcal{A}_{req_4})$ is (after quantifier elimination):

$$\begin{aligned}
&(\mathtt{pc}_{req_3} = 1 \wedge \mathtt{pc}_{req_4} = 1 \rightarrow c_3 < 5 \vee c_4 \ge 2)\\
\wedge\, &(\mathtt{pc}_{req_3} = 2 \wedge \mathtt{pc}_{req_4} = 1 \rightarrow c_3 < 5 \vee c_4 \ge 2)\\
\wedge\, &(\mathtt{pc}_{req_3} = 1 \wedge \mathtt{pc}_{req_4} = 2 \rightarrow c_3 < 5)\\
\wedge\, &(\mathtt{pc}_{req_3} = 2 \wedge \mathtt{pc}_{req_4} = 2 \rightarrow c_3 < 5)
\end{aligned}$$

For example, for location $\mathtt{pc}_{req_3} = 2$ and $\mathtt{pc}_{req_4} = 1$, the existentially quantified formula $\exists \mathtt{A}', \mathtt{B}', \mathtt{C}'.\ (\mathtt{C}' \vee c_3 < 5) \wedge (\mathtt{C}' \vee (\neg\mathtt{A}' \wedge c_4 \ge 2))$ can be simplified to $c_3 < 5 \vee c_4 \ge 2$. Thus, a run reaching the configuration $((2, 1), \beta, \{c_3 \mapsto 4, c_4 \mapsto 0\}, 1)$ is a witness for the rt-inconsistency of req_3 and req_4 from Section II. This is, because the non-deadlock condition does not hold for $\gamma + t$, which is $\{c_3 \mapsto 5, c_4 \mapsto 1\}$.

Given the non-deadlock condition, we can easily transform the problem of detecting deadlocks in $\mathcal{A}$ to a reachability problem of $\mathcal{P}(\mathcal{A})$. For this, we add a new error location ℓ_{err} to *Loc* and an edge $(\ell^n_{inv}, \texttt{assume}\ \neg\mathrm{Ndc}(\mathcal{A}), \ell_{err})$.

Theorem 2: A requirement set is rt-inconsistent if and only if the location ℓ_{err} is reachable in the corresponding program $\mathcal{P}(\mathcal{A})$ with the additional edge $(\ell^n_{inv}, \texttt{assume}\ \neg\mathrm{Ndc}(\mathcal{A}), \ell_{err})$.

B. c-Non-Deadlock-Condition

The formula generated by the non-deadlock condition as presented in Definition 1 is growing exponentially in the number of component automata, as every combination of locations is checked. In practice rt-inconsistencies are seldom the result of a combination of all requirements. They are frequently caused by subtle errors in the formulation of a few related requirements (like in the running example), or can be reduced to the interaction between two or three real-time requirements together with the transitive relation of their observables through invariants.

Therefore we suggest an approximation of the non-deadlock condition by the non-deadlock-conditions of all c-sized subsets of $\mathcal{A}$. We call c the **combination number**. For each subset $B \subseteq \{1, \ldots, n\}$ of size $|B| = c$, we add the edge $(\ell^n_{inv}, \texttt{assume}\ \neg\mathrm{Ndc}(\|_{i \in B} \mathcal{A}_i), \ell^B_{err})$ to the program graph. If the error location ℓ^B_{err} is reachable, the requirements in B cause an rt-inconsistency. Note that not all rt-inconsistent requirement sets are detected, because deadlocks caused by more than c requirements are not checked. The number of subsets B and the size of each Non-Deadlock-Condition grows polynomial in n and exponential in c.

The size of a non-deadlock condition depends on the number of locations in the parallel composition. Since the number of locations does not change if a component with only a single location is added, we improve our approximation by always adding all invariant automata (those that have only one location) to the set B. This optimisation increases the number of rt-inconsistent sets we can detect.

VI. Consistency and Vacuity

In the previous section we have shown how rt-inconsistency can be reduced to a reachability problem in a program. In this section we will go further and encode other well-known properties, namely *consistency* and *vacuity* [3].

A set of requirements is **consistent** iff there is at least one non-Zeno run fulfilling the requirements. To check consistency in the encoding of $\mathcal{A}$ we check if there exists at least one initial configuration for the set of PEAs by adding a test location ℓ_{cns} and inserting the following edge into $\mathcal{P}(\mathcal{A})$:

$$(\ell^n_{inv}, \texttt{assume}\ \mathit{true}, \ell_{cns})$$

If there is an execution that reaches ℓ_{cns}, there exists an initial configuration that satisfies all requirements. The existence of an initial configuration is sufficient to prove that an rt-consistent set of requirements is also consistent. Note, that although the existence of an initial configuration might look like a rather weak property, we had one case of an inconsistent set of requirements in our case study.

A requirement in a set of requirements is vacuous if it could be replaced by a more simple requirement without changing the runs of the whole set. This means that some behaviour of the requirement cannot be triggered due to other requirements.

As an example the requirement req_4 is vacuous in a set $\{\mathit{req}_1, \mathit{req}_4\}$ as $\mathtt{A}$ can never be set. In this case the requirement can never be violated and could be removed from the set.

Post et al. [19] have shown that vacuity can be expressed as a reachability problem. For example the requirement req_3 is vacuous if and only if the locations p_1 and p_2 of $\mathcal{A}_{req_3}$ are unreachable. In general one can identify a set of locations $vac(\mathcal{A}_i)$ for each requirement (for details see [19] and [11]). Intuitively this is the last phase that can be reached before the requirement is violated. In the example, this is the phase where `B` has been seen and less than 5 time units have passed without the occurrence of `C`.

The i-th requirement is **vacuous** if and only if all locations in $vac(\mathcal{A}_i)$ are unreachable. Vacuity can be checked by adding the edge $(\ell^n_{inv}, \texttt{assume} \bigvee_{p \in vac(\mathcal{A}_i)} \texttt{pc}_i = p, \ell^i_{nvac})$ to the program graph. If the location ℓ^i_{nvac} is reachable, the i-th requirement is non-vacuous in the set of requirements. Conversely if ℓ^i_{nvac} is not reachable, then requirement i is vacuous in the given requirements set.

VII. Evaluation and Application

In this section we describe the implementation of our algorithm in the tool Ultimate ReqAnalyzer. We report on the results of a comparison with the method [18] (as implemented in the tool req2ta2UPPAAL) on the small to medium sized requirements sets from [18] and we show results from an ongoing case-study in an automotive project at Bosch with an order of magnitude larger sets of requirements.

The information shown in Table I and II can be reproduced using the artifact [14] that includes the tools and (anonymized) benchmarks used here together with a control script.

A. Implementation

In order to apply our technique, we implemented the translation of formalised requirements to programs (cf. Section IV) and the subsequent analysis of this program as the open-source tool Ultimate ReqAnalyzer[2]. We use the program analysis framework Ultimate, as it already provides multiple program verification tools (e.g., Ultimate Automizer [10]) which support parsing and analysing Boogie programs. It also provides support for the interactions with SMT solvers, e.g., to simplify or even omit program edges to the error location.

The general architecture of ReqAnalyzer consists of two new modules, `Req2Pea` and `Pea2Boogie`, and an interface to Ultimate Automizer. Module `Req2Pea` expects a set of requirements in the form of a `.req` file, consisting of a declaration of typed observables followed by a list of requirement patterns as described in Section II. The module performs syntax- and type-checking and can thereby already uncover various simple issues with requirements. Afterwards, it converts each requirement into a phase event automaton and and compares the PEAs pairwise for structural equality. The occurrence of structurally equal PEAs indicates requirements duplication. If the input is well-formed, well-typed, and free of duplicates, the set of PEAs is passed to the translation module.

The translation module `Pea2Boogie` generates the Boogie program representing the parallel composition of the PEAs (cf. Section IV) and the program edges to the error location labelled with, e.g., non-deadlock conditions (cf. Section V). We also perform an important optimization in this module that decides whether a check is already *locally* satisfied, i.e., whether it already simplifies to $true$. To this end, we first simplify $\mathrm{Ndc}(\cdot)$ by inferring the implication relation for each sub-tree of the formula [8]) and then removing all covered sub-trees. Next, we try to eliminate the existentially quantified sub-formulas one by one. The majority (approx. 90%) of Ndc-checks can be simplified to $true$. In this case, no corresponding edge to the error location is added to the program. In the last step, the resulting Boogie program is transferred to the default toolchain of Automizer, which then decides whether the error locations are reachable or not.

B. Comparison with *req2ta2*UPPAAL

For our comparison, we applied both of the tools to a benchmark set using the benchmarking tool benchexec [4]. We conducted our comparison on a machine with an AMD EPYC 7351P 16-Core CPU with 2.4GHz and 128GB RAM running Linux 4.20.1 and Java 1.8.0_202 64bit. Each tool was given 900s of CPU time, 2 CPU cores and 8GB of memory to analyse each benchmark. We also set the combination number for the rt-inconsistency checks to 2 and specified a timeout per non-trivial rt-inconsistency check of 300 seconds. For ReqAnalyzer we used Ultimate 0.1.24-c551399 and for req2ta2UPPAAL we used 64bit-UPPAAL in version 4.1.22.

The results of our experiment are shown in Table I. The table is separated in four parts. In the first and second part we used benchmarks from [18] and [19], which stem from examples from the automotive domain. Unfortunately, not all benchmarks from these sources were still available, so we had to limit ourselves to a subset. The third part contains benchmarks obtained during regression testing of our tool and various handcrafted examples. The last part consists of only one example ("all") that was obtained by taking the union of all requirement sets, removing duplicates, and removing simple invariants ("Always", "Never"). For this benchmark, we increased the timeout to 9000s and the memory limit to 32GB because we wanted to see whether req2ta2UPPAAL would generate an answer or not.

Column "R." and "RT. R." show the total number of requirements and the number of real-time requirements of the benchmark, respectively. The column "Vac." shows how many vacuous requirements ReqAnalyzer found, "rt-inc." shows how many (ReqAnalyzer) or if any (req2ta2UPPAAL) rt-inconsistencies have been found, and "T. (s)" shows the needed runtime in seconds. The shorthand "TO" in various columns stands for timeout, while "OOM" stands for out of memory.

We can see from this table that our approach always produces results and never runs into a timeout, although it is sometimes slower. This is mainly due to the fact that we try to find all rt-inconsistencies and report them instead of just giving a yes/no answer. In contrast to req2ta2UPPAAL, we can report results for each of the examples, in particular for the concatenation of requirements; even with 9000s timeout

[2] https://ultimate-pa.github.io/hanfor

TABLE I
COMPARISON BETWEEN ULTIMATE REQANALYZER AND REQ2TA2UPPAAL USING BENCHMARKS FROM [18], [19].

ID	R.	RT. R.	ULTIMATE REQANALYZER Vac.	rt-inc.	T. (s)	req2ta2UPPAAL rt-inc.	T. (s)
2 [18]	10	10	no	5	15.41	yes	156.51
3 [18]	10	10	no	5	22.83	yes	17.71
4 [18]	13	3	no	no	12.61	no	0.83
5 [18]	17	3	no	no	12.22	no	1.21
6 [18]	17	9	no	1	20.16	yes	94.09
3 [19]	12	12	no	no	21.50	OOM	803.13
6 [19]	18	8	no	1	17.16	yes	22.87
7 [19]	27	6	no	no	16.46	no	113.20
8 [19]	28	1	no	no	13.46	no	0.76
9 [19]	39	1	no	no	30.56	no	1.29
10 [19]	81	3	1	no	22.79	no	0.84
10' [19]	81	3	no	no	20.82	no	3.40
1	8	2	no	no	9.79	no	0.75
2	30	7	no	no	16.32	no	379.91
3	10	5	no	no	11.56	no	1.94
4	13	3	no	no	12.68	no	1.05
5	30	7	no	no	16.37	no	417.00
6	8	4	4	1	22.88	yes	0.73
7	5	4	no	no	351.64	no	0.45
8	28	1	no	no	12.96	no	0.85
9	15	14	no	no	38.87	no	812.26
10	48	28	1	89	307.51	TO	900.21
11	3	2	no	no	8.31	no	0.38
12	2	0	no	no	7.65	no	0.32
13	5	5	1	1	13.15	yes	0.36
14	24	10	no	no	24.95	TO	900.55
15	11	6	no	no	4.55	no	0.58
16	22	5	no	no	14.46	no	247.83
17	2	1	1	no	7.92	yes	0.31
all	65	33	1	no	95.65	TO	9002.19

req2ta2UPPAAL could not provide a verdict. There are two reasons for this significant improvement. Firstly, our translation does not construct the parallel product from the PEAs, but uses a much smaller representation. Secondly, our check for local satisfaction allows us to discharge most of the expensive verification problems for a fraction of the cost.

Table I shows one case where the results of both tools differ: REQANALYZER classify benchmark 17 as vacuous but not rt-inconsistent, while req2ta2UPPAAL reports it as rt-inconsistent. This is due to a misinterpretation of UPPAAL's deadlock semantics by req2ta2UPPAAL. The parallel composition of the PEAs of this example contains a non-Zeno run, even though it has only a single location and no transitions. From UPPAAL's point of view, it is trivially deadlocked. The requirements that cause this discrepancy are the following.

*Globally, it is always the case that if '`A`' holds then '`B`' holds after **at most** '`10`' time units.*
*Globally, it is **never** the case that '`A`' holds.*

C. Preliminary results from ongoing case-study with BOSCH

Table II shows analysis results and time consumption of ten sets of requirements from the automotive domain. These stem from an ongoing large-scale case study with BOSCH. Each set of requirements is supposed to specify the behaviour of one device under development. Note that the size of the specification of Device 10 is an order of magnitude larger than any example in Table I. The reported figures have been obtained on the same machine and with the same settings as in the previous section, except that we increased the number of cores per benchmark to 4, the memory limit to 100 GB and the timelimit to 72 hours. The columns are the same as in Table I except for column "TO", which states for how many rt-inconsistency checks no result could be obtained due to a timeout, and the different format of column "Time". We can see that even for modestly sized sets (e.g., "Mod. 2") our approach finds cases of rt-inconsistency and vacuity that need to be corrected. The time needed to analyse complete modules is manageable, although there is room for further improvement by, e.g., introducing techniques from regression verification to reuse results from previous analysis runs [21].

TABLE II
RESULTS OF APPLYING ULTIMATE REQANALYZER ON REQUIREMENT SETS OBTAINED FROM AN ONGOING CASE STUDY AT BOSCH.

ID	R.	RT. R.	Vac.	rt-inc.	TO	Time
Dev. 1	26	21	no	6	no	48s
Dev. 2	50	47	no	13	no	5m 40s
Dev. 3	53	12	no	no	no	1m 33s
Dev. 4	58	53	no	13	no	6m 21s
Dev. 5	68	64	no	4	1	7m 11s
Dev. 6	100	95	no	109	no	3m 29s
Dev. 7	107	80	no	38	no	5m 9s
Dev. 8	263	234	6	73	18	3h 58m 36s
Dev. 9	407	358	2	44	no	3h 58m 03s
Dev. 10	701	545	1	no	no	4h 14m 12s

Note that the figures in Table II provide the observed analysis time given sets of *formalised* requirements. From a process perspective, the effort for the formalization of natural language requirements into the ones referred to in Table II needs to be considered. The observed times needed to formalise and validate given natural language specifications corresponds to the ones reported in [17]. Hence the initial step from hundreds of informal to formal requirements requires substantial effort (in the case-study, there are ca. 5000 different observables declared, and a similar number of expressions over observables to characterise observable system behaviour (cf. Section II)). Yet this effort is considered well acceptable by BOSCH for the following reasons. First of all, the issues that are found by our process and our tool are precise and actionable. So far, nearly 70% of our findings (including questions, typos, duplications, type issues) have resulted in a fix in the requirements, while the rest resulted in changes to the formalisation. We were told that we discovered severe defects that would have led to major issues in later stages of the system development. From the ca. 310 results of the tool, 100 were classified as major issues, although the requirements had been reviewed manually several times before we started analysing them. The remaining results

were already superseded by new requirements and currently await classification by the company.

The practically more relevant aspect is the *turnaround time* once a body of requirements has been formalised. The formalisation effort observed in our case-study is insofar partly artificial as our case-study accompanied an established development using natural language specifications. Hence there is some redundant effort, and there are entry costs: all requirements to be analysed needed a formalisation, new ones and re-used ones. Since our case-study runs in parallel to the development at the company, the requirements specifications evolved. This allowed us to observe how well our approach supports changes to requirements and agile development. Interestingly, updating the formalized requirements proved to be very fast. The changes the company made while the base set was formalised could be integrated within a few days, and our current set is now comparably stable. New changes are usually formalised in mere minutes. This allows for an effective introduction of automatic regression analysis on requirement sets, and an overall acceleration of development and requirements engineering in a traditionally slow domain [1].

VIII. Related Work

From the body of research on the analysis of formalised requirements the following three approaches and tools are closest to the work presented here as they explicitly address the automatic analysis of industrial scale requirements specifications described in pattern languages that are designed to be used in industrial practice. All three approaches and tools have in common that they only support untimed or discrete-time requirements specifications in contrast to our work that addresses continuous-time requirements. As a consequence, none of the tools discussed below provides an analysis for rt-inconsistency of real-time requirements.

The BTC Embedded Platform tool [5] works on requirements specified using the so-called simple universal pattern [22]. The simple universal pattern basically expresses if-then relations between so-called triggers and so-called actions with an upper time bound for completion of the whole requirement. Evaluation of the trigger can be modified (initial, first occurrence, or reoccurring). Yet the resulting requirements specification language of BTC Embedded Platform lacks the expressiveness of the PEAs underlying our work or the pattern language [17] that is used as input to our tool. In addition, the BTC Embedded Platform tool allows the engineer to specify the behaviour of an environment relative to which the requirements are to be considered. The tool is based on a definition of the quantitative measure *basic consistency* that combines consistency, non-vacuity, and requirements coverage. Whether absence of inconsistency or vacuity is checked by the tool depends on a user-selected coverage criterion. The BTC Embedded Platform tool particularly emphasises test case generation wrt. the specified environment. Test case generation is not in the scope of the work presented here.

The SpeAR tool [9] uses a requirements language rooted in linear temporal logic (LTL). In addition to some type-based analyses, SpeAR is able to check whether an inconsistency is reachable within a given upper bound of (discrete) computation steps. Our work, in contrast, considers continuous time. Using the bounded analysis, the SpeAR approach promises to yield partial results (it can prove that up to the given bound there is no inconsistency) and early inconsistencies in the requirements can be addressed by the requirements engineers, yet we expect the results to be unsatisfactory in general. Our approach can be used to obtain partial results for subsets of the given requirements and negative results, i.e., that there is no rt-inconsistency for any number of computation steps. The property checked by SpeAR is similar to rt-consistency in that it does not only ask for the existence of a computation satisfying all requirements (classical consistency) but searches for possible inconsistencies.

The ASSERT (Analysis of Semantic Specifications and Efficient generation of Requirements-based Tests) tool suite [7], [16] also emphasises test case generation. The specification language of ASSERT is another variant of a restricted English grammar that in particular supports system structure descriptions and a static analysis for type-consistency by providing means to declare types of system inputs and outputs. These aspects are orthogonal to the behavioural requirements properties of rt-inconsistency, vacuity, etc. considered in our work. ASSERT does not use a detailed time model but models time implicitly. Requirements may relate values at the current to the next (discrete) step and refer to time qualitatively, e.g. by the 'previous' keyword. The analysis engine uses the given requirements and ontological information captured in the requirements definition for, e.g., type checking and unit checking. Although the possibility of timed requirements is mentioned in the requirements definition language [7], real time requirements, especially rt-consistency analysis is not part of the ASSERT tool.

IX. Conclusion

We presented a scalable approach for the analysis of *rt-consistency*, *vacuity*, and *consistency* for real time requirements specifications. The ongoing cooperation with a Bosch automotive project showed that our implementation in form of the open-source tool Ultimate ReqAnalyzer scales an order of magnitude better than previous approaches and works well for industrial requirements. It delivers precise and actionable results in the form of small subsets of requirements responsible for the issues. Many of the issues we found were classified as severe defects that would have led to major issues in later stages of the system development, and thus prompted changes to the requirements.

Our approach allows many possible extensions for future work. For example, by adding additional information in form of discrimination of input, output, and internal observables, we can extend our algorithm and our tool with the ability to generate system test cases from requirements specifications and with a completeness analysis that detects missing treatment of input observable ranges.

REFERENCES

[1] S. M. Ågren, E. Knauss, R. Heldal, P. Pelliccione, G. Malmqvist, et al. The manager perspective on requirements impact on automotive systems development speed. In *RE*, pages 17–28. IEEE, 2018.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[3] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *FMSD*, 18(2):141–163, 2001.

[4] D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: requirements and solutions. *STTT*, 21(1):1–29, 2019.

[5] T. Bienmüller, T. Teige, A. Eggers, and M. Stasch. Modeling requirements for quantitative consistency analysis and automatic test case generation. In *FM&MDD*, 2016.

[6] Z. Chaochen and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Monographs in Theoretical Computer Science. Springer, 2004. An EATCS Series.

[7] A. W. Crapo, A. Moitra, C. McMillan, and D. Russell. Requirements capture and analysis in ASSERT. In *RE*, pages 283–291. IEEE, 2017.

[8] I. Dillig, T. Dillig, and A. Aiken. Small formulas for large programs: Online constraint simplification in scalable static analysis. In *SAS*, volume 6337 of *LNCS*, pages 236–252. Springer, 2010.

[9] A. W. Fifarek, L. G. Wagner, J. A. Hoffman, B. D. Rodes, M. A. Aiello, et al. Spear v2.0: Formalized past LTL specification and analysis of requirements. In *NFM*, volume 10227 of *LNCS*, pages 420–426, 2017.

[10] M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, et al. Ultimate automizer and the search for perfect interpolants - (competition contribution). In *TACAS (2)*, volume 10806 of *LNCS*, pages 447–451. Springer, 2018.

[11] J. Hoenicke. *Combination of processes, data, and time*. PhD thesis, Carl von Ossietzky University of Oldenburg, 2006.

[12] *IEEE Recommended Practice for Software Requirements Specifications*, 1998. Std 830-1998.

[13] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *ICSE*, pages 372–381. ACM, 2005.

[14] V. Langenfeld, D. Dietsch, B. Westphal, J. Hoenicke, and A. Post. Scalable Analysis of Real-Time Requirements (Artifact). https://doi.org/10.5281/zenodo.3341453, June 2019.

[15] K. R. M. Leino. This is Boogie 2. *Manuscript KRML*, 178(131), 2008.

[16] A. Moitra, K. Siu, A. W. Crapo, H. R. Chamarthi, M. Durling, M. Li, H. Yu, P. Manolios, and M. Meiners. Towards development of complete and conflict-free requirements. In *RE*, pages 286–296. IEEE, 2018.

[17] A. Post and J. Hoenicke. Formalization and analysis of real-time requirements: A feasibility study at BOSCH. In *VSTTE*, volume 7152 of *LNCS*, pages 225–240. Springer, 2012.

[18] A. Post, J. Hoenicke, and A. Podelski. rt-inconsistency: A new property for real-time requirements. In *FASE*, volume 6603 of *LNCS*, pages 34–49. Springer, 2011.

[19] A. Post, J. Hoenicke, and A. Podelski. Vacuous real-time requirements. In *RE*, pages 153–162. IEEE, 2011.

[20] A. Post, I. Menzel, and A. Podelski. Applying restricted english grammar on automotive requirements — does it work? In *REFSQ*, pages 166—180, 2011.

[21] B. Rothenberg, D. Dietsch, and M. Heizmann. Incremental verification using trace abstraction. In *SAS*, volume 11002 of *LNCS*, pages 364–382. Springer, 2018.

[22] T. Teige, T. Bienmüller, and H. J. Holberg. Universal pattern: Formalization, testing, coverage, verification, and test case generation for safety-critical requirements. In *MBMV*, pages 6–9. Albert-Ludwigs-Universität Freiburg, 2016.