rt-inconsistency: a new property for real-time requirements

Amalinda Post[1], Jochen Hoenicke[2], and Andreas Podelski[2]

[1] Robert Bosch GmbH, Stuttgart, Germany
Amalinda.Oertel@de.bosch.com
[2] University of Freiburg, Germany
{hoenicke,podelski}@informatik.uni-freiburg.de

Abstract. We introduce rt-inconsistency, a property of real-time requirements. The property reflects that the requirements specify apparently inconsistent timing constraints. We present an algorithm to check rt-inconsistency automatically. The algorithm works via a stepwise reduction to real-time model checking. We implement the algorithm using an existing module for the reduction and the UPPAAL tool for the real-time model checking. As a case study, we apply our prototype implementation to existing real-time requirements for automotive projects at BOSCH. The case study demonstrates the relevance of rt-inconsistency for detecting errors in industrial real-time requirements specifications.

1 Introduction

The specification of requirements allows us to differentiate a *correct* from an *incorrect* system. Often, however, it is difficult to get the requirements specification itself right. In the case of real-time requirements, this difficulty is exacerbated by the presence of subtle dependencies between timing constraints.

A basic problem with getting the requirements right is the lack of unambiguous properties that allow us to differentiate a good from a bad set of requirements. The IEEE Standard 830-1998 of "Recommended Practice for Software Requirements Specifications" defines eight properties, called *correctness, unambiguity, completeness, consistency, ranking for importance, verifiability, modifiability*, and *traceability* [9]. The meaning of these properties is, however, not formally defined. To identify unambiguous properties for requirements remains an active research topic; see, e.g., [3, 5, 10, 12].

In this paper, we propose a formal property of real-time requirements. The property reflects that the requirements specify apparently inconsistent timing constraints. Its violation may thus identify an (otherwise not identifiable) error in a requirements specification. We call the new property *rt-inconsistency* (for lack of a better name).

Errors in a requirement specification are often identified as *inconsistency* (the specification is unsatisfiable by any system, e.g., because it contains two contradicting requirements) or *incompleteness* (the specification lacks a requirement, e.g., because one among several possible cases is not covered) [3]. The new

property lies between inconsistency and incompleteness because the error can be repaired either by removing a requirement (as in the case of inconsistency) or by adding requirements (as in the case of incompleteness).

We demonstrate the relevance of the new property of real-time requirements specifications by a practical case study in an industrial setting. We took six existing sets of real-time requirements for automotive projects at BOSCH. Each of the six sets had undergone a thorough review. Yet, three out of the six sets of requirements contained an error identifiable through rt-inconsistency. The errors were acknowledged and subsequently repaired by the responsible engineers at BOSCH. In one of the three cases, this required a major revision of the requirements. The errors could not have been caught using the property of inconsistency; i.e., each of the six sets was consistent, as we could verify formally. We do not know of any existing property of requirement specifications that would have allowed us to catch these errors.

In the standard industrial praxis, requirements specifications must be checked manually, e.g., by peer reviews [12]. Yet, since requirements affect each other and cannot be analyzed in isolation, this is an considerable effort. Automatic checks are desirable already for small sets of requirements [6, 7, 13].

In this paper, we show that we can check the rt-inconsistency of a set of real-time requirements automatically. We present an algorithm and its theoretical foundation. The algorithm works via a stepwise reduction of the rt-inconsistency of a set of real-time requirements to a certain property of one specific real-time system (that we derive from the set of real-time requirements). I.e., it reduces a property of properties of real-time systems to a property of a real-time system.

The reduction allows us to reduce rt-inconsistency checking to *real-time model checking*. As a theoretical consequence of the reduction, the algorithm inherits the theoretical exponential worst-case complexity. Practically, the reduction allows us to capitalize on the advances of real-time model checking and the industrial strength of existing tools such as UPPAAL [2].

To implement the algorithm, we build upon pre-existing modules from [8] for deriving the real-time system from the set of given real-time requirements and the UPPAAL tool [2] for checking the real-time system. The implementation has allowed us to perform the above-mentioned case study with existing industrial examples. The primary goal was to demonstrate the practical relevance of the new property of real-time requirements. The second goal of the case study was to evaluate the practical potential of our algorithm for checking the property automatically. The results of our experiments are encouraging in this direction. They indicate that checking rt-inconsistency automatically is feasible in principle.

Roadmap. We will next illustrate rt-inconsistency informally with an example. Section 2 introduces rt-inconsistency formally, together with the formalization of real-time requirements. Section 3 presents the algorithm, with (1) the notion of automaton used for the intermediate step of the reduction, (2) the construction of such an automaton from requirements, and (3) its transformation to a timed automaton. Section 4 presents the case study.

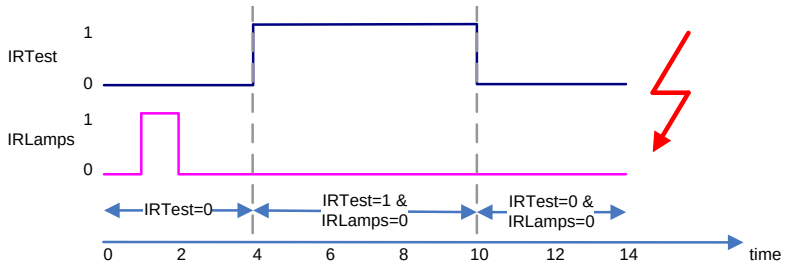


Fig. 1: Witness for the rt-inconsistency of the set of requirements $\{Req_1, Req_2\}$. The timing conflict appears immediately after the time point $t = 14$.

Example of rt-inconsistency. Consider the two informal real-time requirements below.

- Req_1: "If the system's diagnostic request *IRTest* is set, then it is never the case that the infrared lamps stay turned off for more than 10 seconds."
- Req_2: "If the system's diagnostic request *IRTest* is set, then it is never the case that the infrared lamps will be on in the next 6 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 circumstances may arise where the two requirements are in conflict. Consider the trace depicted in Figure 1. At time point $t = 4$ the diagnostic request *IRTest* is set. By Req_1, the infrared lamps must be turned on within the next 10 seconds. In the (right-open) time interval $[4, 10)$ *IRTest* stays. It disappears at the time point $t = 10$. By Req_2, the infrared lamps are turned off for at least 6 seconds (and, thus, during the whole time interval $[10, 16]$). I.e., for any possible continuation of the trace after $t = 14$, the two requirements clash: by Req_1, the infrared lamps are turned on, and by Req_2, they are turned off for further two seconds; i.e., the requirements claim contradicting valuations for the infrared lamps. The set of the two requirements, while consistent, is rt-inconsistent!

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

- Req_2': "If the system's diagnostic request *IRTest* is set *and it was not set in the last 10 seconds*, then it is never the case that the infrared lamps will be on in the next 6 seconds."

Another way to resolve the rt-inconsistency is to add both, the requirements Req_3 and Req_4 to the set containing Req_1 and Req_2.

- Req_3: "Once the system's diagnostic request *IRTest* is set, *IRTest* stays active for at most 3 seconds."
- Req_4: "Once the system's diagnostic request *IRTest* disappears, *IRTest* is absent for at least 10 seconds."

2 Defining rt-inconsistency

To find rt-inconsistencies we need to interpret requirements on both the infinite time axis $\mathcal{R}_{\geq 0}$ and on finite time intervals $[0, t]$ from zero to some time point t. A convenient way to obtain a suitable formalization of requirements is to borrow the notation of the *Duration Calculus* [14, 15]. 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 1. We first restate Req_1 in a less ambiguous form.

- Req_1: If the system's diagnostic request *IRTest* is set *at a time when the infrared lamps are turned off*, then it is never the case that the infrared lamps stay turned off for more than 10 seconds.

We introduce the predicates *IRTest* and *IRLampsOn* (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, and $t_3, t_1 < t_2 < t_3$ such that *IRTest* is true between t_1 and t_2, and *IRLampsOn* is false between t_1 and t_3, 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 *IRTest* is true and *IRLampsOn* is false,
3. the third phase (from time point t_2 to t_3) underlies the constraint that *IRLampsOn* is false and the constraint that its length (the difference between t_3 and t_2) is greater than 10.
4. the fourth phase (from time point t_3 until infinity) does not underlie any constraint.

In formal syntax, the requirement Req_1 is expressed as the formula φ_1 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).

$$\varphi_1 = \neg(\mathit{true}\,;\, \lceil \text{IRTest} \wedge \neg \text{IRLampsOn} \rceil\,;\, \lceil \neg \text{IRLampsOn} \rceil \wedge \ell > 10\,;\, \mathit{true})$$

The formalization of three other requirements from Section 1 is given below.

$$\varphi_2 = \neg(\mathit{true}\,;\, \lceil \text{IRTest} \rceil\,;\, \mathit{true} \wedge \ell < 6\,;\, \lceil \text{IRLampsOn} \rceil\,;\, \mathit{true})$$

$$\varphi_3 = \neg(\mathit{true}\,;\, \lceil \text{IRTest} \rceil \wedge \ell > 3\,;\, \mathit{true})$$

$$\varphi_4 = \neg(\mathit{true}\,;\, \lceil \text{IRTest} \rceil\,;\, \lceil \neg \text{IRTest} \rceil \wedge \ell < 10\,;\, \lceil \text{IRTest} \rceil\,;\, \mathit{true})$$

Syntax. Formally, the syntax of phases π and requirements φ is defined by the BNF below. The predicate symbol P refers to a fixed set $Preds$ of predicate symbols (for *observations* whose truth values change over time). The correctness of the algorithm presented in this paper (more precisely, the soundness of the answer "rt-consistent") relies on the fact that we have only *strict* inequalities ($\ell > k$ and $\ell < k$) in the definition of phases π. The extension to non-strict inequalities ($\ell \geq k$ and $\ell \leq k$) would complicate the algorithm unnecessarily, i.e., without being motivated by practical examples.

$$\begin{array}{ll} \text{phase} & \pi \ ::= \ true \mid \lceil P \rceil \mid \pi \wedge \ \ell > k \mid \pi \wedge \ \ell < k \\ \text{requirement} & \varphi \ ::= \ \neg(\pi_1 \ ; \ \ldots ; \pi_n \ ; \ true) \end{array}$$

A set of requirements denotes their conjunction. We overload the metavariable φ for requirements and sets of requirements.

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 $t \in \mathcal{R}_{\geq 0}$) an observation, i.e., a valuation of the family of given predicates P.

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

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.

We use "*prefix* of $\mathcal{I}$ until t" for the special case of the segment of $\mathcal{I}$ from 0 to t, i.e., for the restricted function $(\mathcal{I}, [0, t])$. Given two interpretations $\mathcal{I}$ and $\mathcal{I}'$ we say that the prefix of $\mathcal{I}$ until t *coincides* with the prefix of $\mathcal{I}'$ until t if the (restricted) functions are equal, i.e., $(\mathcal{I}, [0, t]) = (\mathcal{I}', [0, t])$.

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

$$\begin{array}{ll} (\mathcal{I}, [b, e]) \models \lceil P \rceil & \text{if } \ \mathcal{I}(P)(t) \ \text{ is true for } t \in [b, e] \text{ and } b \neq e \\ (\mathcal{I}, [b, e]) \models \ell > k & \text{if } \ (e - b) > k \\ (\mathcal{I}, [b, e]) \models \pi_1 \, ; \, \pi_2 & \text{if } \ (\mathcal{I}, [b, m]) \models \pi_1 \text{ and } (\mathcal{I}, [m, e]) \models \pi_2 \ \text{ for some } m \in [b, e] \end{array}$$

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

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

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 prefix of $\mathcal{I}$ until t satisfies φ).

Safety. A requirement φ, which we have defined to be a negated formula of the form $\varphi = \neg(\pi_1 \,;\, \ldots ; \pi_n \,;\, \mathit{true})$, expresses that "something bad may not happen", where "bad" refers to the possibility of splitting the time axis into $n+1$ intervals satisfying the phases $\pi_1, \ldots, \pi_n$, and true, respectively. The requirement φ expresses a so-called *safety property* (which means that "if an execution violates φ, then there is a prefix of the execution such that any execution with this prefix violates φ"). The syntactic restriction that the last phase is true is crucial here.

rt-inconsistency. The satisfaction of a requirement is defined not only for a 'full' interpretation on the infinite time axis ("$\mathcal{I} \models \varphi$") but also for the prefix of an interpretation until a time point t ("$(\mathcal{I}, [0,t]) \models \varphi$"). We need both satisfaction relations in our definition of rt-inconsistency.

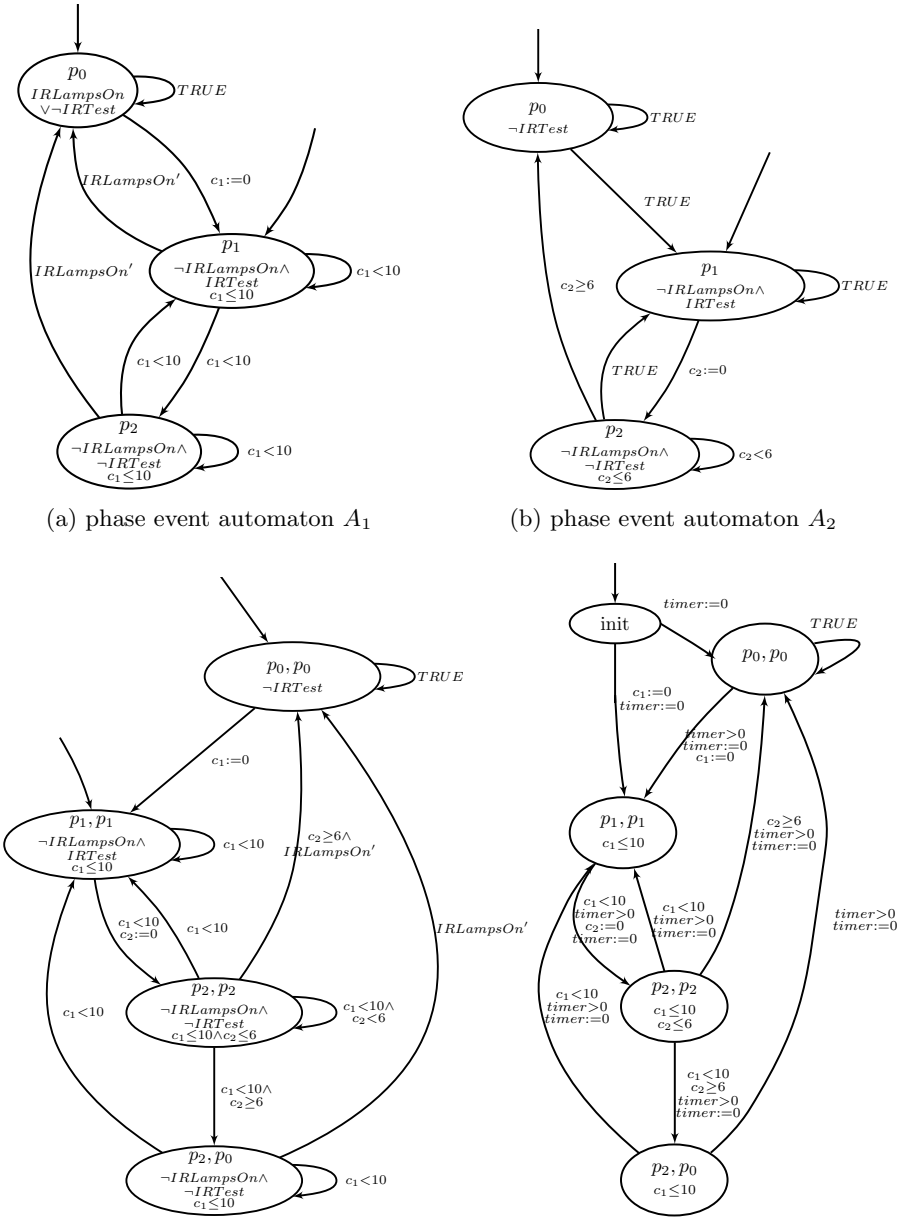
Definition 1. [rt-inconsistency] A set of requirements φ is *rt-inconsistent* if there exists a prefix $(\mathcal{I}, [0,t])$ of an interpretation $\mathcal{I}$ until a time point t that satisfies φ but no extension of the prefix to a full interpretation does (i.e., on the whole time axis), formally:

$$\begin{aligned} &(\mathcal{I}, [0,t]) \models \varphi \\ &\quad \mathcal{I}' \quad \not\models \varphi \quad \text{if} \;\; (\mathcal{I}', [0,t]) = (\mathcal{I}, [0,t]) \end{aligned}$$

i.e., the full interpretation $\mathcal{I}'$ does not satisfy φ whenever its prefix until t coincides with the prefix of the interpretation $\mathcal{I}$ until t.

Remark 1. In essence, a set of requirements φ is rt-inconsistent if it does not exclude the existence of a prefix of an interpretation which leads to a conflict. The conflict prevents the possibility of the extension of the prefix to a full interpretation. More precisely, in the setting of Definition 1 (of an interpretation $\mathcal{I}$, a time point t, and the prefix $(\mathcal{I}, [0,t])$ satisfying φ), the rt-inconsistency is due to one of two reasons.

1. No extension of the prefix $(\mathcal{I}, [0,t])$ to any time point t' after t is possible without violating φ (i.e., $(\mathcal{I}, [0,t']) \not\models \varphi$ for all $t' > t$).
   In other words: The conflict inherent in φ hits directly after t. The conflict does not hit at any time point in the closed interval $[0,t]$ but it does hit when time leaves the interval, i.e., at any time point t' after t. No passing of time after t is possible without a conflict.
2. There are (uncountably many) time points t' after t such that the extension of the prefix $(\mathcal{I}, [0,t])$ to t' is possible without violating φ, but there exists a time point t_0 after all those time points t' such that the extension of the prefix $(\mathcal{I}, [0,t])$ to t_0 violates φ (i.e., $(\mathcal{I}, [0,t_0]) \not\models \varphi$ for some $t_0 > t$).
   In other words: The conflict inherent in φ strikes after the time point t_0 after t. The conflict does not hit at t or at any other time point in the right-open interval $[0,t_0)$ but it does hit at the time point t_0. No passing of time into t_0 is possible without a conflict.



(a) phase event automaton A_1 (b) phase event automaton A_2

(c) phase event automaton $A = A_1||A_2$ (d) timed automaton S

Fig. 2: Algorithm 1 of Section 3 applied to the set of the requirements φ_1 and φ_2 from Section 2 (formalizing Req_1 and Req_2 from Section 1) constructs the phase event automata A_1 and A_2, forms their parallel product $A = A_1||A_2$ and transforms A into the timed automaton S.
$\varphi_1 = \neg(true\,;\, \lceil \text{IRTest} \wedge \neg \text{IRLampsOn} \rceil\,;\, \lceil \neg \text{IRLampsOn} \rceil \wedge \ell > 10\,;\, true)$
$\varphi_2 = \neg(true\,;\, \lceil \text{IRTest} \rceil\,;\, true \wedge \ell < 6\,;\, \lceil \text{IRLampsOn} \rceil\,;\, true)$

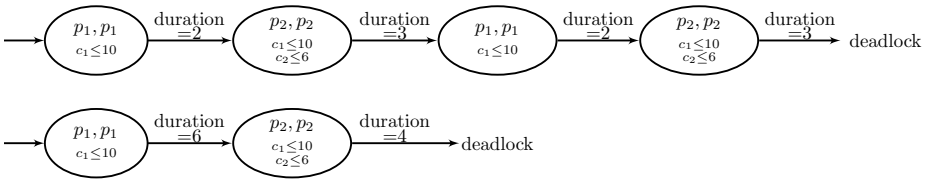


Fig. 3: Algorithm 1 of Section 3 applied to the set of the requirements φ_1 and φ_2 from Section 2 returns a run of S leading to a deadlock as a witness for the answer "φ is rt-inconsistent", e.g. one of the depicted runs. The depicted witnesses suggest adding φ_3 respectively φ_4.

Remark 2. It is easy to find examples that show, respectively, that neither does the rt-consistency imply the absence of *deadlocks* for every system that satisfies the real-time requirement φ, nor does the rt-inconsistency imply the presence of deadlocks in every system that satisfies φ.

Remark 3. Although the idea of using rt-inconsistency to detect flaws in real-time requirements is new (and in particular no algorithm for deciding rt-inconsistency was given before), the property has already appeared in a different form in [1]. There, however, the concern is the completeness of proof methods for the treatment of real time in standard linear temporal logic. The goal in [1] is a method to prove that a system specification S satisfies a requirement φ. Here, S is an "old-fashioned" program where updates of a program variable *now* over the reals are used to model the progress of time. If φ is a liveness property, then one may need to add to S a fairness assumption NZ for the scheduler that updates *now*. Still, a proof method may be incomplete (i.e., it may be incapable of showing the correctness of S together with NZ wrt. the requirement φ). The completeness of a proof method may hold only for correctness problems where the pair (S, NZ) is *machine-closed*. Machine-closure of S for NZ is essentially the same as rt-consistency of S. The formal definitions are not directly comparable (since machine-closure is formalized using sequences of pairs of states and time points, as opposed to continuous interpretations $\mathcal{I}$).

3 Checking rt-inconsistency

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

As already explained in the introduction, we reduce the problem of checking the rt-inconsistency of φ to the problem of checking a certain temporal property (existence of *deadlocks*) of a real-time system, formally a *timed automaton* S. We construct S from φ. More precisely, we first construct a certain kind of automaton A, a so-called *phase event automaton* (PEA), from φ and then transform A

Algorithm 1 check rt-inconsistency of set of requirements $\varphi = \{\varphi_1, \ldots, \varphi_n\}$

for all $i = 1, \ldots, n$ **do**
  $A_i :=$ req2pea(φ_i) {transform requirement to phase event automaton}
end for
$A := A_1 \| \ldots \| A_n$ {form the parallel product of phase event automata}
$S :=$ pea2ta(A) {transform phase event automaton to timed automaton}
{call timed model checker for existence of deadlocks}
if (S is deadlock-free) **then**
  return "φ is rt-consistent"
else
  return "φ is rt-inconsistent" {return path to deadlock in timed automaton}
end if

into a timed automaton S such that φ, A, and S are related in a sense that we will make formal.

Fig. 2 presents the intermediate results of the different steps of the application of Algorithm 1 to the rt-inconsistent set of the requirements φ_1 and φ_2 from Section 2 (formalizing Req_1 and Req_2 from Section 1). Algorithm 1 transforms the requirements φ_1 and φ_2 into the phase event automata A_1 and A_2 in Figure 2a resp. Figure 2b. It forms their parallel product $A = A_1||A_2$ which is given in Figure 2c. It then transforms A into the timed automaton S given in Figure 2d. After that it checks whether S contains a deadlock. In this example, it finds a deadlock and returns the answer "φ is rt-inconsistent", together with a witness given in Figure 3 (a run of S leading to a deadlock). The first witness depicted in Figure 3(where *IRTest* toggles too quickly) suggests adding the requirement φ_4. The second witness (where *IRTest* stays on too long) suggests adding the requirement φ_3.

3.1 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 observations, i.e., to valuations of predicates). Syntactically, a phase event automaton resembles a timed automaton in that it has the same notion of *clocks*; semantically, there are differences such as in the minimal duration between transitions. Below, for a set of variables X, we use X' 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 $\mathcal{L}(X)$ to denote a set of formulae with free variables in X.

A *phase event automaton* (PEA) is a tuple $A = (P, V, C, E, s, I, P^0)$ where

- P is the set of locations p (*phases*),
- C is the set of clocks c,
- V is the Boolean variables P (*observation predicates*),
- E is a set of *transitions* of the form (p, g, X, p') where p and 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 (i.e., g specifies also the updates of Boolean variables), and X is the set of clocks that are reset to 0, i.e., $E \subseteq P \times \mathcal{L}(C \cup V \cup V') \times 2^C \times P$,
- 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 $k \in \mathcal{R}_{\geq 0}$, 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 $\gamma : C \to \mathcal{R}_{\geq 0}$, 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) and (p', β', γ'), 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 $\gamma + t$ otherwise.

The *duration* of a run r is the sum of the durations t in its quadruples. An infinite run r is *non-Zeno* if its duration is infinite. An *unextendable run* of A is a finite run r of A which is not prefix of any non-Zeno run of 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 t if one adds up the durations of all quadruples in r preceding it. We omit the cumbersome formal definition (which is analogous for finite runs and prefixes of an interpretation).

An interpretation $\mathcal{I}$ is *accepted* by A, formally $\mathcal{I} \in \mathcal{L}(A)$, if there is a non-Zeno run r of A that matches $\mathcal{I}$. The next lemma implies that every run r of A gives rise to an interpretation $\mathcal{I}$ accepted by A.

Lemma 1. *For every non-Zeno run r of a phase event automaton A there exists an interpretation $\mathcal{I}$ such that r 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 $A_1 || A_2$ (defined in the canonical way) represents their conjunction $\varphi_1 \wedge \varphi_2$.

3.2 Characterizing rt-inconsistency via phase event automata

We will use the algorithm of [8, 11] which, given a requirement φ, constructs a phase event automaton A that represents φ. In this section, we show that the properties of the algorithm that are stated in Lemmas 2 and 3 (and proven in [8]) suffice to characterize the rt-inconsistency of φ. From now on, we refer to the construction of A from φ by the algorithm of [11].

Lemma 2. *The phase automaton A constructed from φ is* deterministic*; i.e., if A accepts the prefix of the interpretation until the time point t, then there is exactly one run r of A that matches $\mathcal{I}$ for duration t.*

Lemma 3. *The prefix of the interpretation $\mathcal{I}$ until the time point t satisfies the requirement φ if and only if it is accepted by the phase automaton A constructed from φ; i.e., $(\mathcal{I}, [0, t]) \models \varphi$ if and only if $(\mathcal{I}, [0, t]) \in \mathcal{L}(A)$.*

The "$\Leftarrow$" direction of Lemma 3 relies on the restriction to *strict* inequalities in the definition of the syntax of φ in Section 2. The restriction entails that the PEA constructed from φ contains only non-strict clock invariants "$c \leq k$".

Theorem 1. *The set of requirements φ is rt-inconsistent if and only if the phase event automaton A constructed from φ contains an unextendable run.*

Proof. "$\Rightarrow$" If φ is *rt-inconsistent* and $\mathcal{I}$ is an interpretation as in Definition 1, then the prefix of $\mathcal{I}$ until the time point t satisfies φ and thus, by Lemma 3, it is accepted by A. Hence, by the definition of acceptance, there is a run r in A that matches $\mathcal{I}$ for duration t. We are done if we show that r is unextendable. Assume, for a proof by contraction, that there is a non-Zeno run r' of A that extends r. By Lemma 1, r' matches an interpretation $\mathcal{I}'$. Hence, again by the definition of acceptance, A accepts $\mathcal{I}'$ and also the prefix of $\mathcal{I}'$ until t', for every time point t'. By Lemma 3, the prefix of $\mathcal{I}'$ until t' satisfies φ, for every time point t'. Thus, by the definition of the satisfaction relation, the full interpretation $\mathcal{I}'$ satisfies φ. Since the prefix of $\mathcal{I}$ until the time point t coincides with the one of $\mathcal{I}'$, we have found an interpretation $\mathcal{I}'$ as in Definition 1, i.e., one which cannot exist.

"$\Leftarrow$" If r is an unextendable run of A for, say, the duration t, then, by Lemma 3, there is an interpretation $\mathcal{I}$ such that the prefix of $\mathcal{I}$ until t matches r. By Lemma 3, the prefix of the interpretation $\mathcal{I}$ until the time point t satisfies the requirement φ. We are done if we show that there exists no interpretation $\mathcal{I}'$ that satisfies φ and whose prefix until t coincides with the one of $\mathcal{I}$. Assume, for a proof by contraction, that such an $\mathcal{I}'$ exists. Then, since A represents φ, A accepts φ. By the definition of acceptance, there exists a non-Zeno run r' of A that matches $\mathcal{I}'$. By Lemma 2, A is deterministic, i.e., r' coincides with r for the duration t, or: r is a prefix of r'. Thus, we have found a non-Zeno run of A which has r as a prefix, which cannot exist by the assumption that r is an unextendable run of A. □

Algorithm 2 transform phase event automaton A to timed automaton S

if A has more than one initial location p_{0i} **then**
  add a new initial location, with an transition to every p_{0i}
end if
normalize transitions such that each guard is a conjunct of literals
for all transitions (p, g, X, p') of A **do**
  if $s(p) \wedge g \wedge (s(p'))'$ is unsatisfiable **then**
    remove this transition
  end if
end for
remove unreachable locations
remove all literals from the guards except clock constraints
set all state invariants to true
for all transitions (p, g, X, p') of A **do**
  $g := g \wedge timer > 0$
  $X := X \cup \{timer\}$
  for all constraints $c \leq k$ in $I(p')$ where $c \notin X$ **do**
    $g := g \wedge c < k$
  end for
end for

3.3 Characterizing rt-inconsistency via timed automata

From phase event automata to timed automata. Algorithm 2 transforms a phase event automaton to a timed automaton. The transformation extends a similar one in [8] which preserves reachability but not unextendability. The transformation introduces a special clock *timer* in order to capture the fact that, in a phase event automaton, every location getting active has to stay active for a non-zero period of time. The clock *timer* is reset when the location is entered, and every outgoing transition must satisfy the guard that specifies $timer > 0$. To prevent introducing artificial deadlocks, the new transformation strengthens the guard of the outgoing transition with the strict inequality $c < k$ derived from the non-strict inequality $c \leq k$ in the clock invariant of the target location (thus, after a transition, there is always some time left to stay in the target location).

Lemma 4. *Every run of the phase event automaton A corresponds to a run of the timed automaton S constructed from A (with the same sequence of locations and clock valuations), and vice versa.*

Deadlock. Following [2], a timed automaton S contains a *deadlock* if there is a reachable state (p, γ) such that for all durations $d > 0$ there is no action successor of $(p, \gamma + d)$. In particular, the self-loop is not enabled. We will next characterize rt-inconsistency in terms of deadlocks, and thus obtain the correctness of Algorithm 1.

Theorem 2 (Correctness). *The timed automaton S constructed from the set of requirements φ via the phase event automaton A and Algorithm 2 contains a deadlock if and only if the set of requirements φ is rt-inconsistent.*

Proof. "⇒" By Lemma 4, a run to a deadlock state (p, γ) in S corresponds to an unextendable run in A to the same location p with the same clock valuation γ without any further transition being possible (the successor would be reachable in S as well). By Theorem 1, φ is rt-inconsistent.

"⇐" We assume that S is deadlock-free and show that φ is not rt-inconsistent. By Theorem 1 it suffices to show that, for every finite run r of A, there is a non-Zeno run r' of A that extends r.

By Lemma 4, a finite run r of A corresponds to a run in the timed automata S leading to some state (p, γ). We extend r by staying in p until the bound of its invariant is reached (the clock invariants in S are non-strict by the construction). If p has no bound, r' can be chosen as the non-Zeno run which stays in p forever. Otherwise, since S contains no deadlock, there must be an action successor from p, at a different location whose time bound is not yet reached. The infinite iteration of this reasoning leads to an infinite run in the timed automaton and, again by Lemma 4, to an infinite run r' in the PEA A. We need to show that r' is non-Zeno.

Assume that there exists a time point, say, t, such that r' never reaches t. If b is the smallest among the bounds of all clocks in the invariant of some location in S, then each location can be visited at most t/b times in r' (since the location is active until the clock reaches the bound and then the clock must be reset before the location can be visited again). Since there are only finitely many locations in S, r' is a finite run. This is in contradiction to our construction of r'. Hence there is no such time point t and r' is a non-Zeno run. □

4 Using rt-inconsistency in a case study

The goal of our experimental study is to evaluate the practical relevance of rt-inconsistency. The primary question we need to investigate is whether the property is useful to improve a requirement specification, namely by providing a criterion that helps to differentiate good from bad, or desirable from undesirable requirement specifications. According to our preliminary results, this is indeed the case; see Table 1.

Table 1 refers to six examples from different automotive projects at BOSCH. 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 6 (first column). The second column refers to the number of requirements in the example. 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 2) in a somewhat lengthy process of iterations with feedback from the responsible requirement engineers. We had the final formalization reviewed by a requirements engineer.

As Table 1 shows, three out of the six examples have an error that is identifiable as rt-inconsistency. I.e., for Components 1, 2, and 3, the rt-inconsistency identifies an actual flaw in the requirement specification that needed to be re-

	reqs	TA nodes	TA transitions	$\{\text{reqs}\} \mapsto \text{TA}$	UPPAAL	rt-consistent?	*correction*
1	9	900	69183	34s	37s	no	A:3, C:4
2	10	2520	418365	322s	28min 49s	no	A:4, C:4
3	10	895	36541	9.4s	1h 16min	no	A:7, D:2, C:5
4	13	28	310	1s	$< 1s$	yes	—
5	16	27	729	6s	< 1s	yes	—
6	17	1614	318267	160s	n/a	n/a	n/a

Table 1: Checking rt-inconsistency for existing examples of sets of real-time requirements for software components in automotive projects at BOSCH using a prototypical implementation (Fig. 3) of the algorithm presented in Section 2, on a PC Windows XP system with 2GHz Intel Core 2 Duo processor and 1GB RAM, whereas only one core was used. The examples are numbered from 1 to 6. The columns refer to: the size of the input in the number of requirements, the number of nodes resp. the (much higher!) number of transitions of the timed automaton (TA) obtained by the automatic translation of the set of requirements (via the translation to a phase event automaton), the time used for the automatic translation, the time used by the timed model checker UPPAAL for checking the existence of deadlocks in the timed automaton (where n/a here means out-of-memory when loading the input), the outcome of the rt-inconsistency check, and the cost of the *correction* of an rt-inconsistency (in the number of requirements that were newly added (A), changed (C), and deleted (D), respectively).

paired. As the last column shows, major changes were needed to correct the requirement specification. E.g., for Component 3, two of the existing requirements were deleted, five were changed, and seven new requirements were added. If positive, the rt-inconsistency test returns a run of the time automaton that is helpful for analyzing the error; yet, debugging the requirement specification demands the help of a requirements engineer with domain knowledge and takes a considerable amount of time (about half a day to one day per example for debugging and fixing). Most of the detected flaws based on conflicts similar to the conflict described in the example of Section 1.

The requirement specifications for Component 4 and Component 5 are examples for rt-consistency. Without these examples one might wonder if rt-inconsistency implies a high degree of specificity (obtainable only through a large number of precise requirements) which cannot be found in realistic examples. The examples, requirement specifications that are rt-consistent *as is* (i.e., not after a revision), indicate that this is not the case. For safety-critical systems (e.g., in the automotive domain), the high degree of specificity enforced by rt-consistency seems appropriate.

In order to evaluate the practical relevance of rt-inconsistency, the second question we need to investigate is whether the property can be checked on realistic examples automatically. For the purpose of proof of concept, we have implemented the algorithm presented in Section 3; see Figure 4. Our prototypical, non-optimized implementation relies on existing tool kits [8, 2] for implementing three procedures called by the algorithm: the translation of a set of requirements

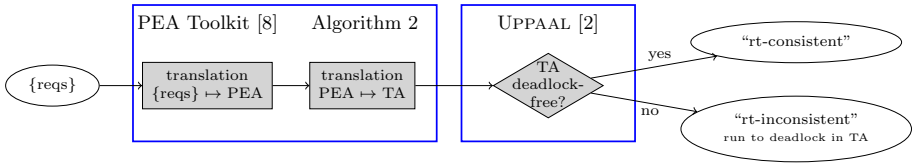


Fig. 4: Prototype implementation of Algorithm 1 for checking rt-inconsistency of a set of requirements, with modules using tools for phase event automata (PEA) resp. timed automata (TA).

to a phase event automata (req2pea), the translation of a phase event automaton into a timed automaton (pea2ta), the check for the existence of deadlocks in a timed automaton. The results of our experiments (see Table 1) show that in five out of six examples, the algorithm is able to automatically prove resp. disprove rt-inconsistency. The examples are relatively small but they are realistic (and apparently so complex that a manual review is no longer sufficient). The results indicate that checking rt-inconsistency automatically is feasible in principle.

As one could expect by the theoretical complexity of the algorithm, the check does not succeed for every input (and, as often with the automatic analysis tools, the size of the input does not necessarily correlate with the difficulty of its analysis for the tool). In the sixth example, UPPAAL runs out of memory when loading the timed automaton generated from the phase event automaton in this example. We still need to analyze the cause (which is not solely the size of the input), but it is clear that UPPAAL is not optimized for the timed automata generated in this setting; in the examples of Table 1, the number of transitions is two orders of magnitude larger than the number of nodes.

An experimental study is incomplete (and somewhat unsatisfying) if it does not expose deficiencies of the evaluated concepts (and opportunities for improvement). The sixth example shows that the state explosion problem occurs not only in theory, but also in practice.

Furthermore, the case study shows that the state explosion is not directly related to the number of requirements: although Component 4 and Component 5 consist of more requirements than the first three components the number of nodes and transitions of A is much smaller for Component 4 and 5. This is due to the fact that not all requirements blow up the state space exponentially. There are also requirements that solely constrain the state space and thus reduce the number of states, e.g., a requirement like "If *IRTest* holds, then *Diagnosis.Running* holds as well" forbids every state in which holds $IRTest \wedge \neg Diagnosis.Running$.

5 Conclusion and future work

We have introduced rt-inconsistency, a new property of requirements for real-time systems. We have shown that it has an interesting practical potential for

unambiguously identifying subtle timing errors in a requirements specification. We have presented an algorithm to check rt-inconsistency automatically. We have implemented the algorithm to demonstrate its feasibility *in principle*, by applying it to prove the absence resp. presence of rt-inconsistency in a number of existing requirement specifications in automotive projects. Our experiments discovered previously unknown errors in some of those specifications, errors which got subsequently repaired.

As already mentioned, one line of future work is to adapt heuristics and optimizations from real time model checking to checking rt-inconsistency. Another, more speculative line of research are methods to automatically correct (or help to correct) an rt-inconsistent set of requirements, possibly using algorithms from *real-time synthesis* [4].

References

1. M. Abadi and L. Lamport. An old-fashioned recipe for real time. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 1–27, London, UK, 1992. Springer-Verlag.
2. G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. 2004.
3. A. G. Dahlstedt and A. Persson. Requirements interdependencies - moulding the state of research into a research agenda. In *REFSQ*, pages 71–80, 2003.
4. R. Ehlers, R. Mattmüller, and H.-J. Peter. Combining symbolic representations for solving timed games. *FORMATS*, pages 107–121, 2010.
5. J. H. Hayes. Building a requirement fault taxonomy: Experiences from a NASA verification and validation research project. In *ISSRE*, 2003.
6. M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency analysis of state-based requirements. In *IEEE Trans. on SW Engineering*, pages 3–14, 1995.
7. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. on SW Eng. and Methodology*, 5(3):231–261, 1996.
8. J. Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, July 2006.
9. IEEE. *Recommended Practice for Software Requirements Specifications*. 1998.
10. N. G. Leveson. System safety in computer-controlled automotive systems. In *SAE World Conference*, 2000.
11. R. Meyer, J. Faber, J. Hoenicke, and A. Rybalchenko. Model checking duration calculus: a practical approach. *Formal Asp. Comput.*, 20(4-5):481–505, 2008.
12. G. S. Walia and J. C. Carver. A systematic literature review to identify and classify software requirement errors. *Inf. Softw. Technol.*, 51(7):1087–1109, 2009.
13. L. Yu, S. Su, S. Luo, and Y. Su. Completeness and consistency analysis on requirements of distributed event-driven systems. In *TASE*, 2008.
14. C. Zhou and M. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems*. Springer-Verlag, 2004.
15. C. Zhou, C. Hoare, and A. Ravn. A calculus of durations. *IPL*, 1991.