Non-vacuous Real-time Requirements

Amalinda Post
Research and Advance Engineering
Robert Bosch GmbH
Stuttgart, Germany
Amalinda.Post@de.bosch.com

Jochen Hoenicke
Software Engineering
University of Freiburg
Freiburg, Germany
hoenicke@informatik.uni-freiburg.de

Andreas Podelski
Software Engineering
University of Freiburg
Freiburg, Germany
podelski@informatik.uni-freiburg.de

***Abstract*—We introduce the property of vacuity for requirements. A requirement is vacuous in a set of requirements if it is equivalent to a simpler requirement in the context of the other requirements. For example, the requirement "if A then B" is vacuous together with the requirement "not A". The existence of a vacuous requirement is likely to indicate an error. We give an algorithm that proves the absence of this kind of error for real-time requirements. A case study in an industrial context demonstrates the practical potential of the algorithm.**

***Keywords*-behavioral requirements; real-time requirements; automotive; validation; verification; tool; case study;**

I. Introduction

In this paper we propose a new property to ensure the quality of a set of real-time requirements. We denote that property *vacuity*. The property reflects that we want to ensure that there exists a system such that every requirement is *non-vacuously* satisfied in the system.

The notion of customary consistency (there exists a system that satisfies the requirements [1]) is too weak to ensure this. For certain sets of requirements, every system satisfying all requirements satisfies at least one requirement only *vacuously*. For example, consider the following two requirements:

- Req_1: If *ButtonPressed* holds for more than 5 seconds then it is never the case that *AssistFunction* stays turned off for more than 10 seconds.
- Req_2: It is never the case that *ButtonPressed* holds for more than 3 seconds.

The set of the two requirements is consistent (one can find systems that satisfy both requirements). However, a closer inspection of the requirements shows that the requirements are in conflict. Because of Req_2 the precondition of Req_1 never holds, i. e., the postcondition of Req_1 does not have to come to pass. Thus, in any system satisfying both requirements Req_1 is only vacuously satisfied. The set of requirements, while consistent, is *vacuous*!

One way to resolve the partial inconsistency is to delete Req_2 or to change it to the *weaker* requirement Req_2'.

- Req_2': Before $Startup$ holds, it is never the case that *ButtonPressed* holds for more than 3 seconds.

This way, after $Startup$, *ButtonPressed* may hold for more than 5 seconds, thus the precondition of Req_1 (and therefore also its postcondition) may hold in this time slot.

We think that vacuities frequently occur in the process of requirements elicitation. Most of these vacuities are resolved in manual reviews. However, one may never be sure that all vacuities are resolved without an automatic check. Thus, in this paper, we propose a definition for vacuity of requirements and an algorithm to automatically check a set of requirements for vacuity. We evaluate the practical potential of the algorithm in a case study over more than 200 requirements taken from 10 components of automotive Bosch projects. Our case study shows that such a check is beneficent. Although one component was not tractable by the current implementation, we could ensure that 8 components had no vacuity and we discovered one vacuity.

Note that the concept of vacuity is related to the work of [2], [3], [4]. In our example, the two requirements are vacuous as in the context of Req_2 the precondition of Req_1 is never satisfied, i. e., Req_1 is only trivially valid. The problem of a trivially valid formula was first noted by Beatty and Bryant [2], who termed it *antecedent failure*. Antecedent failure means that a formula is trivially valid because the precondition (antecedent) of the formula is not satisfiable in a given model. This idea was then further developed by [3], [4] and renamed to *vacuous satisfiability*. However the concept of antecedent failure and vacuous satisfiability was always used with respect to model checking. They check whether a *given* implementation satisfies the requirements in a vacuous way. In contrast we work solely on the requirements set: we investigate whether a set of requirements is vacuous in the sense that in *any* implementation satisfying the requirements at least one requirement is only vacuously satisfied. Another difference is that the earlier work bases on qualitative temporal logics LTL/ACTL, whereas we define vacuity first independently of any logic and then for a Duration Calculus fragment.

II. Defining vacuity

First, we define *vacuity* in an abstract way, i. e., independently of any logic. After that we instantiate the definition for requirements formalized in Duration Calculus.

A. Abstract definition of vacuity

To define vacuity we need to define a relation *simpler* between requirements. The definition may be purely syntactical, e. g., for a requirement formalized in a formula φ one might define that a requirement $\widetilde{\varphi}$ is simpler than φ if it is a subformula of φ. We write $\widetilde{\varphi} \prec \varphi$ to denote that $\widetilde{\varphi}$ is simpler than φ.

Consider the requirement Req_1 of Section I, which can be restated as

- Req_1: It is never the case that *ButtonPressed* holds for more than 5 seconds and then *AssistFunction* stays turned off for more than 10 seconds.

One might define "simpler" in such a way such that the following requirements are all simpler than Req_1:

- Req_1': It is never the case that *ButtonPressed* holds for more than 5 seconds and then *AssistFunction* stays turned off.
- Req_1'': It is never the case that *ButtonPressed* holds for more than 5 seconds.
- Req_1''': It is never the case that *ButtonPressed* holds.

In the following, we assume that there is a given definition of "simpler" requirements. Note that we only require a syntactical definition. Thus $\widetilde{\varphi} \prec \varphi$ does **not** imply that the requirement $\widetilde{\varphi}$ is *strictly stronger* than φ in a semantical way, i.e., $(\widetilde{\varphi} \prec \varphi) \not\Rightarrow (\widetilde{\varphi} \Rightarrow \varphi \wedge \varphi \not\Rightarrow \widetilde{\varphi})$.

Consider a set of requirements Φ (i. e., a conjunction of requirements $\Phi = \bigwedge_{i=1}^{n} \varphi_i$) and a separate requirement φ. Say there is a requirement $\widetilde{\varphi}$ that is simpler than the requirement φ and *in the context of the set of requirements* Φ both requirements φ and $\widetilde{\varphi}$ are equivalent. This means that in the context of Φ both requirements are exchangeable. Then the question arises, why the requirements engineer did not choose the simplest requirement.

Consider the example from the introduction. Req_2 states that *ButtonPressed* never holds for more than 3 seconds. Thus, Req_2 implies that the simpler requirement Req_1'' holds as well — as *ButtonPressed* does not hold for more than 3 seconds, it does not hold for more than 5 seconds as well. Thus, in the context of Req_2 the precondition of Req_1 never holds, i.e. the second part *"and then AssistFunction stays turned off for more than 10 seconds"* is useless. We assume that a requirements engineer specifies only requirements with behavior that shall be visible in a system, thus we assume that there is an error in the requirements and call Req_1 *vacuous* with the set of requirements containing Req_2.

Definition 1 (vacuity of a requirement φ in a set Φ): A requirement φ is *vacuous in* Φ if there is a requirement $\widetilde{\varphi}$ that is simpler than φ (i. e., $\widetilde{\varphi} \prec \varphi$) and in the context of Φ the requirements φ and $\widetilde{\varphi}$ are equivalent (i. e., $\Phi \Rightarrow (\varphi \Leftrightarrow \widetilde{\varphi})$).

A set of requirements is vacuous if any of its requirements is vacuous in the set of remaining requirements.

Definition 2 (vacuity of a set of requirements Φ): A set of requirements $\Phi = \bigwedge_{j=1}^{n} \varphi_j$ is *vacuous* if there is a requirement φ_i that is vacuous in $\bigwedge_{j \neq i} \varphi_j$.

B. Instantiated definition of vacuity

A convenient way to obtain a suitable formalization of requirements is to borrow the notation of the *Duration Calculus* [5], [6], [7]. Before we introduce the formal syntax of our class of real-time requirements, we will derive the formalization of the example requirement Req_1 from Section I. We first restate Req_1 in a less ambiguous form.

- Req_1: It is never the case that *ButtonPressed* holds for more than 5 seconds (*while AssistFunction does not already hold*) and then *AssistFunction* stays turned off for more than 10 seconds.

Then, we introduce the two predicates *ButtonPressed* and *AssistFunction* (with their obvious meaning) and reformulate Req_1 as follows.

- Req_1: For any run of the system, it must not be the case that there are time points $t_1 < t_2 < t_3$, such that *ButtonPressed* is true between t_1 and t_2, *AssistFunction* is false between t_1 and t_3, the length of the interval $[t_1, t_2]$ is greater than 5 seconds, and the length of the interval $[t_2, t_3]$ is greater than 10 seconds.

Equivalently, for any run of the system, it must not be possible to split the time axis into four consecutive *phases* where:

1) the first phase (from time point 0 to t_1) does not underlie any constraint,
2) the second phase (from time point t_1 to t_2) underlies the constraint that *ButtonPressed* is true and *AssistFunction* is false and its length (the difference between t_1 and t_2) is greater than 5,
3) the third phase (from time point t_2 to t_3) underlies the constraint that *AssistFunction* is false and its length is greater than 10,
4) the fourth phase (from time point t_3 until infinity) does not underlie any constraint.

In formal syntax, the requirements Req_1 and Req_2 are expressed as the formulas φ_1, φ_2 below. Here the symbol "$\neg$" denotes negation, the symbol "$;$" separates two phases, the phase "$\lceil P \rceil$" refers to a nonzero-length period of time during which the predicate P is satisfied, adding the conjunct "$\ell > k$" to a phase means that its length is strictly greater than the constant k, and the constant phase "*true*" refers to a period of time during which the behavior does not underlie any constraint (and which is possibly of zero length).

- $\varphi_1 = \neg(true; \lceil ButtonPressed \wedge \neg AssistFunction \rceil \wedge \ell > 5; \lceil \neg AssistFunction \rceil \wedge \ell > 10\,; true)$
- $\varphi_2 = \neg(true\,; \lceil ButtonPressed \rceil \wedge \ell > 3\,; true)$

Syntax: Formally, the syntax of phases π and requirements φ is defined by the BNF below. The predicate symbol P is a propositional formula over a fixed set *Preds* of predicate symbols (for *observations* whose truth values change over time). Optionally, the duration ℓ of a phase can be bounded by a timing bound $k \in \mathbb{N}^+$. The correctness of the algorithm presented in this paper relies on the fact that we have only *strict* inequalities ($\ell > k$ or $\ell < k$). An extension to non-strict inequalities unnecessarily complicates the algorithm and the proof of correctness, without being motivated by practical examples.

requirement	$\varphi ::= \neg\,(\pi_1\ ;\ \ldots ; \pi_m\ ;\ true)$
phase	$\pi ::= true \mid \lceil P \rceil \mid true \wedge \ell \sim k \mid \lceil P \rceil \wedge \ell \sim k$
	$\sim ::= < \ \mid\ >$

We denote with Φ the conjunction of the requirements, i. e., $\Phi = \bigwedge_{i=1}^{n} \varphi_i$.

Interpretation $\mathcal{I}$: Avoiding the confusion about the different meanings of other terms in the literature, we use the term *interpretation* to refer to a mapping that assigns to each time point t on the time axis (i. e., each $t \in \mathbb{R}_{\geq 0}$) an observation, i. e., a valuation of the family of given predicates P.

$$\mathcal{I} : \mathbb{R}_{\geq 0} \to \{true,\ false\}^{Preds}\ ,\quad \mathcal{I}(t)(P) \in \{true,\ false\}$$

We use "*segment* of $\mathcal{I}$ from b to e" and write "$(\mathcal{I}, [b, e])$" for the restriction of the function $\mathcal{I}$ to the interval $[b, e]$ between the ("begin") time point b and the ("end") time point e.

Satisfaction of a requirement by an interpretation: We first define the satisfaction of a requirement by a segment of an interpretation, $(\mathcal{I}, [b, e]) \models \varphi$.

$(\mathcal{I}, [b, e]) \models \lceil P \rceil$	if $\mathcal{I}(t)(P)$ is true for almost all $t \in [b, e]$ and $b \neq e$
$(\mathcal{I}, [b, e]) \models \ell \sim k$	if $(e - b) \sim k$
$(\mathcal{I}, [b, e]) \models \pi_1\ ;\ \pi_2$	if $(\mathcal{I}, [b, m]) \models \pi_1$ and $(\mathcal{I}, [m, e]) \models \pi_2$ for some $m \in [b, e]$

We can then define the satisfaction of a requirement by a ('full') interpretation.

$$\mathcal{I} \models \varphi \qquad \text{if}\ \ (\mathcal{I}, [0, t]) \models \varphi\ \ \text{for all}\ t$$

That is, an interpretation $\mathcal{I}$ satisfies the requirement φ if every prefix of $\mathcal{I}$ does (i. e., if for every time point t, the segment of $\mathcal{I}$ from 0 to t satisfies φ).

Characterization of "simpler": We obtain a *simpler* requirement $\widetilde{\varphi}$ from a requirement φ by omitting a sequence of phases of φ beginning on the right, and optionally omitting the time bound of the new last phase.

Definition 3 (simpler requirement): Given a requirement $\varphi = \neg(\pi_1; \ldots; \pi_j; \ldots; \pi_m; true)$, a requirement $\widetilde{\varphi}$ *is simpler than* φ, denoted as $\widetilde{\varphi} \prec \varphi$, if $\widetilde{\varphi}$ is not syntactically equal to φ and

$\widetilde{\varphi} = \neg(\pi_1; \ldots; \pi_{j-1}; \widetilde{\pi}_j; true)$, where $1 \leq j \leq m$ and $\widetilde{\pi}_j = \pi_j$ or ($\widetilde{\pi}_j = \lceil P \rceil$ and $\pi_j = \lceil P \rceil \wedge \ell \sim k$).

Note that this is a purely syntactical definition, thus for a given requirement φ and a *simpler* requirement $\widetilde{\varphi}$ the simpler $\widetilde{\varphi}$ may be semantically equivalent to φ. A requirement like, e. g., $\neg(\lceil P \rceil; \lceil P \rceil; true)$ is equivalent to its simpler requirement $\neg(\lceil P \rceil; true)$. In that case the requirement is vacuous in every context. Also note that the definition implies that the simpler $\widetilde{\varphi}$ is stronger than φ, i. e., $\widetilde{\varphi} \Rightarrow \varphi$.

Vacuity: Using the simpler relation from above, vacuity is defined as in Section II-A.

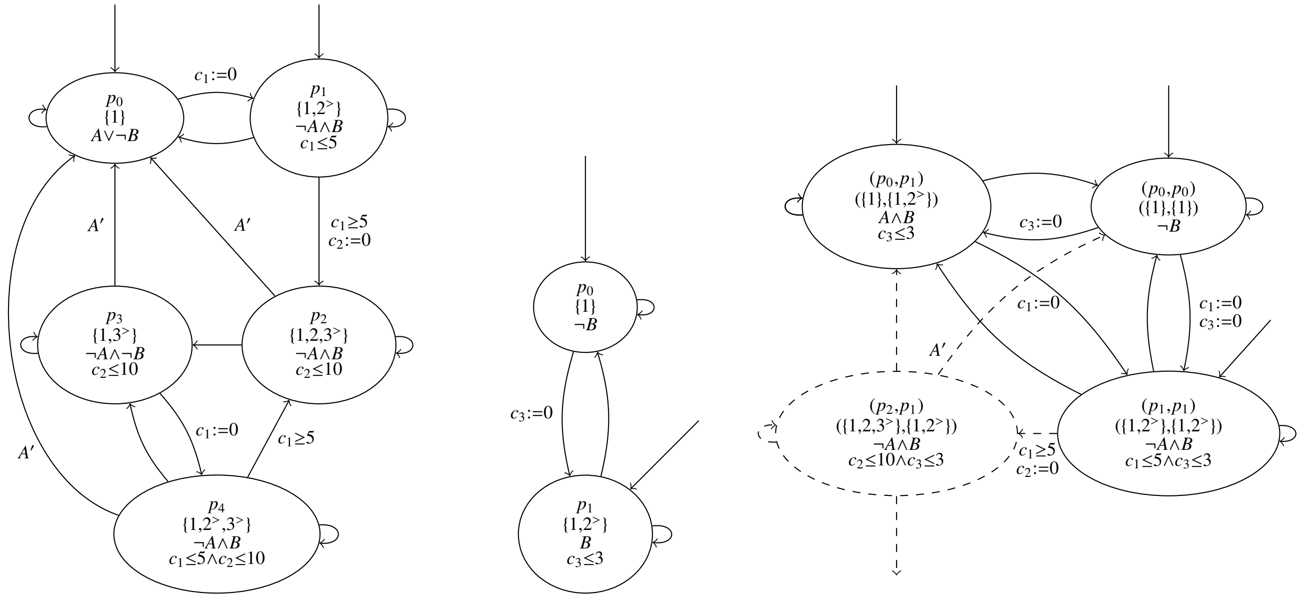
III. Checking vacuity

In this section we present an algorithm (Algorithm 1) to check whether a set of requirements Φ is vacuous. To simplify the presentation we assume that Φ is consistent. We have implemented the check for consistency, not presented here, and use it as a preliminary step in our experiments.

A set of requirements Φ is vacuous if it contains a requirement φ_i that is vacuous in $\Phi' = \bigwedge_{i \neq j} \varphi_j$, i. e., if in the context of Φ' the requirement φ_i is equivalent to a simpler requirement $\widetilde{\varphi_i} \prec \varphi_i$. The idea of the algorithm is to solve the problem on an automaton-representation. To do so we construct a certain kind of automaton, a so-called *phase event automaton* (PEA), from every φ_i in Φ.

The algorithm then exploits three properties of this construction. First, we construct for every φ_i an automaton A_i that represents φ_i, i. e., A_i has a run if and only if φ_i has a matching interpretation. Second, the parallel product of the automata represents the conjunction of the requirements, i. e., the parallel product $A = \|_{i=0}^{n} A_i$ has a run if and only if $\Phi = \bigwedge_{i=0}^{n} \varphi_i$ has a matching interpretation. Third, every location in A_i is labeled by the set of phases of φ_i that were observed when reaching this location, i. e., index j is in the set *phases* of a location if and only if the automaton (being in this location) has observed the requirement's first j phases including the j-th phase.

Figure 1 presents the intermediate results of the different steps of the application of Algorithm 1 to the set of requirements $\Phi = \varphi_1 \wedge \varphi_2$, where φ_1, φ_2 are defined in Section II. For the sake of readibility we abbreviated *ButtonPressed* as B and *AssistFunction* as A. Algorithm 1 transforms the requirements φ_1 and φ_2 into the phase event automata A_1 and A_2 in Figure 1a resp. Figure 1b. It labels every location with a set *phases*, depicting the observed phases in this location. E.g., in location p_3 of A_1 the automaton observes the phase $\pi_1 = true$ and the phase $\pi_3 = \lceil \neg AssistFunction \rceil \wedge \ell > 10$. Since π_3 has a time bound, we need to differentiate two cases: either *AssistFunction* held for more than 10 seconds or *AssistFunction* holds for at most 10 seconds (and the clock c measures the duration it held). If the time bound is not already satisfied, then we denote that with a superscript "$>$" on the corresponding phase. Thus, the phase labeling of p_3 contains the phases 1 and $3^{>}$. In the next step, the algorithm computes $maxPhase(\varphi_i)$, which denotes the phase before the last phase in φ_i, i. e., $maxPhase(\varphi_1) = 3^{>}$ and



(a) phase event automaton A_1 for the requirement $\varphi_1 = \neg(true\,;\lceil B \wedge \neg A \rceil \wedge \ell > 5\,;\lceil \neg A \rceil \wedge \ell > 10\,;\,true)$

(b) phase event automaton A_2 for the requirement $\varphi_2 = \neg(true\,;\lceil B \rceil \wedge \ell > 3\,;\,true)$

(c) phase event automaton $A = A_1 \| A_2$. The dashed part is not reachable, since in (p_1, p_1) it holds that $c_1 \leq c_3 \leq 3$.

Figure 1. Algorithm 1 applied to the set of the requirements φ_1 and φ_2 from Section II. Algorithm 1 constructs the phase event automata A_1 and A_2, forms their parallel product $A = A_1 \| A_2$, and checks for locations containing $maxphase(\varphi_1) = 3^>$ resp. $maxPhase(\varphi_2) = 2^>$. Since these locations exist the algorithm checks for both components whether such a location is reachable. For Component 1, it observes that no such location is reachable (note that location (p_2, p_1) is not reachable), thus it deduces that the set of requirements is vacuous.

$maxPhase(\varphi_2) = 2$. Here, $3^>$ and 3 are separate phases and $3^>$ comes before 3. The phase $maxPhase(\varphi_i)$ is the last phase of φ_i where φ_i is not violated.

The algorithm then forms the parallel product $A = A_1 \| A_2$ (given in Figure 1c) and checks whether the phases $maxPhase(\varphi_1)$ and $maxPhase(\varphi_2)$ occur in the labelings of locations in A. In this example A still contains locations labeled with these phases. However, a closer look shows that some of these locations are not reachable in any run. In location (p_1, p_1) the value of clock c_5 is always less than or equal to c_3. Hence, the clock c_5 can never reach a value greater than 3 and the location (p_2, p_1) and all other locations (which we omitted in the figure) with 2 or $3^>$ in the first component of the phase labeling are not reachable. This means that in the context of Φ the formula $\widetilde{\varphi}_1 \prec \varphi_1$ where phase π_3 is omitted is equivalent to φ_1. Thus, the algorithm returns that Φ is vacuous.

A. Phase Event Automata

We will use *phase event automata* as a means to define sets of interpretations $\mathcal{I}$ (i.e., mappings from time points to valuations of predicates). We base our work on the definitions by [8], [7]. Syntactically, a phase event automaton resembles a timed automaton [9] in that it has the same notion of *clocks*; semantically, there are differences such as in the minimal duration between transitions. For a set of variables V, we use V' for the set of their primed versions (which stand, as usual, for the value of the corresponding variable in a successor state after a transition). We use $\mathcal{L}(V)$ to denote the set of formulae with free variables in V.

A *phase event automaton* (PEA) is a tuple

$$A = (P, V, C, E, s, I, P^0) \text{ where}$$

- P is a set of locations p (*phases*),
- C is a set of clocks c,
- V is a set of Boolean variables (*observation predicates*),
- E is a set of *transitions* of the form (p, g, X, p') where $p, p' \in P$ specify the from- and to-locations, the guard g is a formula in the unprimed clock variables and in the unprimed and primed Boolean variables (g specifies also the updates of Boolean variables), and X is the set of clocks that are reset to 0; $E \subseteq P \times \mathcal{L}(C \cup V \cup V') \times 2^C \times P$,
- the mapping s assigns each location p its *state invariant* which is stated as a formula in the Boolean variables, i.e., $s : P \to \mathcal{L}(V)$,
- the mapping I assigns each location p its *clock invariant* which is stated as a formula in the clocks, more precisely a conjunction of inequalities $c \leq k$ or $c < k$ with $c \in C$ and $k \in \mathbb{R}_{\geq 0}$, i.e., $I : P \to \mathcal{L}(C)$,
- P^0 is the set of initial locations, i.e., $P^0 \subseteq P$.

We use *runs* to describe the operational semantics of a PEA. A run r is a (finite or infinite) sequence of quadruples (p, β, γ, t) consisting of a location p, a valuation of the Boolean variables $\beta : V \to \{true, false\}$, a valuation of the clocks $\gamma : C \to \mathbb{R}_{\geq 0}$, and a *non-zero duration* t (the amount of time spent in the location p), i.e., $t > 0$.

Given the PEA A of the form above, r is a run of A if it starts in an initial location with clock values 0, and for each quadruple (p, β, γ, t) in r, the valuation of variables β satisfies the state invariant of location p (i. e., $\beta \models s(p)$), the clock valuation γ satisfies the clock invariant at location p during the whole duration t (i. e., $\gamma + t \models I(p)$), and for each pair of consecutive quadruples (p, β, γ, t) and $(p', \beta', \gamma', t')$, the valuations satisfy the guard and the update constraint of a transition in E of the form (p, g, X, p'), i. e., $(\beta, \beta', \gamma + t) \models g$ (where β' is applied to the primed variables in g) and $\gamma'(c)$ is 0 if $c \in X$ and $\gamma(c) + t$ otherwise. The *duration* of a run r is the sum of the durations t in its quadruples. We denote the *set of all runs* of A by $Run(A)$.

Interpretations accepted by A, $\mathcal{L}(A)$: A run r *matches* an interpretation $\mathcal{I}$ if for almost all time points t, the value of $\mathcal{I}$ coincides with the valuation β in the quadruple of r that corresponds to time t, i. e., the last quadruple such that the sum of durations of all quadruples preceding it in r is smaller than t. We omit the cumbersome formal definition. For every run r of a phase event automaton A there exists an interpretation $\mathcal{I}$ such that r matches $\mathcal{I}$.

An interpretation $\mathcal{I}$ is *accepted* by A, formally $\mathcal{I} \in \mathcal{L}(A)$, if there is a run r of A that matches $\mathcal{I}$. The prefix of the interpretation $\mathcal{I}$ until the time point t is *accepted* by A, formally $(\mathcal{I}, [0, t]) \in \mathcal{L}(A)$, if there is a run r of A with duration t that matches $(\mathcal{I}, [0, t])$.

A phase event automaton A *represents* a requirement φ if it accepts exactly the interpretations that satisfy φ, i. e., $\mathcal{I} \in \mathcal{L}(A)$ if and only $\mathcal{I} \models \varphi$. Given two PEAs A_1 and A_2 representing the requirements φ_1 resp. φ_2, their parallel product $A_1 \parallel A_2$ (defined in the canonical way) represents their conjunction $\varphi_1 \wedge \varphi_2$.

Phase labeling of A: Phase event automata constructed according to the algorithm given in [8] have a phase labeling assigning to each location p a set $phases(p)$. For a PEA A_i representing a requirement φ_i, phase j is in the set $phases(p)$ if and only if the automaton (being in location p) has observed the requirement's first j phases including the j-th. If the j-th phase has an lower time bound $\ell > k$ and that time bound is not yet satisfied, $phases(p)$ contains the element $j^>$ instead. Thus, the elements in $phases(p)$ come from the set $Phases = \{1^>, 1, \ldots, m^>\}$). For a parallel product of automata $A = \|_{i=1}^{n} A_i$, the labeling *phases* assigns to a location $p \in P$ of the automaton a tuple of sets of phases, e. g., in Figure 1c $phases(p_0, p_1) = (\{1\}, \{1, 2^>\})$.

The function *maxPhase* assigns to each a requirement φ the index of the last reachable phase, i. e., for a requirement $\varphi = \neg(\pi_1, \ldots, \pi_m, true)$,

$$maxPhase(\varphi) := \begin{cases} m^> & \text{iff } \pi_m \text{ has an lower time bound} \\ m-1 & \text{otherwise} \end{cases}$$

In our running example we have $maxPhase(\varphi_1) = 3^>$ and $maxPhase(\varphi_2) = 2^>$.

B. General idea

Consider a requirement $\varphi_0 = \neg(\pi_1; \pi_2; \ldots; \pi_m; true)$. In the corresponding automaton A_0 no location is labeled with the phase m, since reaching this location would indicate that the requirement is not satisfied. Hence, the maximum phase label that may occur in A_0 is either $m^>$ (if π_m has a lower bound $\ell > k$) or $m-1$ (otherwise). Further, the automaton $\widetilde{A_0}$ representing the simpler requirement $\widetilde{\varphi_0} := \neg(\pi_1; \pi_2; \ldots; \pi_{m-1}; true)$, is a subgraph of A_0, i. e., $\widetilde{A_0}$ contains all locations of A_0 that cannot observe the $m-1$-th phase. Moreover $\widetilde{A_0}$ contains the same transitions between these locations as A_0, the same guards, invariants, etc. It only differs from A_0 in that all locations labeled with $m-1$ and all transitions to these locations are removed. Similarly, the automaton implementing a simpler requirement $\widetilde{\varphi_0}$ where only the time bound of the last phase is omitted is also a subgraph of A_0. It contains all locations of A_0 except the locations p that contain $m^>$ in the phase labeling. This property is illustrated in Figure 2.

Now suppose that in some parallel product $A_0 \parallel A$ where A represents a set of requirements Φ, no location with $maxPhase(\varphi_0)$ is reachable. Let $\widetilde{A_0}$ be the subautomaton of A_0 where all locations containing $maxPhase(\varphi_0)$ are removed. Then $\widetilde{A_0}$ is the automaton for a simpler requirement $\widetilde{\varphi_0} \prec \varphi_0$. The parallel product $\widetilde{A_0} \parallel A$ is equivalent to $A_0 \parallel A$, since it contains the same reachable locations and transitions. Hence, $\widetilde{\varphi_0}$ and φ_0 are equivalent in the context of Φ.

We use this property to check whether a requirement φ_0 is vacuous in Φ. Let A_0 and A be the automata corresponding to φ_0 and Φ. Then φ_0 is vacuous to Φ, if no location labeled with $maxPhase(\varphi_0)$ is reachable in the product automaton $A_0 \parallel A$. So we can check incongruity with a reachability analysis on a timed automaton.

C. Algorithm

In our setting we check incongruity for a set of requirements $\Phi = \bigwedge_{i=1}^{n} \varphi_i$. Each requirement is translated to an automaton A_i and $A = \|_{i=1}^{n} A_i$ is the parallel product of these automata. After building the parallel product, Algorithm 1 computes for each property φ_i the locations that contain the maximum phase of that property, $maxPhase(\varphi_i)$. If such locations exists, then it checks whether one of the locations is indeed reachable using Procedure 2. this is not the case, the property φ_i is reported as vacuous in Φ.

The procedure $Reachable(A, locs)$ is needed because the PEA-construction algorithm [8] executes no reachability-analysis itself. Although it deletes transitions with unsatisfiable guards and locations that are not reachable from a start location in the PEA interpreted as *graph*, it does *not* check whether the locations are really reachable in a *run*. However, a PEA might contain locations that are reachable in the graph, but not in a run, e.g., a PEA as depicted in Figure 1c. Thus, to make sure that a location with $maxPhase(\varphi_i)$ in the phase labeling is in fact reachable, we use Procedure 2.

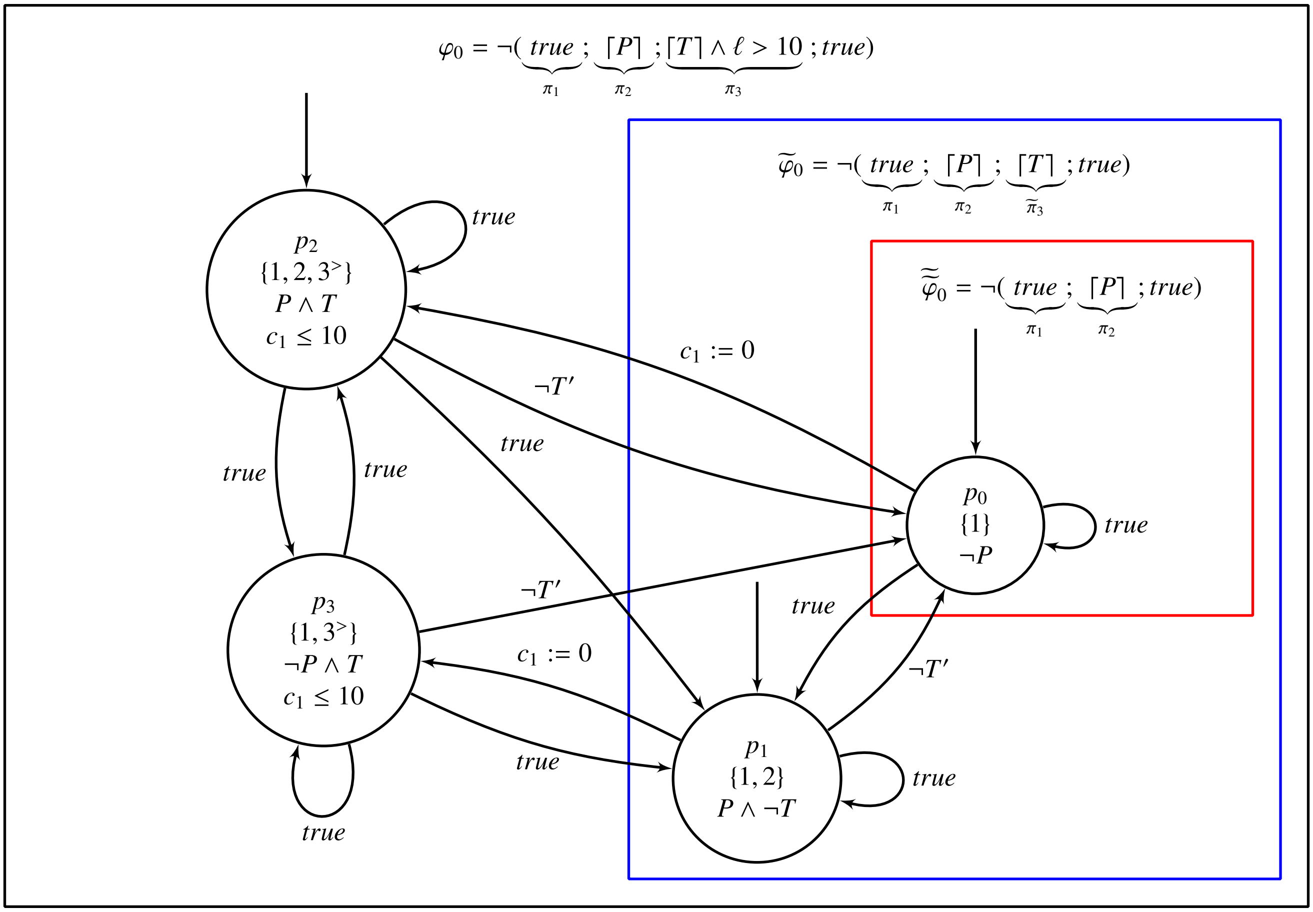


Figure 2. The corresponding automata of simpler requirements are subgraphs of the automaton representing the original requirement φ_i

IV. Proof of correctness

To prove the correctness of our algorithm, we need to show that if a requirement φ_i is vacuous in a set of requirements Φ then the parallel product $A_i \parallel A$ contains no reachable location p with $maxPhase(\varphi_i) \in phases(p)[i]$ where A_i represents φ_i and A represents Φ. To do so we first recall from [8] that (1) there is an algorithm that correctly constructs a phase event automaton representing a requirement φ and (2) the phase labeling of this algorithm is correct.

Lemma 4.1: Given a requirement φ_i the algorithm given in [8] constructs a deterministic PEA representing the requirement.

The proof is given in [8]. The idea of this algorithm is similar to the power set construction of a deterministic finite automaton from a nondeterministic one. A nondeterministic PEA that accepts a sequence $(\pi_1; \ldots; \pi_m; true)$ can be constructed by introducing a location labeled with j for each phase π_j of the requirement that ensures that the predicate of π_j holds. For a location with a lower bound another location labeled with $j^>$ is introduced before location j. This observes the timing constraint and enters the next location when the bound is satisfied. The location labeled with $j^>$ is not really necessary but simplifies the construction of the deterministic automaton. A location labeled with j is reached in a run if there is an interpretation that fits to the run and that satisfies $\pi_1; \ldots; \pi_j$. The final location is accepting and is never left.

The PEA for the requirement $\varphi = \neg(\pi_1; \ldots; \pi_m; true)$ is constructed by determinizing the nondeterministic automaton and removing all accepting states (as these violate the requirement). Since the locations of the nondeterministic automaton are labeled by the phases, the locations of the deterministic automaton are then labeled with sets of phases. More precisely, the phase labeling of the location p that the automaton visits at time t in a run contains those phases j of the requirement for which the matching interpretation in the interval $[0, t]$ satisfies $(\pi_1; \ldots; \pi_j)$. Thus, whenever the automaton has detected a prefix of the requirement up to a certain phase then the corresponding index is in the phase labeling of the current location.

The usage of dense time in the automaton complicates the power set construction. Also for general timed automata,

Algorithm 1 vacuity check on $\Phi = \bigwedge_{i=1}^{n} \varphi_i$

```
for all i = 1, ..., n do
   φ_i ↦ A_i
   A := A ∥ A_i
end for
for all i = 1, ..., n do
   /* Collect locations in A with maxPhase(φ_i) ∈ phases*/
   locs := ∅
   for all locations p of A do
      if maxPhase(φ_i) ∈ phases(p)[i] then
         locs := locs ∪ {p}
      end if
   end for
   /* Check if they exists and are reachable */
   if locs = ∅ ∨ ¬Reachable(A, locs) then
      return "requirement φ_i is vacuous in Φ"
   end if
end for
return "Φ is non-vacuous"
```

Procedure 2 *Reachable*$(A, locs)$

```
for all locations p ∈ locs do
   if exists run in A visiting location p then
      return true
   end if
end for
return false
```

determinization is impossible since there is no way to represent a set of clock values. However, for the restricted language of requirements the construction is feasible and the details are in [8].

A. correct labeling

In its locations the automaton needs to remember discrete parts of the real-time behavior, i. e., if the automaton detected a prefix of the requirement up to a phase and whether the lower bounds of the duration have passed. We differentiate two cases:

Lemma 4.2: Given a requirement $\varphi = \neg(\pi_1; \ldots; \pi_m; true)$ the algorithm given in [8] determines a phase labeling for the PEA representing φ, such that:

- The index j is in the phase labeling of the current location if and only if the automaton has detected a prefix $\pi_1; \ldots; \pi_j$ of the requirement up to the j-th phase.
- The index $j^>$ (">" indicates that a time bound $\ell > k$ of phase j has not already elapsed) is in the phase labeling if the automaton has detected a prefix $\pi_1; \ldots; \widetilde{\pi}_j$ of the requirement up to the j-th phase, and the j-th phase has a lower time bound ($\pi_j = \widetilde{\pi}_j \wedge \ell > k$), and the clock measuring the duration of the last phase has not yet reached the lower bound k.

In the second case a corresponding clock measures the duration of the phase and as soon as the lower bound is reached, a new location is entered, where "$j^>$" in the *phases*-labeling is replaced with "j". The proof of Lemma 4.2 is given in [8] in Lemma 5.15 and 5.17.

B. Reachability of locations with $maxPhase(\varphi_0) \in phases$

Our algorithm checks vacuity by determining reachability of phases that contain $maxPhase(\varphi_0)$ for some property φ_0 in their labeling. This is justified by the following lemma.

Lemma 4.3: Given requirements φ_0, Φ and their representing automata A_0, A, the automaton $A_0 \| A$ contains a reachable location (p_0, p) with $maxPhase(\varphi_0) \in phases(p_0)$ if and only if φ_0 is non-vacuous in Φ.

Proof: "$\Rightarrow$": Assume that a location (p_0, p) with $maxPhase(\varphi_0) \in phases(p_0)$ is reachable in $A_0 \| A$, i. e., there is a run reaching that state. Because of Lemma 4.1 there is a matching interpretation $\mathcal{I}$ that satisfies φ_0 and Φ. Because of Lemma 4.2 this interpretation satisfies the formula $\pi_1; \ldots; \pi_{m-1}; \widetilde{\pi}_m$ if $maxPhase(\varphi_0) = m^>$, resp. $\pi_1; \ldots; \pi_{m-1}$ if $maxPhase(\varphi_0) = m - 1$. Thus for every $\widetilde{\varphi}_0 \prec \varphi_0$ the formula $\neg\widetilde{\varphi}_0$ is satisfied by this interpretation. Hence $\phi \Rightarrow (\varphi_0 \Rightarrow \widetilde{\varphi}_0)$ is not valid.

"$\Leftarrow$": Let $\psi = \pi_1; \ldots; \pi_{m-1}$ if $maxPhase(\varphi_0) = m - 1$ resp. $\psi = \pi_1; \ldots; \pi_{m-1}; \widetilde{\pi}_m$ if $maxPhase(\varphi_0) = m^>$. Further, let $\widetilde{\varphi}_0 = \neg(\psi; true)$, then $\widetilde{\varphi}_0 \prec \varphi_0$. If φ_0 is non-vacuous in Φ then we have that $\Phi \Rightarrow (\varphi_0 \Rightarrow \widetilde{\varphi}_0)$ is not valid (obviously $\widetilde{\varphi}_0 \Rightarrow \varphi_0$ holds). Hence, there is an interpretation that satisfies Φ, φ_0 and $\neg\widetilde{\varphi}_0$. A prefix of this interpretation satisfies ψ. Because of Lemma 4.1 there is a run in $A_0 \| A$ matching this prefix. This run leads to a location (p_0, p). By Lemma 4.2 we have $maxPhase(\varphi_0) \in phases(p_0)$. ■

C. vacuity detection on PEA

Theorem 4.4: Algorithm 1 correctly calculates for a set of requirements Φ whether Φ contains a requirement φ_i that is vacuous in the other requirements $\bigwedge_{i \neq j} \varphi_j$.

Proof: As described in Section III-A, it holds that the parallel product $A = \|_{i=1}^{n} A_i$ represents a set of requirements $\Phi = \bigwedge_{i=1}^{n} \varphi_i$ [8]. With Lemma 4.3 Φ is non-vacuous if for all A_i there is a reachable location p in A such that $maxPhase(\varphi_i) \in phases(p)[i]$. This property is checked in the last if-statement. Thus, Algorithm 1 is correct. ■

V. Case study: evaluation of the benefit of vacuity

The goal of our experimental study is to evaluate the practical relevance of vacuity. The primary question we need to investigate is whether the property is useful in terms of quality assurance for requirements resp. discovery of subtle specification errors. According to our preliminary results, this is indeed the case; see Table I.

To allow the experimental study we implemented Algorithm 1 in Java as depicted in Figure 3. We based our implementation on modules taken from the PEA-Toolkit [8]

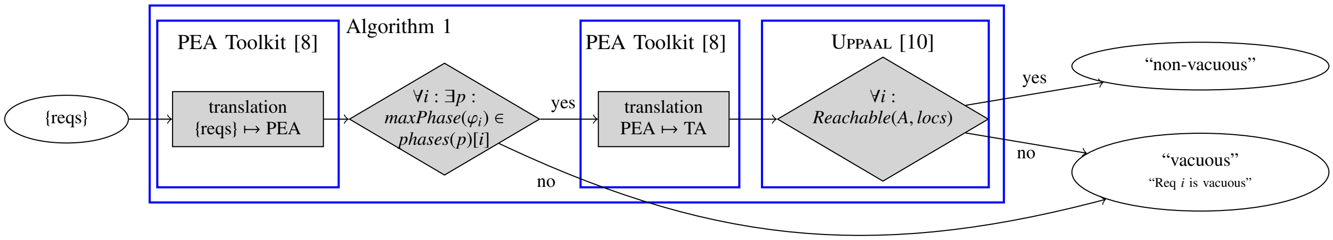


Figure 3. Prototype implementation of Algorithm 1 for checking vacuity of a set of requirements, with modules using tools for phase event automata (PEA) resp. timed automata (TA).

and the model checker Uppaal [10]. More precisely, we divided the calculation into two tasks. In Task 1 we determine $maxPhase(\varphi_i)$, use the PEA-Toolkit to build up the PEAs A_i and the parallel product A, determine for each property φ_i the locations $p \in locs$ that contain the last phase of that property ($maxPhase(\varphi_i) \in phases(p)[i]$), and check whether this set is empty. If so we return that Req_i is vacuous in Φ. Otherwise we start Task 2 which transforms A to a Timed Automaton and uses Uppaal to check whether there is a run to a location $p \in locs$. If Task 2 returns with "no" then requirement φ_i is vacuous in Φ. Otherwise if Task 2 returns with "yes" for every requirement φ_i, then the set is non-vacuous.

For the case study we took ten examples from different automotive projects at Bosch, namely projects of the application domains car multimedia, driving assistance, engine controlling, and powertrain development. Each example is a set of real-time requirements for a single software component. The specifics of the components are not relevant; hence we do not present them and just number the examples from 1 to 10. Each requirement specification had previously undergone a thorough albeit informal review. We formalized the requirements (i. e., we translated them to formal requirements as defined in Section II-B) in a somewhat lengthy process of iterations with feedback from the responsible requirement engineers. The final formalization was reviewed by a requirements engineer.

Table I refers to the results of our study. It is build up as following: The first column refers to the index of the example-component. The second column refers to the number of requirements for this component. The third and fourth column depict the problem size, measured in the number of locations and transitions in the PEA A. Column 5 and 6 refer to the calculation times of Task 1 and Task 2, i. e., the calculation time to the check whether A contains for every requirement φ_i at least one location with $maxPhase(\varphi_i)$ in *phases* (Column 5) and the calculation time to check whether such a location is reachable in a run (Column 6). All times (except those for Task 2 on Components 5 and 7, see below) are for a PC Windows XP system with 2 GHz Intel Core 2 Duo processor and 1 GB RAM. Column 7, 8, and 9 give the result of the checks, i. e., Column 7 depicts the result of Task 1, Column 8 the result of Task 2, and Column 9 the subsequent result of the vacuity-check.

As Table I shows the vacuity check guaranteed the absence of vacuities for eight components. It is interesting to note that engineers at Bosch are quite keen on this functionality. We think this is because vacuities often arise during requirements elicitation. Most of these errors are detected (and then resolved) during a manual review, but review can only detect errors, they cannot guarantee the absence of errors.

Further, the vacuity check helped to discover a subtle specification error in Component 10 that needed to be repaired (and that had gone undetected in the previous informal review). A minor change was needed to correct the requirement specification, i. e., only one requirement was changed. Debugging the requirements was quickly done, we needed about 30 Minutes to find and resolve the error.

The tool output that Req_{71} was vacuous in the set of requirements. It is defined as:

Req_{71}: If $accelerationPedal = 0$ and $brakePedalActivated$ then $regeneration$ holds after less than 1 ms.

Debugging the requirements, we found out that the antecedent $accelerationPedal = 0$ could never occur. This was due to a misinterpretation of an ambiguous requirement. Requirement Req_3 was ambiguously specified as

Req_3: The value range of the acceleration pedal is between 0 and 100.

It was interpreted as "$0 < accelerationPedal < 100$". Instead it should have been interpreted as "$0 \leq accelerationPedal \leq 100$". Thus, we resolved the ambiguity and changed Req_3 to

Req_3': The value range of the acceleration pedal is between 0 and 100 where the endpoints of the interval are included ($0 \leq accelerationPedal \leq 100$).

After that change the set of requirements was non-vacuous, as depicted in the last row of Table I.

We think that there are two reasons why this error was not detected in the manual review: first, ambiguities are difficult to detect in reviews and, second, big sets of requirements are difficult to review for humans. Reviewers have difficulties in detecting ambiguities as they often subconsciously disambiguate the requirements and think that their interpretation is the only interpretation [11]. If Req_3 and Req_{71} would have been next to each other, then in the context of Req_{71} it would have been more obvious how Req_3 had to be interpreted.

component	reqs	#locs A	#trans A	Task1	Task2	result Task1	result Task2	non-vacuous
comp. 1	10	2520	340326	3m 40s	OOM	yes	OOM	?
comp. 2	10	839	30519	5s	3s	yes	yes	yes
comp. 3	12	28	310	1s	1s	yes	yes	yes
comp. 4	17	27	729	6s	1s	yes	yes	yes
comp. 5	17	1506	207751	1m 22s	3m 30s	yes	yes	yes
comp. 6	18	633	48037	35s	2s	yes	yes	yes
comp. 7	27	639	174231	21m 32s	4m 43s	yes	yes	yes
comp. 8	27	3	9	13s	1s	yes	yes	yes
comp. 9	39	10	48	3s	1s	yes	yes	yes
comp. 10	81	7	35	6s	—	no	—	no
comp. 10'	81	21	241	2m 49s	1s	yes	yes	yes

Table I

VACUITY RESULTS FOR SEVERAL BOSCH SW-COMPONENTS. COLUMN 2 REFERS TO THE NUMBER OF REQUIREMENTS; COLUMN 3 AND 4 TO THE NUMBER OF LOCATIONS AND TRANSITIONS IN A; COLUMN 5, 6 REFER TO THE CPU TIME OF TASK 1, RESP. TASK 2 (IN MINUTES AND SECONDS); COLUMN 7, 8, 9 REFER TO THE ANALYSIS RESULTS.

However, there were more than 60 requirements specified in between. We think that it is too difficult for a human to have the specifics of so many requirements in mind. Thus, an automatic check is beneficent.

The first columns of Table I show that the problem size of checking vacuity is not directly linked to the number of requirements. E.g., the problem size (measured in the number of locations and transitions in A) of Component 10 with 81 requirements is only a fraction of the problem size of Component 1 with 10 requirements. This is due to the fact that there are requirements that decrease the size of A. E.g., a requirement "It is always the case that if $IRTest$ holds then *IRLampsOn* holds as well" reduces the space of solutions in forbidding any states with $IRTest \wedge \neg IRLamspOn$. Nevertheless, often, adding requirements will blow up the space of solutions. Thus, the algorithm may scale badly for big sets of requirements.

In particular for components 1, 5 and 7 we could execute the first task of Algorithm 1 but UPPAAL was not capable to load the timed automaton on our PC with 1 GB of RAM. We retried this on a 64-bit PC with enough RAM, however, UPPAAL is still 32-bit only and uses at most 4 GB of RAM. For Components 5 and 7 the larger machine could prove the requirements to be non-vacuous, however, for Component 1 it still runs out of memory (denoted in the table as OOM). We assume that the presented algorithm may be used to check requirements of single SW-components but that it will likely fail for the set of requirements over all SW- and HW-components. The scaling problem is induced by state explosion when building up A. Hence, to improve the performance, the state explosion problem needs to be handled.

VI. RELATED WORK

Various work exists on the topic of *vacuity* detection [12], [13], [14], [15], [4], [16], [17]. There are two main differences to our work. First, the goal of the work by [12], [13], [14], [15], [4], [16] is to check whether a *given system* satisfies the requirements only vacuously. In particular, there may be systems that satisfy the requirements *non*-vacuously. In contrast, the goal of our work is to check whether a *requirement* in a *set of requirements* is only vacuously satisfied. Our property is independent of any given system, i. e., we check whether there *exist* systems satisfying the requirements non-vacuously.

Another difference is the definition of *vacuity*. The property is often intuitively defined as the question whether a specification is satisfied in a system *in some non-interesting way* [13], [14], [4]. For example, the requirement "every request is eventually followed by a grant" is satisfied vacuously in a model with no requests. Kurshan defines a system to be *vacuous*, if the enabling condition is never satisfied, and thus the fulfilling condition is never checked [12]. Beer et al. [15] defined vacuity as follows: a formula φ is satisfied in a system S vacuously if it is satisfied in S, but some subformula ψ of φ does not affect φ in S, which means that S also satisfies $\varphi[\psi \leftarrow \psi']$ for all subformulas ψ' (here, $\varphi[\psi \leftarrow \psi']$ denotes the result of substituting ψ' for ψ in φ). In [4], [14], [16] further definitions are discussed. The idea of our property vacuity is similar, however, our definition only refers to requirements and not to a system.

Recent work on vacuity has also considered how to assess the quality of a given set of properties. Two approaches have emerged: One consists of measuring the coverage of a set of properties [18], [19], i. e., incomplete coverage exposes features of the system not adequately verified. The second approach [3], [20] consists of detecting vacuous passes in temporal logic formulae (again for a given system S). A formula φ passes vacuously in a model S if it passes in S, and there is a subformula φ' of φ that can be changed arbitrarily without affecting the outcome of model checking. One goal of this second approach is to generate so called *witnesses*.

In the work mentioned so far, vacuity detection was used to check whether a given system satisfies the requirements non-vacuously. In [17] Ball and Kupferman map vacuity detection to the testing context, i. e., they define and study vacuous satisfaction in the context of testing, and demonstrate how vacuity analysis can lead to better specifications and test suits.

To our knowledge there exists no concept of vacuity detection in the requirements context. To identify properties for requirements analysis remains an active research topic; see, e.g., [21], [22].

VII. Conclusion

We have introduced vacuity, a new property of requirements for real-time systems. We have shown that it has an interesting practical potential for ensuring the quality of real-time requirements. We have presented an algorithm to check vacuity automatically. We have implemented the algorithm to demonstrate its feasibility *in principle*, by applying it to prove the absence resp. presence of vacuity in a number of existing requirement specifications in automotive projects. Our experiments guaranteed the absence of vacuities for eight specifications and discovered a previously unknown error in one specifications, which got subsequently repaired.

In [3], Beer et al. describe that vacuity is a serious problem: "our experience has shown that typically 20% of specifications pass vacuously during the first formal-verification runs of a new hardware design, and that vacuous passes always point to a real problem in either the design or its specification or environment". Our work shows, that vacuity (i. e., vacuity) occurs as well in requirements specifications. We think, that most vacuities are detected and subsequently resolved in manual reviews, however, our work shows that an automatic check that proves the absence of vacuity is beneficial. In fact with the help of our vacuity check we discovered one error that was not discovered in the manual review.

For vacuous requirements, *all* systems satisfy the requirements *only vacuously*, provided they satisfy them at all. Thus, our proposed property might help to avoid the problems that were noted by Beer et al. in later development stages. It would be interesting to investigate in future work whether vacuity checks on requirements lead to less vacuity errors on the implemented systems.

References

[1] IEEE, "Recommended practice for sw requirements specifications," *The Institute of Electrical and Electronics Engineers IEEE Std 830-1998*, 1998.

[2] D. L. Beatty and R. E. Bryant, "Formally verifying a microprocessor using a simulation methodology," *DAC*, pp. 596–602, 1994.

[3] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in ACTL formulas," in *CAV*, 1997, pp. 279–290.

[4] O. Kupferman, "Sanity checks in formal verification," in *CONCUR*, 2006, pp. 37–51.

[5] C. Zhou and M. Hansen, *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer-Verlag, 2004.

[6] C. Zhou, C. Hoare, and A. Ravn, "A calculus of durations," *IPL*, vol. 40, no. 5, pp. 269–276, 1991.

[7] A. Post, J. Hoenicke, and A. Podelski, "rt-inconsistency: a new property for real-time requirements," in *FASE*, 2011, pp. 34–49.

[8] J. Hoenicke, "Combination of Processes, Data, and Time," Ph.D. dissertation, University of Oldenburg, July 2006.

[9] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Information and Computation*, vol. 104, pp. 2–34, 1993.

[10] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *Formal Methods for the Design of Real-Time Systems*. Springer, 2004, pp. 200–236.

[11] D. M. Berry and E. Kamsties, "Ambiguity in requirements specification," *Persp. on SW Requirements*, pp. 7–44, 2003.

[12] R. Kurshan, *FormalCheck User's Manual*. Cadence Design, Inc., 1998.

[13] K. Y. Rozier and M. Y. Vardi, "LTL satisfiability checking," in *SPIN*, 2007, pp. 149–167.

[14] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Y. Vardi, "Enhanced vacuity detection in linear temporal logic," in *CAV*. Springer, 2003, vol. 2725, pp. 368–380.

[15] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in temporal model checking," *Form. Methods Syst. Des.*, vol. 18, pp. 141–163, March 2001.

[16] A. Gurfinkel and M. Chechik, "Extending extended vacuity," in *In 5th FMCAD, LNCS 2212*. Springer, 2004, pp. 306–321.

[17] T. Ball and O. Kupferman, "Vacuity in testing," in *Proceedings of the 2nd international conference on Tests and proofs*, ser. TAP'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 4–17.

[18] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage metrics for temporal logic model checking," *Form. Methods Syst. Des.*, vol. 28, pp. 189–212, May 2006.

[19] S. Katz, O. Grumberg, and D. Geist, ""have i written enough properties?" - a method of comparison between specification and implementation," *CHARME*, pp. 280–297, 1999.

[20] M. Purandare and F. Somenzi, "Vacuum cleaning CTL formulae," *CAV*, pp. 485–499, 2002.

[21] A. G. Dahlstedt and A. Persson, "Requirements interdependencies - moulding the state of research into a research agenda," in *REFSQ*, 2003, pp. 71–80.

[22] G. S. Walia and J. C. Carver, "A systematic literature review to identify and classify software requirement errors," *Inf. Softw. Technol.*, vol. 51, no. 7, pp. 1087–1109, 2009.