Incremental methods for checking real-time consistency*

Thierry Jéron[1], Nicolas Markey[1], David Mentré[2], Reiya Noguchi[2], and Ocan Sankur[1]

[1] Univ Rennes, INRIA, CNRS, Rennes (France)
firstname.lastname@inria.fr
[2] Mitsubishi Electrics R&D Centre Europe, Rennes (France)
initial-of-firstname.lastname@fr.merce.mee.com

Abstract. Requirements engineering is a key phase in the development process. Ensuring that requirements are consistent is essential so that they do not conflict and admit implementations. We consider the formal verification of *rt-consistency*, which imposes that the inevitability of definitive errors of a requirement should be anticipated, and that of *partial consistency*, which was recently introduced as a more effective check. We generalize and formalize both notions for discrete-time timed automata, develop three incremental algorithms, and present experimental results.

1 Introduction

In the process of developing computer systems, requirement engineering consists in defining, documenting and maintaining the requirements. Requirements can be of different nature, but since we are interested in timed systems, i.e. systems where time constraints are of importance, we will focus here on timed functional ones. Requirements are the primary phase of the development process, and are used to partly drive the testing campaign in order to check that they are indeed satisfied by the implementation. In a formal approach, it is thus important to design formal requirements that are consistent, *i.e.* that are not contradictory and admit implementations that conform to them.

In this paper, we study two prominent consistency notions studied in the literature for real-time system requirements, called *rt-consistency* [PHP11a] and *partial consistency* [Bec19]. Partial consistency concentrates the notion of consistency on Simplified Universal Patterns (SUP) [BTES16] which are simple real-time temporal patterns used to define real-time requirements, essentially comprising an assumption (named *trigger*), a guarantee (named *action*), together with timed constraints on delays of these and between them. The advantage of SUPs is that they define a specification language that is expressive enough yet

* This work was partially funded by ANR project Ticktac (ANR-18-CE40-0015), and by a MERCE/Inria collaboration.

easy to understand, even by non experts. The counterpart is that the notion of partial consistency is specific to them and tricky.

Rt-consistency requires that all finite executions that do not violate the requirements, have infinite extensions that satisfy all of requirements. Put differently, this means that if an implementation produces a finite execution whose all continuations necessarily lead to the violation of some requirement, then there must be a requirement that is already violated by the finite execution. In simple words, inevitability of errors should be anticipated by the set of requirements. Thus, rt-consistency ensures that the set of requirements is well designed and sane. This is interesting in that it may reveal conflicts between requirements and catch subtle problems, but it is rather expensive to check. Several directions can be investigated to mitigate this complexity: restrict to sub-classes of requirements, in particular SUPs, restrict to subsets of requirements, examine alternative and cheaper notions of consistency. However these lead in general to false positives and false negatives, and avoiding them requires additional conditions or checks.

Partial consistency is one of these alternative notions of consistency that only considers pairs of SUP requirements. It checks that if there are possibly different executions that trigger both requirements and satisfy one of them, then there should be a common execution in which both requirements are triggered and satisfied. This check is perhaps better understood as a necessary condition for the rt-consistency of *subsets* of requirements (but this does not imply the rt-consistency of the whole set). We formalize this link in this paper. The general motivation is to gain in efficiency, both by restricting to pairs of requirements, but also by focusing on particular situations where inconsistencies may arise. Nevertheless partial consistency can still be costly to check.

Contributions. We address the efficiency issue mentioned above by considering an incremental approach to checking consistency and finding inconsistencies in real-time requirements. In fact, rt-consistency and (bounded) partial consistency are rather expensive to check already on small examples, and because of the state-space explosion problem (which is a classical problem when composing several systems or properties), there is no hope that the approaches would scale to large sets of requirements. Our algorithms improve the scalability of this approach by allowing one to check larger sets of requirements. We also define a new notion of incremental consistency, and allow to get different degrees of confidence about consistency (up to full rt-consistency).

We show that checking rt-consistency can be reduced to CTL model checking for discrete-time systems, providing an alternative approach to duration calculus and timed automata model checking of [PHP11a]. Then, we develop incremental algorithms for checking rt-consistency and a variant of partial consistency generalized for automata. Inconsistencies are searched by starting with small batches of requirements. Whenever we find a counterexample to consistency, we either confirm it (by checking that it fulfills the other requirements) or start the analysis again with more precision by adding a new requirement in the batch. This helps us to scale our analysis to larger sets of requirements. This idea is applied separately for both consistency notions. Moreover, we formalize

the relation between the two notions, showing how to obtain counterexamples to rt-inconsistency from counterexamples to partial consistency. Due to space constraints, all proofs are given in the appendix of the full paper [JMM+20].

Related works. Consistency notions appear naturally in the contract-based design of systems [BCN+18]. In this setting, consistency is defined as the existence of an implementation of a contract, which relates environment and system behaviors via assumptions and guarantees. The related notion of *existential consistency* is studied in [ESH14], where consistency consists in the existence of an execution satisfying the requirements.

Simplified Universal Patterns were introduced in [BTES16] to simplify the writing of requirements by non-experts. The patterns are in the form of an assumption and guarantee. In this paper, the notion of consistency ensures the existence of an execution which realizes one requirement (both the assumption and the guarantee) without violating any other one. In [BTES16], the authors also use coverage notions to measure sets of consistent executions to give a quantitative measure of consistency. The notion considered there is thus related to *non-vacuity* (see e.g. [PHP11b]).

More reactive notions were studied as in [AHL+17] where consistency requires that the system should react to uncontrollable inputs along the execution so as to satisfy all requirements. The notion is thus formalized as a game between the system and the environment, and an SMT-based algorithm is given to check consistency within a given bound. This notion thus relies on alternation of quantifiers at each step. Rt-consistency and partial consistency, which we consider in this paper, lie between the two extreme approaches (that is simply existential *versus* game semantics). In fact, a single quantifier alternation is needed to define rt-consistency (see Section 2.4). The rt-consistency checking algorithm of [PHP11a] considers systems in a continuous-time setting, and uses duration calculus and timed automata model checking. We consider discrete-time systems (with unit delays rather than arbitrary real-valued delays).

2 Definitions

2.1 Computation Tree logic

We use CTL to characterize certain kinds of inconsistencies. CTL formulas are defined as $\mathsf{CTL} \ni \phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{AX}\phi \mid \mathbf{EG}\phi \mid \mathbf{E}\phi\mathbf{U}\phi$, where p ranges over AP. CTL formulas are evaluated at the root of computation trees. We thus consider computation trees labeled by valuations of atomic propositions: a tree t is a set of finite non-empty traces, i.e. words over 2^{AP}, closed under prefix, hence containing exactly one trace of size 1 (called its root, and denoted with $r(t)$). We denote $\prec_{\mathrm{p}}$ the prefix ordering on traces. Given a node in the tree represented by a trace $\sigma \in t$, we write t_σ for the subtree of t rooted at σ (i.e., the set of all traces σ' such that $\sigma \cdot \sigma' \in t$). We write $\sigma[i]$ for the prefix of length i of σ. That a tree t satisfies a formula $\phi \in \mathsf{CTL}$ is defined as follows:

$$t \models p \qquad \iff \quad p \in r(t)(p)$$

$$
\begin{array}{lcl}
t \models \neg\phi & \iff & t \not\models \phi \\
t \models \phi \vee \phi' & \iff & t \models \phi \text{ or } t \models \phi' \\
t \models \mathbf{AX}\phi & \iff & \forall \sigma \in t.\ (t_{\sigma[1]} \models \phi) \\
t \models \mathbf{E}\phi\mathbf{U}\phi' & \iff & \exists \sigma \in t.\ (t_\sigma \models \phi' \text{ and } \forall \sigma'.\ (r(t) \prec_{\mathrm{p}} \sigma' \prec_{\mathrm{p}} \sigma) \Rightarrow t_{\sigma'} \models \phi) \\
t \models \mathbf{EG}\phi & \iff & \exists \sigma \in t.\ (\forall i.\ t_{\sigma[i]} \models \phi)
\end{array}
$$

Using $\mathbf{AX}$, we can define $\mathbf{EX}$ by $\mathbf{EX}\phi \equiv \neg\mathbf{AX}\neg\phi$. Similarly, $\mathbf{AF}\phi \equiv \neg\mathbf{EG}\neg\phi$ means that ϕ holds along any infinite branch of the tree, and finally $\mathbf{A}\phi\mathbf{U}\phi' \equiv \mathbf{AF}\phi' \wedge \neg\mathbf{E}(\neg\phi')\mathbf{U}(\neg\phi \wedge \neg\phi')$ means that along all infinite branch, ϕ' eventually holds and ϕ holds at all intermediary nodes.

2.2 Timed automata

We consider requirements expressible by a class of *timed automata* (TA) [AD90]. These extend finite-state automata with variables, called *clocks*, that can be used to measure (and impose constraints on) delays between various events along executions. More precisely, given a set $\mathcal{X} = \{c_i \mid 1 \leq i \leq k\}$ of clocks, the set of *clock constraints* is defined by the grammar: $g ::= c \sim n \mid g \wedge g$, where $c \in \mathcal{X}$, $n \in \mathbb{N}$, and $\sim \in \{<, \leq, =, \geq, >\}$. Let $\mathcal{C}(\mathcal{X})$ denote the set of all clock constraints.

We consider integer-valued clocks whose semantics of constraints is defined in the expected way: given a clock valuation $v\colon \mathcal{X} \to \mathbb{N}$, a constraint $g \in \mathcal{C}(\mathcal{X})$ is true at v, denoted $v \models g$, if the formula obtained by replacing each occurrence of c by $v(c)$ holds. For a valuation $v\colon \mathcal{X} \to \mathbb{N}$, an integer $d \in \mathbb{N}$, and a subset $R \subseteq \mathcal{X}$, we define $v + d$ as the valuation $(v + d)(c) = v(c) + d$ for all $c \in \mathcal{X}$, and $v[R \leftarrow 0]$ as $v[R \leftarrow 0](c) = 0$ if $c \in R$, and $v[R \leftarrow 0](c) = v(c)$ otherwise. Let $\mathbf{0}$ be the valuation mapping all variables to 0.

We consider timed automata as monitors of the evolution of the system through the observation of values of Boolean variables. We thus consider a set $\mathsf{AP} = \{b_i \mid 1 \leq i \leq n\}$ of atomic propositions, and define the set of Boolean constraints $\mathcal{B}(\mathsf{AP})$ as the set of all propositional formulas built on AP.

Definition 1. *A* timed automaton *is a tuple $\mathcal{T} = \langle S, S_0, AP, \mathcal{X}, T, F\rangle$ where S is a finite set of states, $S_0 \subseteq S$ is a set of initial states, AP is a finite set of atomic propositions, $\mathcal{X}$ is a finite set of clocks, $T \subseteq S \times \mathcal{B}(AP) \times \mathcal{C}(\mathcal{X}) \times 2^{\mathcal{X}} \times S$ is a finite set of transitions, and $F \subseteq S$ is the set of accepting states.*

We distinguish the following classes of timed automata. A *safety* timed automaton is such that there are no transitions from $S \setminus F$ to F. Conversely a *co-safety* timed automaton is such that there are no transitions from F to $S \setminus F$.

For a transition $t = (s, c, g, r, s') \in T$ of a timed automaton, we define $\mathsf{src}(t) = s$, $\mathsf{tgt}(t) = s'$, $\mathsf{bool}(t) = c$, $\mathsf{guard}(t) = g$, and $\mathsf{reset}(t) = r$. Note that guards are pairs of Boolean and timed guards that can be interpreted (and will be noted) as conjunctions since the two types of guards do not interfere.

With a timed automaton $\mathcal{T}$, we associate the infinite-state automaton $\mathcal{S}(\mathcal{T}) = \langle Q, Q_0, \Sigma, D, Q_F\rangle$ that defines its semantics, where

- the set of states Q contains all *configurations* $(s, v) \in S \times \mathbb{N}^{\mathcal{X}}$;
- the initial states are obtained by adjoining the null valuation (all clocks are mapped to zero) to initial states S_0, i.e. $Q_0 = S_0 \times \mathbf{0}$;
- $\Sigma = 2^{AP}$ is the alphabet of actions, i.e. valuations of all Boolean variables;
- transitions in D are combinations of a transition of the TA and a one-time-unit delay. Formally, given a letter $\sigma \in \Sigma$ and two configurations (s, v) and (s', v'), there is a transition $((s, v), \sigma, (s', v'))$ in D if, and only if, there is a transition (s, c, g, r, s') in T such that $\sigma \models c$ and $v \models g$, and $v' = (v[r \leftarrow 0]) + 1$.
- $Q_F = F \times \mathbb{N}^{\mathcal{X}}$ is the set of accepting configurations.

Our semantics thus makes it compulsory to alternate between taking a transition of the TA (possibly a self-loop) and taking a one-time-unit delay. Self-loops can be used to emulate invariants in states.

The transition system $\mathcal{S}(\mathcal{T})$ is infinite because we impose no bound on the values of the clocks during executions. However, as in the setting of TA [AD90], the exact value of a clock is irrelevant as soon as it exceeds the largest integer constant with which it is compared. We could thus easily modify the definition of $\mathcal{S}(\mathcal{T})$ in such a way that it only contains finitely many states.

A *run* of $\mathcal{T}$ is a run of its associated infinite-state automaton $\mathcal{S}(\mathcal{T})$. It can be represented as a sequence along which configurations and actions alternate: $(s_0, v_0) \cdot \sigma_1 \cdot (s_1, v1) \cdot \sigma_2 \cdots (s_n, v_n) \cdots$. A finite run is accepted if it ends in Q_F.

A *trace* of a run is its projection on the set of actions. In other terms, it is a finite or infinite sequence $\sigma = (\sigma_i)_{0 \leq i < l}$ of actions where $l \in \mathbb{N} \cup \{+\infty\}$ is the length of σ, denoted by $|\sigma|$. Finite traces belong to Σ^* and infinite ones to Σ^ω. A finite trace is accepted by $\mathcal{T}$ if a run on that trace is accepted. We note $\mathrm{Tr}(\mathcal{T})$ the set of accepted traces. For $P \subseteq Q$ we will also note $\mathrm{Tr}_P(\mathcal{T})$ the set of traces of runs ending in P.

Consider the following sets, where F is an atomic proposition denoting Q_F:

- $\mathbf{Success}_{\mathcal{T}} = F \wedge \mathbf{AG} F$: accepting configurations from which non-accepting configurations are unreachable are called *success*; notice that it is impossible to escape from $\mathbf{Success}_{\mathcal{T}}$ since $\mathbf{Success}_{\mathcal{T}} \implies \mathbf{AG}\ \mathbf{Success}_{\mathcal{T}}$;
- $\mathbf{Error}_{\mathcal{T}} = \neg F \wedge \mathbf{AG} \neg F$: non-accepting configurations from which accepting configurations are unreachable are called *error*; notice also that it is impossible to escape from $\mathbf{Error}_{\mathcal{T}}$ since $\mathbf{Error}_{\mathcal{T}} \implies \mathbf{AG}\ \mathbf{Error}_{\mathcal{T}}$;

Note that in safety TAs, $\neg F \implies \mathbf{AG} \neg F$ since it is impossible to escape from the set of non-accepting configurations, thus $\mathbf{Error}_{\mathcal{T}} = \neg F$; symmetrically in co-safety TAs, $F \implies \mathbf{AG}\ F$ since it is impossible to escape from the set of accepting configurations, thus $\mathbf{Success}_{\mathcal{T}} = F$.

We require that our TAs are *complete*, meaning that from any (reachable) configuration (s, v), and for any subset b of AP, there is $t = (s, c, g, r, s') \in T$ such that $b \models c$ and $v \models g$. This is no loss of generality since missing transitions can be directed to a trap state, and self-loops can be added to allow time elapse.

The TAs that we consider are also *deterministic*: for any two transitions (s, c_1, g_1, r_1, s_1) and (s, c_2, g_2, r_2, s_2) issued from a same source s, if both $c_1 \wedge c_2$

and $g_1 \wedge g_2$ are satisfiable, then $s_1 = s_2$ and $r_1 = r_2$. Examples of complete, deterministic TAs expressing requirements are depicted on Fig. 2, in Example 1.

We consider the product of timed automata, as follows:

Definition 2. *Given two TAs* $\mathcal{T}_1 = \langle S_1, S_{1,0}, AP_1, \mathcal{X}_1, T_1, F_1 \rangle$ *and* $\mathcal{T}_2 = \langle S_2, S_{2,0}, AP_2, \mathcal{X}_2, T_2, F_2 \rangle$ *with disjoint clock sets (i.e.,* $\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$*), their* product $\mathcal{T}_1 \otimes \mathcal{T}_2$ *is a TA* $\mathcal{T} = \langle S, S_0, AP, \mathcal{X}, T, F \rangle$ *where* $S = S_1 \times S_2$*,* $S_0 = S_{1,0} \times S_{2,0}$*,* $AP = AP_1 \cup AP_2$*,* $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$*,* $F = F_1 \times F_2$ *and the set of transitions is defined as follows: there is a transition* $((s_1, s_2), c, g, r, (s'_1, s'_2))$ *in* T *if there are transitions* $(s_1, c_1, g_1, r_1, s'_1)$ *in* T_1 *and* $(s_2, c_2, g_2, r_2, s'_2)$ *in* T_2 *with* $c = c_1 \wedge c_2$*,* $g = g_1 \wedge g_2$*, and* $r = r_1 \cup r_2$*.*

Note that completeness and determinism are preserved by product. The product of TAs can be generalized to an arbitrary number of TAs: for a set $\mathcal{R} = \{R_i\}_{i \in I}$ of requirements, each specified by a TA $\mathcal{T}_i(R_i)$, we note $\otimes\mathcal{R}$ the requirement specified by the TA $\otimes_{i \in I} \mathcal{T}_i(R_i)$.

Note that in this definition, clocks of factor automata are disjoint, while atomic propositions are not, which may cause conflicts in guards of the product, and possibly inconsistencies as will be seen later. Also note that the product of two automata visits its accepting states if both automata do ($F = F_1 \wedge F_2$), while by complementation it visits non-accepting states if one of the automata does ($\neg F = \neg F_1 \vee \neg F_2$). For the product automaton, we directly define (without relying on F) $\mathbf{Success}_{\mathcal{T}} = \mathbf{Success}_{\mathcal{T}_1} \wedge \mathbf{Success}_{\mathcal{T}_2}$ and $\mathbf{Error}_{\mathcal{T}} = \mathbf{Error}_{\mathcal{T}_1} \vee \mathbf{Error}_{\mathcal{T}_2}$, and both are trap sets. The definitions of **Error** and **Success** thus depend on the context: these are defined by the formulas $\neg F_i \wedge \mathbf{AG}\neg F_i$ and $F_i \wedge \mathbf{AG}F_i$ for the TAs $\mathcal{T}_i$ representing the given requirements; for the *products* of these automata, $\mathbf{Error}_{\mathcal{T}}$ (resp. $\mathbf{Success}_{\mathcal{T}}$) is the disjunction (resp. conjunction) of $\mathbf{Error}_{\mathcal{T}_i}$ (resp. $\mathbf{Success}_{\mathcal{T}_i}$) of their operands. Notice that we have $\mathbf{Success}_{\mathcal{T}} = F \wedge \mathbf{AG}F$, but only $\mathbf{Error}_{\mathcal{T}} \subseteq \neg F \wedge \mathbf{AG}\neg F$. The inclusion is in general strict, but becomes an equality when both $\mathcal{T}_1$ and $\mathcal{T}_2$ are safety TAs.

For the rest of this document, we consider complete deterministic timed automata (CDTAs for short) with accepting states F.

2.3 Timed automata as requirements

We use complete deterministic TAs to encode requirements and identify the requirements with the CDTAs that define them. Remember that **Error** (resp. **Success**) are sets of configurations from which one cannot escape. Intuitively, entering an **Error** (resp. **Success**) configuration of a CDTA corresponds to violating (resp. satisfying) the corresponding requirement definitively:

Definition 3. *For any requirement* R *defined by a complete deterministic timed automaton and any finite or infinite trace* σ*, we write* σ **fails** R *if running* σ *in* R *enters* $\mathbf{Error}_R$*, and write* σ **succeeds** R *if it enters* $\mathbf{Success}_R$*.*

Note that for a finite trace σ, it could be the case that it does not hit $\mathbf{Error}_R$ (resp. $\mathbf{Success}_R$) but all infinite continuations inevitably do. We are particularly interested in such cases; we thus define the following notations for finite traces:

Definition 4. *For a finite trace σ, and a requirement R defined by a CDTA, we write σ* **I-fails** *R if for all infinite traces σ', $\sigma \cdot \sigma'$* **fails** *R. Similarly σ* **I-succeeds** *R if for all infinite traces σ', $\sigma \cdot \sigma'$* **succeeds** *R.*

Clearly, for finite traces, **fails** (resp. **succeeds**) is stronger than **I-fails** (resp. **I-succeeds**). Indeed σ **fails** R (σ **succeeds** R) means reaching a configuration in $\mathbf{Error}_R$ (resp. $\mathbf{Success}_R$), while σ **I-fails** R (σ **I-succeeds** R) means reaching a configuration in **AF** $\mathbf{Error}_R$ (resp. **AF** $\mathbf{Success}_R$). And $\mathbf{Error}_R$ implies **AG** $\mathbf{Error}_R$, which implies **AF** $\mathbf{Error}_R$ (and similarly for $\mathbf{Success}_R$).

For a given trace σ, and *set* of timed automata $\mathcal{R} = \{\mathcal{T}_i\}_{i \in I}$, we write σ **fails** $\mathcal{R}$ (resp σ **succeeds** $\mathcal{R}$) to mean that σ **fails** $\otimes \mathcal{R}$ (resp. σ **succeeds** $\otimes \mathcal{R}$). Note the following simple facts: given $\mathcal{R}' \subseteq \mathcal{R}$, for any finite trace σ, if σ **fails** $\mathcal{R}'$ then σ **fails** $\mathcal{R}$, and if σ **I-fails** $\mathcal{R}'$ then σ **I-fails** $\mathcal{R}$, while conversely, if σ **succeeds** $\mathcal{R}$ then σ **succeeds** $\mathcal{R}'$, and if σ **I-succeeds** $\mathcal{R}$ then σ **I-succeeds** $\mathcal{R}'$.

Simplified Universal Patterns (SUP). TAs can be used to express the semantics of Simplified Universal Pattern (SUP) [TBH16, Bec19], a pattern language that is used to define requirements. Compared to TAs, SUPs offer a more intuitive but less expressive way of writing requirements. Since partial consistency was introduced for SUP, we briefly introduce them. An SUP has the following form:

$$(\mathrm{TSE}, \mathrm{TC}, \mathrm{TEE})[\mathrm{Tmin}, \mathrm{Tmax}] \xrightarrow{[\mathrm{Lmin},\mathrm{Lmax}]} (\mathrm{ASE}, \mathrm{AC}, \mathrm{AEE})[\mathrm{Amin}, \mathrm{Amax}],$$

where TSE, TC, TEE, ASE, AC, AEE, are Boolean formulas on a set AP of atomic propositions, Tmin, Tmax, Lmin, Lmax, Amin, Amax are integers.

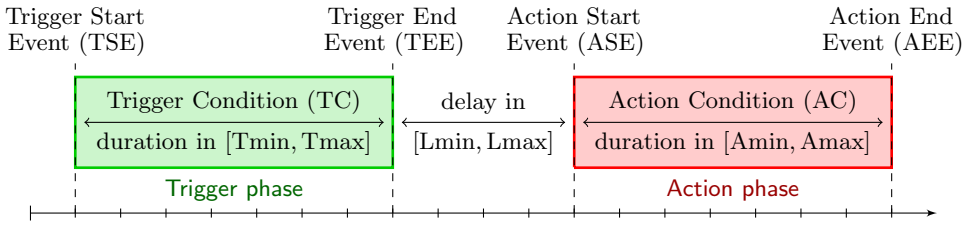

Fig. 1. Intuitive semantics of SUP

Figure 1 illustrates the intuitive semantics of SUP. A *trigger phase* (left) is realized, if TSE is confirmed within a duration in [Tmin; Tmax], that is, if TC holds until TEE occurs; otherwise the trigger is *aborted*. For the SUP instance to be satisfied, following each realized trigger phase, an *action phase* must be realized: an action phase starts with ASE within [Lmin; Lmax] time units after the end of the trigger phase, and then AC must hold until AEE occurs within [Amin, Amax] time units. Otherwise, the SUP is *violated*. Following [Bec19], one can translate SUP instances (and repetitions of them) into complete deterministic timed automata. In fact all SUPs can be written as safety or co-safety CDTAs.

Example 1. Consider the following two SUPs: R_1 : request $\xrightarrow{[3;4]}$ response, and R_2 : repair $\xrightarrow{[5;5]}$ $\neg$response$[3;3]$, where an SUP of the form $(p,p,p)[0;0] \xrightarrow{[0;1]} (q,q,q)[0;0]$ is written $p \xrightarrow{[0;1]} q$.

The first requirement models a system that has to respond to any request within 3 to 4 time units. The second requirement states that if the system enters a maintenance phase, then it will be off (and cannot respond) after 5 time units, and for a duration of 3 time units. Figure 2 displays the (safety) automata encoding these two SUPs where E_i states are non-accepting trap states and all other ones are accepting.

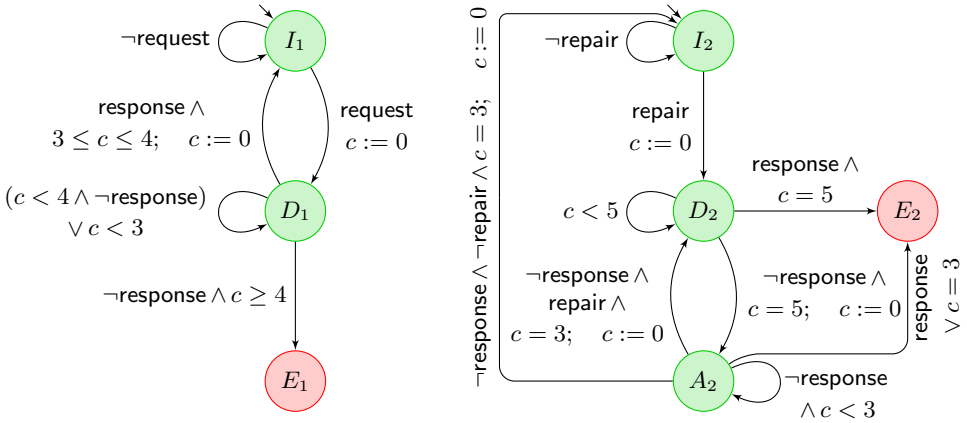

Fig. 2. Timed automata encoding R_1 and R_2

2.4 Consistency notions

RT-consistency. We reformulate the original rt-consistency notion, introduced in [PHP11a].

Definition 5. *Let $\mathcal{R}$ be a set of requirements. Then $\mathcal{R}$ is* rt-consistent *if, and only if, for all finite traces σ, if σ* **I-fails** *$\mathcal{R}$ then σ* **fails** *$\mathcal{R}$.*

Thus the set $\mathcal{R}$ is rt-consistent if any finite trace that inevitably fails, immediately fails. This is indeed equivalent to the formulation in [PHP11a], which says that all finite traces not violating any requirement can be extended to an infinite trace not violating any of them (i.e. $\neg(\sigma$ **fails** $\mathcal{R})$ implies $\neg(\sigma$ **I-fails** $\mathcal{R}))$. Notice that rt-consistency (w.r.t. $\mathbf{Error}_{\mathcal{R}}$) could be generalized to rt-consistency w.r.t $\mathbf{Success}_{\mathcal{R}}$: if σ **I-succeeds** $\mathcal{R}$ then σ **succeeds** $\mathcal{R}$; and all following results easily generalize to rt-consistency w.r.t. $\mathbf{Success}_R$ with similar treatment.

Observe that even when all individual requirements are rt-consistent (i.e., for all $R \in \mathcal{R}$ and all traces σ, it holds σ **I-fails** $R \implies \sigma$ **fails** R) their conjunction (i.e. the product $\otimes\mathcal{R}$) may not be rt-consistent; for instance, taken individually, both requirements R_1 and R_2 of Example 1 are rt-consistent, but their product is not, as explained in Example 2). Rt-consistency requires that **fails** and **I-fails** be equivalent for all traces in the product automaton.

Rather than using duration calculus as in [PHP11a], we show that CTL model checking can be used in a discrete-time setting to check rt-consistency. In CTL, rt-consistency of $\mathcal{R}$ can be expressed by requiring $\mathbf{AF}\ \mathbf{Error}_{\mathcal{R}} \Leftrightarrow \mathbf{Error}_{\mathcal{R}}$ at all reachable states. Since $\mathbf{Error}_{\mathcal{R}}$ is absorbing, a trace ending in a configuration in $\neg\mathbf{Error}_{\mathcal{R}} \wedge \mathbf{AF}\ \mathbf{Error}_{\mathcal{R}}$ is a *witness to rt-inconsistency.* Moreover, only configurations in $\neg\mathbf{Error}_{\mathcal{R}}$ need to be traversed to reach such configurations; and such a configuration exists if, and only if, configurations exist in $\neg\mathbf{Error}_{\mathcal{R}}$ with all immediate successors in $\mathbf{Error}_{\mathcal{R}}$, i.e., $\mathbf{AX}\ \mathbf{Error}$ is true. In fact, we obtain the following property.

Lemma 1. *A given set of requirements $\mathcal{R}$ has a witness to rt-inconsistency if, and only if, $\mathcal{R} \models \mathbf{E}(\neg\mathbf{Error}_{\mathcal{R}}\ \mathbf{U}\ (\neg\mathbf{Error}_{\mathcal{R}} \wedge \mathbf{AX}\ \mathbf{Error}_{\mathcal{R}}))$.*

Example 2. The requirements in Example 1 are not rt-consistent: consider a finite trace σ where the repair signal is received, followed 3 time units later with a request. Then $\neg(\sigma\ \mathbf{fails}\ R_1 \wedge R_2)$; the joint run of the automata are as follows:

$$(I_1, I_2, \tfrac{c_1=0}{c_2=0}) \xrightarrow[(+\text{delay})]{\mathsf{repair}} (I_1, D_2, \tfrac{c_1=1}{c_2=1}) \xrightarrow[(+\text{delay})]{\star} (I_1, D_2, \tfrac{c_1=2}{c_2=2})$$
$$\xrightarrow[(+\text{delay})]{\star} (I_1, D_2, \tfrac{c_1=3}{c_2=3}) \xrightarrow[(+\text{delay})]{\mathsf{request}} (D_1, D_2, \tfrac{c_1=1}{c_2=4}).$$

From this last configuration, it can be checked that no continuations of this trace will avoid reaching E_1 or E_2: indeed, both automata will first loop in their current states D_1 and D_2, reaching configuration $(D_1, D_2), c_1 = 2, c_2 = 5$. In order to avoid visiting E_2, the next two steps must satisfy $\neg$response, thereby reaching $(D_1, A_2), c_1 = 4, c_2 = 2$. From there, we have a conflict: if response is true at the next step, R_2 reaches E_2, while if response is false, R_1 reaches E_1.

Now, assume we add the following requirement, which expresses that no request can be received during maintenance: $R_3 : \mathsf{repair} \rightarrow \neg\mathsf{request}[5; 5]$. This rules out the above trace, and it can be checked that the resulting set of requirements is now rt-consistent.

Partial consistency. *Partial consistency* was introduced in [Bec19] as an alternative, more efficient check to detect inconsistencies in SUP requirements. We here generalize this notion to CDTAs. The name *partial consistency* might be misleading since it does not directly compare with rt-consistency: partial inconsistency identifies risky situations for pairs of requirements that could cause rt-inconsistency of the whole set. In this paper, we formalize this link, and show how to lift witnesses of partial inconsistencies to witnesses of rt-inconsistencies.

In a requirement R_i, let us call *action* configurations those configurations allowing to enter immediately $\mathbf{Error}_{R_i}$ (i.e. satisfying $\mathbf{EX}\ \mathbf{Error}_{R_i}$)[3]. Then, action configurations that have an infinite continuation that avoids $\mathbf{Error}_{R_i}$ are characterized by $\mathbf{EX}\ \mathbf{Error}_{R_i} \wedge \neg\mathbf{AF}\ \mathbf{Error}_{R_i}$. Now, $\mathbf{EX}\ \mathbf{Error}_{R_1} \wedge \mathbf{EX}\ \mathbf{Error}_{R_2}$ means we are simultaneously at action configurations of both R_1 and R_2. In

[3] For SUPs, such configurations correspond to *action* phases, hence the name.

this case, even though there are separate continuations that avoid $\mathbf{Error}_{R_1}$ and $\mathbf{Error}_{R_2}$, there may not be a common one. In our generalization of partial consistency, we focus our attention to checking that a common continuation exists for this type of configurations which are seen as "risky" since they are in the proximity of error.

Let $\mathbf{reach}_k(\mathcal{R})$ denote the set of configurations of $\mathcal{R}$ reachable within k steps.

Definition 6. *Consider requirements R_1, R_2 and a set $\mathcal{R}'$ of requirements. We say that R_1 and R_2 are* partially consistent w.r.t. $\mathcal{R}'$ *if for all $k \in \mathbb{N}$,*

if, for all $i \in \{1,2\}$,

$$\exists s_i \in \mathbf{reach}_k(\mathcal{R}_1 \times \mathcal{R}_2 \times \mathcal{R}').\ s_i \models \mathbf{EX}\,\mathbf{Error}_{R_1} \wedge \mathbf{EX}\,\mathbf{Error}_{R_2} \wedge \neg\mathbf{AF}(\mathbf{Error}_{\mathcal{R}'} \vee \mathbf{Error}_{R_i})$$

then

$$\exists s \in \mathbf{reach}_k(\mathcal{R}_1 \times \mathcal{R}_2 \times \mathcal{R}').\ s \models \mathbf{EX}\,\mathbf{Error}_{R_1} \wedge \mathbf{EX}\,\mathbf{Error}_{R_2} \wedge \neg\mathbf{AF}(\mathbf{Error}_{\mathcal{R}'} \vee \mathbf{Error}_{R_1} \vee \mathbf{Error}_{R_2}). \quad (1)$$

Partial consistency requires that for all depths k, if infinite traces for both requirements can be found leading to an action configuration within k steps, and neither violate the requirement itself nor $\mathcal{R}'$, then a single infinite trace must exist that reaches action configurations of both requirements within k steps, and does not violate any of them, nor $\mathcal{R}'$. Therefore, a witness of partial inconsistency is a number $k \geq 0$ and two infinite sequences σ_1 and σ_2 such that, σ_i reaches actions phases of both requirements within k steps, and never fails R_i or $\mathcal{R}'$, such that there are no infinite traces that do so without violating one of the requirements R_1, R_2 or $\mathcal{R}'$.

We establish that partial consistency is a necessary condition for the rt-consistency of the *subset* $\mathcal{R}' \cup \{R_1, R_2\}$, since counterexamples for the former provide counterexamples for the latter:

Lemma 2. *If R_1 and R_2 are partially inconsistent w.r.t. $\mathcal{R}'$, then $\mathcal{R}' \cup \{R_1, R_2\}$ is rt-inconsistent.*

To efficiently find counterexamples to partial consistency, we consider the following approximation, which is similar to that of [Bec19] but generalized to CDTAs. Given bounds $\alpha, \beta > 0$, requirements R_1, R_2 are (α, β)-*bounded partially consistent* if for all $k \leq \alpha$,

if, for all $i \in \{1,2\}$,

$$\exists s_i \in \mathbf{reach}_k(\mathcal{R}_1 \times \mathcal{R}_2 \times \mathcal{R}').\ s_i \models \mathbf{EX}\,\mathbf{Error}_{R_1} \wedge \mathbf{EX}\,\mathbf{Error}_{R_2} \wedge \neg\mathbf{AF}_{\alpha-k}(\mathbf{Error}_{\mathcal{R}'} \vee \mathbf{Error}_{R_i})$$

then

$$\exists s \in \mathbf{reach}_k(\mathcal{R}_1 \times \mathcal{R}_2 \times \mathcal{R}').\ s \models \mathbf{EX}\,\mathbf{Error}_{R_1} \wedge \mathbf{EX}\,\mathbf{Error}_{R_2} \wedge \neg\mathbf{AF}_{\alpha+\beta-k}(\mathbf{Error}_{\mathcal{R}'} \vee \mathbf{Error}_{R_1} \vee \mathbf{Error}_{R_2}). \quad (2)$$

where $\mathbf{AF}_l\phi$ means the inevitability of ϕ within l steps, which can be expressed in CTL as the disjunction of all formulas of the form $\mathbf{AX}(\phi \vee \mathbf{AX}(\cdots \phi \vee \mathbf{AX}\phi))$ with l repetitions of $\mathbf{AX}$. Thus the approximation consists in looking for witnesses of

bounded length for the satisfaction of the Equation 1). But notice that witnesses of failure of Equation 2 are not witnesses of failure of Equation 1 which require infinite traces (see below).

Example 3. We consider again the requirements of Example 1. Requirements R_1 and R_2 are not partially consistent under empty $\mathcal{R}'$: as soon as a trace reaches action configurations of both requirements, error states of any of them can be avoided, but not both of them. Under requirement R_3, requirements R_1 and R_2 cannot reach their action phases simultaneously, so that with $\mathcal{R}' = \{R_3\}$, those two requirements are partially consistent.

There are a few differences with the original definition of partial consistency of [Bec19]. First, partial consistency of [Bec19] only checks the very first trigger of the traces. Moreover, it focuses on situations where, after respective triggers, no timing allows requirements to avoid being simultaneously in action phases. In our case, $\mathbf{EX\ Error}_{R_1} \wedge \mathbf{EX\ Error}_{R_2}$ does not restrict simultaneous action phases to such particular ones. Thus we can detect more subtle inconsistencies.

The second difference is that the bounded approximation in [Bec19] checks for the existence of a lasso-shaped execution in the automata that recognize the SUP requirements. The advantage of this is that such a lasso describes an infinite execution, so if partial consistency holds, so does the bounded approximation; while the converse is not true. In other terms, a witness for bounded partial inconsistency is a witness for partial inconsistency. In our case, we do not look for a lasso in the premise of (2), so this implication does not hold. We do prove, on the other hand, that rt-consistency implies (2); see Lemma 5.

Third, in [Bec19], $\mathcal{R}'$ contains only a specific type of requirements called invariants. In our case, $\mathcal{R}'$ is an arbitrary subset of the requirement set.

3 Incremental algorithms

We provide three incremental methods to check rt-consistency of a given set of requirements $\mathcal{R}$. The first one provides strong guarantees and can assess the rt-consistency of the whole set $\mathcal{R}$, or that of its subsets, and uses CTL model checking. The second one uses SAT/SMT solving and scales to larger sets. It can *detect* rt-inconsistencies of $\mathcal{R}$, but cannot *prove* rt-consistency; it can only ensure partial consistency. The third one can quickly find rt-inconsistencies.

In all algorithms we consider a set $\mathcal{R} = \{R_i\}_{i \in I}$ of requirements, each given as a CDTA, and their product $\otimes\mathcal{R}$.

3.1 Incremental rt-consistency checking

In this section, we present our incremental algorithm for rt-consistency checking. Unlike the previous work of [Hoe06], which uses duration calculus [ZHR91], our algorithm is based on *computation tree logic* (CTL) model checking. Rt-inconsistency of $\mathcal{R}$ reduces to checking whether a finite trace exists along which $\mathbf{Error}_{\mathcal{R}}$ remains false such that, from the last configuration, $\mathbf{Error}_{\mathcal{R}}$ is inevitable.

Input: A set $\mathcal{R}$ of requirements given as CDTAs, $2 \leq n \leq |\mathcal{R}|$
$\phi(\mathcal{R}) \leftarrow \mathbf{E}[\neg\mathbf{Error}_{\mathcal{R}}\ \mathbf{U}(\neg\mathbf{Error}_{\mathcal{R}} \wedge \mathbf{AX}\ \mathbf{Error}_{\mathcal{R}})]$
for all *pairs* $\{R_1, R_2\} \subseteq \mathcal{R}$ **do**
  $\mathcal{R}' \leftarrow \{R_1, R_2\}$
  while $|\mathcal{R}'| \leq n$ ***and*** $\mathcal{R}' \models \phi(\mathcal{R}')$ **do**
    $\sigma \leftarrow$ witness of $\phi(\mathcal{R}')$ `// σ witnesses rt-inconsistency of R'`
    if $\exists R \in \mathcal{R} \setminus \mathcal{R}'$ *s.t.* σ **fails** R **then**
      $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{R\}$
    else
      return σ `// σ witnesses rt-inconsistency of R`
return $\emptyset$ `// no witness for the rt-inconsistency of R is found`

Algorithm 1: Incremental rt-consistency checking algorithm. In order to avoid checking the same subsets of $\mathcal{R}'$ several times, one can store the subsets seen so far and break the while loop when $\mathcal{R}'$ has already been treated.

Such a finite trace σ is called a *witness* for the rt-inconsistency of $\mathcal{R}$. Remember that, by Lemma 1, this can be written in CTL as $\mathbf{E}[\neg\mathbf{Error}_{\mathcal{R}}\ \mathbf{U}\ (\neg\mathbf{Error}_{\mathcal{R}} \wedge \mathbf{AX}\ \mathbf{Error}_{\mathcal{R}})]$ to be checked in $\otimes\mathcal{R}$.

When the size of $\mathcal{R}$ is too large for model-checking tools to handle, we consider subsets $\mathcal{R}'$ of $\mathcal{R}$. Such incomplete checks alone do not provide any guarantee; indeed if $\mathcal{R}' \subseteq \mathcal{R}$, consistency of $\mathcal{R}$ does not imply consistency of $\mathcal{R}'$, nor the opposite. Nevertheless, they can be used to detect rt-inconsistencies with an additional check:

Lemma 3. *Let $\sigma \in \Sigma^*$ be a witness for the rt-inconsistency of $\mathcal{R}' \subseteq \mathcal{R}$. If $\neg(\sigma$ **fails** $\mathcal{R})$, then σ is also a witness for the rt-inconsistency of $\mathcal{R}$.*

Let us now describe our procedure summarized in Algorithm 1. Given $\mathcal{R}$ and a bound $n \leq |\mathcal{R}|$, we consider subsets of $\mathcal{R}$ of size up to n, starting with subsets of size 2. Assume a subset $\mathcal{R}' \subseteq \mathcal{R}$ is found to be rt-inconsistent with a witness trace σ. We check whether σ **fails** $\mathcal{R} \setminus \mathcal{R}'$. If this is the case, we select $R \in \mathcal{R} \setminus \mathcal{R}'$ such that σ **fails** R, and restart the analysis with $\mathcal{R}' \cup \{R\}$. Notice that if $\mathcal{R}' \cup \{R\}$ is inconsistent, then σ cannot be a witness trace since it violates R. This ensures that a new requirement will be added to the set at each iteration. Otherwise, by Lemma 3, we conclude that $\mathcal{R}$ is rt-inconsistent and σ is a witness. If no confirmed witnesses are found, then we stop and report that no rt-inconsistency is found. If $n \geq |\mathcal{R}|$, then one can conclude that $\mathcal{R}$ is rt-consistent; otherwise the check is incomplete.

To increase the precision (to have a better chance to detect rt-inconsistencies), one can increase the bound n. In order to reduce the number of cases to check, thus giving up on completeness, one might restrict only to some subsets, for instance making sure that each requirement is covered by at least one subset.

3.2 Incremental partial consistency checking

We now present an incremental algorithm for checking partial consistency via the *bounded* partial consistency checking in the same vein as the previous section.

Input: A set $\mathcal{R}$ of requirements given as CDTAs, parameters $\alpha, \beta > 0$
for all *pairs* $\{R_1, R_2\} \subseteq \mathcal{R}$ **do**
  $\mathcal{R}' \leftarrow \emptyset$
  while *Equation* (2) *fails* **do**
    $(\sigma_1, \sigma_2) \leftarrow$ witness traces for the premise of (2) for some $k \leq \alpha$
    if $\exists i \in \{1, 2\}, \neg(\sigma_i$ **fails** $\mathcal{R})$ **then**
      return σ_i `// witness of rt-inconsistency of` $\mathcal{R}$
    else
      if $\mathcal{R} = \mathcal{R}' \cup \{R_1, R_2\}$ **then**
        break `// No witness is found for this pair`
      else
        Choose $R \in \mathcal{R}$ such that σ_i **fails** R for some $i \in \{1, 2\}$
        $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{R\}$
return $\emptyset$ `// no counterexample is found`

Algorithm 2: Incremental partial consistency checking algorithm.

Ideally, we would like to check Equation(2) for all pairs $\{R_1, R_2\}$ of requirements with respect to $\mathcal{R}' = \mathcal{R} \setminus \{R_1, R_2\}$; in fact, considering the whole set $\mathcal{R}'$ makes sure that counterexample traces do not trivially violate requirements. This is costly in general, so we will start with an empty $\mathcal{R}'$ and let it grow incrementally by adding requirements as needed. The following lemma exhibits when such counterexamples can be lifted to witnesses of rt-inconsistency:

Lemma 4. *Let σ_1, σ_2 and k be witnesses of bounded partial inconsistency for $R_1, R_2 \in \mathcal{R}$ and $\mathcal{R}' \subseteq \mathcal{R}$, i.e. counterexamples of Equation 2. If, for some i, $\neg(\sigma_i$ **fails** $\mathcal{R})$, then σ_i is also a witness for the rt-inconsistency of $\mathcal{R}$.*

The procedure is summarized in Algorithm 2. Given pair (R_1, R_2) and set $\mathcal{R}' \subseteq \mathcal{R} \setminus \{R_1, R_2\}$, integer parameters $\alpha, \beta > 0$, checking the (α, β)-bounded partial-consistency consists in verifying Equation (2). A negative check is witnessed by some $k \leq \alpha$ and a pair of traces σ_1, σ_2. If $\neg(\sigma_i$ **fails** $\mathcal{R})$ holds for some $i \in \{1, 2\}$, the trace is returned as a counterexample by Lemma 4. Otherwise, a requirement $R \in \mathcal{R}$ such that σ_i **fails** R is added to the set $\mathcal{R}'$ and the procedure is repeated. Thus, subsequent iterations will discard σ_i and look for other traces. The following lemma shows that all counterexamples returned by Algorithm 2 are witnesses to rt-inconsistency:

Lemma 5. *Let $\mathcal{R}$ be a set of requirements, and σ be a finite trace returned by Algorithm 2. Then σ is a witness for rt-inconsistency for $\mathcal{R}$.*

3.3 Incremental partial rt-consistency checking

We now propose an algorithm for rt-consistency checking, that combines an incremental approach targeting subsets of requirements (hence the name partial), and a bounded search, providing an alternative to Algorithm 1 amenable to using SMT solvers. Intuitively, we check for the existence of configurations where all

Input: A set $\mathcal{R}$ of requirements, parameters $\alpha > 0$, $n \in [1, |\mathcal{R}|]$
for all *subsets* $\mathcal{S} \subseteq \mathcal{R}$ *such that* $|\mathcal{S}| \leq n$ **do**
  $\mathcal{R}' \leftarrow \emptyset$
  while $\mathcal{S} \times \mathcal{R}' \models \phi_{p,\alpha}$ **do**
    $\sigma \leftarrow$ witness trace for $\phi_{p,\alpha}$
    if $\neg(\sigma$ **fails** $\mathcal{R})$ **then**
      return σ `// Counterexample for` $\mathcal{R}$
    else
      if $\mathcal{R} = \mathcal{R}' \cup \mathcal{S}$ **then**
        break `// No counterexample is found for this subset`
      else
        Choose $R \in \mathcal{R}$ such that σ **fails** R
        $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{R\}$
return $\emptyset$ `// no counterexample is found`

Algorithm 3: Incremental partial rt-consistency checking algorithm.

requirements in a subset $\mathcal{S}$ of $\mathcal{R}$ *immediately conflict i.e.* $\mathbf{AX}\ \mathbf{Error}_{\mathcal{S}}$, meaning that at the next step they inevitably violate at least one requirement of $\mathcal{S}$.

Let $\mathcal{S}$ be a subset of requirements of $\mathcal{R}$. We say that $\mathcal{S}$ is *partially rt-consistent* with respect to $\mathcal{R}'$ if for all configurations s,

$$s \models \neg\mathbf{Error}_{\mathcal{S}\cup\mathcal{R}'} \implies \neg\mathbf{AX}\ \mathbf{Error}_{\mathcal{S}}. \tag{3}$$

This clearly implies that $\mathcal{S}$ is rt-consistent, but also that no immediate conflict affects the subset $\mathcal{S}$ in any configuration. A witness of partial rt-inconsistency is a trace σ that reaches a configuration s satisfying $\neg\mathbf{Error}_{\mathcal{S}\cup\mathcal{R}'} \wedge \mathbf{AX}\ \mathbf{Error}_{\mathcal{S}}$. Since $\mathbf{AX}\ \mathbf{Error}_{\mathcal{S}}$ implies $\mathbf{AX}\ \mathbf{Error}_{\mathcal{R}}$ (because $\mathbf{Error}_{\mathcal{S}}$ implies $\mathbf{Error}_{\mathcal{R}}$), if additionally $\neg(\sigma$ **fails** $\mathcal{R})$ it is also a witness of rt-inconsistency by Lemma 3. Similarly to Lemma 1, the existence of a witness of partial inconsistency reduces to checking the formula $\phi_p = \mathbf{E}(\neg\mathbf{Error}_{\mathcal{S}\cup\mathcal{R}'}\ \mathbf{U}\ (\neg\mathbf{Error}_{\mathcal{S}\cup\mathcal{R}'} \wedge \mathbf{AX}\ \mathbf{Error}_{\mathcal{S}}))$.

Partial rt-consistency can be further restricted by bounding the size of $\mathcal{S}$ and restricting the exploration depth. For integers n and α, we say that $\mathcal{R}$ is *α-bounded n-partially rt-consistent* if Formula 3 holds for any subset $\mathcal{S}$ of size $|\mathcal{S}| \leq n$, and configurations $s \in \mathbf{reach}_{\alpha}(\mathcal{R})$. Checking *$\alpha$-bounded n-partial rt-inconsistency* can be done by replacing $\mathbf{U}$ by $\mathbf{U}_{\alpha}$ in ϕ_p thus checking $\phi_{p,\alpha} = \mathbf{E}(\neg\mathbf{Error}_{\mathcal{S}\cup\mathcal{R}'}\ \mathbf{U}_{\alpha}\ (\neg\mathbf{Error}_{\mathcal{S}\cup\mathcal{R}'} \wedge \mathbf{AX}\ \mathbf{Error}_{\mathcal{S}}))$.

We summarize the procedure in Algorithm 3, where, similarly to Algorithm 2, the set $\mathcal{R}'$ is augmented by requirements failed by tentative counterexamples. We easily get the following lemma since a witness of α-bounded n-partial rt-inconsistency that does not fail $\mathcal{R}$ is also a witness of rt-inconsistency.

Lemma 6. *Let $\mathcal{R}$ be a set of requirements, and σ be a finite trace returned by Algorithm 3. Then σ is a witness for rt-inconsistency.*

set	size	rt-consistency Algorithm 1	partial consistency Algorithm 2	partial rt-consistency Algorithm 3
#1	6 + 9	5 inconsist. (24s)	4 inconsist. (36s)	5 inconsist. (39s)
#2	8 + 10	1 inconsist. (21s)	✓(55s)	1 inconsist. (101s)
#3	8 + 10	✓(24s)	✓(61s)	✓(115s)
#4	10 + 16	✓(359s)	✓(85s)	✓(141s)
#5	12 + 16	✓(1143s)	✓(133s)	✓(227s)
#6	13 + 16	✓(5311s)	✓(138s)	✓(232s)

Table 1. Experiments on our case study. The size shows the number of timed requirements + the number of (non-timed) Boolean requirements of the instance. The parameters were chosen as $\alpha = 40$ and $n = 2$. The sign ✓means that no inconsistencies were found. The experiments were run on a 1.9Ghz processor with a timeout of 3 hours.

4 Preliminary Experiments

We experimented the different algorithms on a factory automation use case. In this system, a carriage and an arm cooperate to convey material: objects are pushed onto the carriage, which brings them to a position where a pushing arm places them on a conveyor belt. The correctness of this system relies on several timed requirements between different elements of the system.

Table 1 shows the inconsistencies found with our algorithms on sets of requirements of varying sizes. The largest set we considered contained 29 requirements of which 13 are timed and the other 16 are purely Boolean requirements. We compare the incremental partial consistency and partial rt-consistency algorithms (implemented using the SMT solver Z3 [Z3]), with the incremental rt-consistency algorithm (implementing CTL model-checking using NuSMV [NuS]). Inconsistencies were detected in the first two sets, but partial consistency failed in detecting any in set #2.

These preliminary experiments show that the incremental method can help detect inconsistencies quickly. However, since the methods are not complete, we encourage using several algorithms in parallel.

5 Conclusion

In this paper, we studied the notions of rt-consistency and partial consistency. We showed how to reduce the problem to CTL model checking on timed automata models, and presented algorithms that can detect rt-inconsistencies. Our preliminary experiments show encouraging results. As future work, we will extensively evaluate the ability of these algorithms to capture inconsistencies, and their performances on large realistic use cases. One might investigate other variants of the (partial) consistency notions, with the goal of detecting more inconsistencies more efficiently. There is a trade-off to find for such partial consistency algorithms. In fact, they might allow one to examine more potential counterexample witnesses, which means that one might detect more inconsistencies, but one might also have to deal with more false positives. Another interesting question is how to correct rt-inconsistencies e.g. by adding new requirements.

References

[AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In Mike Paterson, editor, *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, UK, July 16-20, 1990, Proceedings*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer, 1990.

[AHL+17] Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Ničković, and Stefan Tiran. Require, test, and trace it. *International Journal on Software Tools for Technology Transfer*, 19(4):409–426, Aug 2017.

[BCN+18] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto L. Sangiovanni-Vincentelli, Werner Damm, Thomas A. Henzinger, and Kim G. Larsen. Contracts for system design. *Foundations and Trends in Electronic Design Automation*, 12(2-3):124–400, 2018.

[Bec19] Jan Steffen Becker. Analyzing consistency of formal requirements. *Electronic Communications of the EASST (AVOCS 2018)*, 76, 2019.

[BTES16] Tom Bienmüller, Tino Teige, Andreas Eggers, and Matthias Stasch. Modeling requirements for quantitative consistency analysis and automatic test case generation. In *Workshop on Formal and Model-Driven Techniques for Developing Trustworthy Systems at 18th International Conference on Formal Engineering Methods*, 2016.

[ESH14] Christian Ellen, Sven Sieverding, and Hardi Hungar. Detecting consistencies and inconsistencies of pattern-based functional requirements. In Frédéric Lang and Francesco Flammini, editors, *Formal Methods for Industrial Critical Systems*, pages 155–169, Cham, 2014. Springer International Publishing.

[Hoe06] Jochen Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, 2006.

[JMM+20] Thierry Jéron, Nicolas Markey, David Mentré, Reiya Noguchi, and Ocan Sankur. Incremental methods for checking real-time consistency, 2020.

[NuS] NuSMV: a new symbolic model checker. `http://nusmv.fbk.eu/`.

[PHP11a] Amalinda Post, Jochen Hoenicke, and Andreas Podelski. rt-inconsistency: a new property for real-time requirements. In *Fundamental Approaches to Software Engineering (FASE)*, volume 6603 of *LNCS*. Springer, 2011.

[PHP11b] Amalinda Post, Jochen Hoenicke, and Andreas Podelski. Vacuous real-time requirements. In *IEEE 19th International Requirements Engineering Conference*, pages 153–162, Aug 2011.

[TBH16] Tino Teige, Tom Bienmüller, and Hans Jürgen Holberg. Universal pattern: Formalization, testing, coverage, verification, and test case generation for safety-critical requirements. In Ralf Wimmer, editor, *19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'16)*, pages 6–9. Albert-Ludwigs-Universität Freiburg, 2016.

[Z3] The Z3 theorem prover. `https://github.com/Z3Prover/z3`.

[ZHR91] Chaochen Zhou, C.A.R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters (IPL)*, 40(5):269–276, 1991.

A Proofs

Lemma 1. *A given set of requirements $\mathcal{R}$ has a witness to rt-inconsistency if, and only if, $\mathcal{R} \models \mathbf{E}(\neg\mathbf{Error}_{\mathcal{R}}\ \mathbf{U}\ (\neg\mathbf{Error}_{\mathcal{R}} \wedge \mathbf{AX}\ \mathbf{Error}_{\mathcal{R}}))$.*

Proof. Let us consider the following formula

$$\phi(\mathcal{R}) = \mathbf{E}(\neg\mathbf{Error}_{\mathcal{R}}\ \mathbf{U}\ (\neg\mathbf{Error}_{\mathcal{R}}\ \wedge\ \mathbf{AX}\ \mathbf{Error}_{\mathcal{R}}))$$

By definition, $\mathcal{R}$ is rt-inconsistent if there is a reachable configuration s such that $s \models \neg\mathbf{Error}_{\mathcal{R}} \wedge \mathbf{AF}\ \mathbf{Error}_{\mathcal{R}}$. It is thus clear that if the initial state of $\otimes\mathcal{R}$ satisfies $\phi(\mathcal{R})$, then $\mathcal{R}$ is rt-inconsistent.

Let us assume that $\mathcal{R}$ is rt-inconsistent, and consider a reachable configuration s satisfying $\neg\mathbf{Error}_{\mathcal{R}} \wedge \mathbf{AF}\ \mathbf{Error}_{\mathcal{R}}$. Let ρ denote the run that ends in s. Since $\mathbf{Error}_{\mathcal{R}}$ is absorbing, all states of ρ satisfy $\neg\mathbf{Error}_{\mathcal{R}}$. We show that there exists some configuration s' reachable from s with both $s' \models \neg\mathbf{Error}_{\mathcal{R}}$ and $s' \models \mathbf{AX}\ \mathbf{Error}_{\mathcal{R}}$. To see this, we build a run from s inductively as follows. Initially, the run is at configuration s. At any moment, if the current configuration has a successor satisyfing $\neg\mathbf{Error}_{\mathcal{R}}$, we choose one arbitrarily and extend the run. If there are no such successors, then this provides the configuration s' as desired. Notice that this constructed run cannot be infinite, since this would contradict that $s \models \mathbf{AF}\ \mathbf{Error}_{\mathcal{R}}$, so such a s' must exist.

Now the run we obtain from the initial configuration to s' is a witness for $\phi(\mathcal{R})$. □

Lemma 2. *If R_1 and R_2 are partially inconsistent w.r.t. $\mathcal{R}'$, then $\mathcal{R}' \cup \{R_1, R_2\}$ is rt-inconsistent.*

Proof. Consider $k \geq 0$, and traces σ_1, σ_2 which are witnesses to partial inconsistency, as well as configurations $s_i \in \mathbf{reach}_k(\mathcal{R}_1 \times \mathcal{R}_2 \times \mathcal{R}')$. We have $s_1 \models \neg\mathbf{Error}_{\mathcal{R}_1} \wedge \neg\mathbf{Error}_{\mathcal{R}_2} \wedge \neg\mathbf{Error}_{\mathcal{R}'}$. Rt-consistency requires that there exists an infinite continuation from s_1 satisfying $\neg\mathbf{Error}_{\mathcal{R}_1} \wedge \neg\mathbf{Error}_{\mathcal{R}_2} \wedge \neg\mathbf{Error}_{\mathcal{R}'}$. However, since (1) does not hold, there is no state $s \in \mathbf{reach}_k(\mathcal{R}_1 \times \mathcal{R}_2 \times \mathcal{R}')$ satisfying both $s \models \mathbf{EX}\ \mathbf{Error}_{\mathcal{R}_1} \wedge \mathbf{EX}\ \mathbf{Error}_{\mathcal{R}_2}$ and admitting such an infinite continuation. Therefore, s_i cannot have such a continuation, which proves that $\mathcal{R}' \cup \{R_1, R_2\}$ is rt-inconsistent. □

Lemma 3. *Let $\sigma \in \Sigma^*$ be a witness for the rt-inconsistency of $\mathcal{R}' \subseteq \mathcal{R}$. If $\neg(\sigma\ \mathbf{fails}\ \mathcal{R})$, then σ is also a witness for the rt-inconsistency of $\mathcal{R}$.*

Proof.

In fact, if σ is a witness of rt-inconsistency in $\mathcal{R}'$, by definition $\neg(\sigma\ \mathbf{fails}\ \mathcal{R}')$ but $\sigma\ \mathbf{I\text{-}fails}\ \mathcal{R}'$. Since $\mathcal{R}' \subseteq \mathcal{R}$, (inevitably) failing $\mathcal{R}$ implies (inevitably) failing $\mathcal{R}'$ ($\sigma\ \mathbf{I\text{-}fails}\ \mathcal{R}'$ implies $\sigma\ \mathbf{I\text{-}fails}\ \mathcal{R}$). By hypothesis, $\neg(\sigma\ \mathbf{fails}\ \mathcal{R}')$, but it may be the case that $\sigma; \mathbf{fails}\ \mathcal{R}$. If additionnally $\neg(\sigma\ \mathbf{fails}\ \mathcal{R})$, then we can conclude that σ is a witness of rt-inconsistency of $\mathcal{R}$. □

Lemma 4. *Let σ_1, σ_2 and k be witnesses of bounded partial inconsistency for $R_1, R_2 \in \mathcal{R}$ and $\mathcal{R}' \subseteq \mathcal{R}$, i.e. counterexamples of Equation 2. If, for some i, $\neg(\sigma_i$ **fails** $\mathcal{R})$, then σ_i is also a witness for the rt-inconsistency of $\mathcal{R}$.*

Proof. For any $i \in \{1, 2\}$, if σ_i witnesses (α, β)-bounded partial inconsistency, by definition σ_i reaches a configuration s_i in $reach_k(\mathcal{R}_1 \times \mathcal{R}_2 \times \mathcal{R}')$ satisfying $\mathbf{EX\ Error}_{R_1} \wedge \mathbf{EX\ Error}_{R_2} \wedge \neg\mathbf{AF}_{\alpha-k}(\mathbf{Error}_{\mathcal{R}'} \vee \mathbf{Error}_{R_i})$ but no configuration $s \in reach_k(\mathcal{R}_1 \times \mathcal{R}_2 \times \mathcal{R}')$ satisfies $\mathbf{EX\ Error}_{R_1} \wedge \mathbf{EX\ Error}_{R_2} \wedge \neg\mathbf{AF}_{\alpha+\beta-k}(\mathbf{Error}_{\mathcal{R}'} \vee \mathbf{Error}_{R_1} \vee \mathbf{Error}_{R_2})$. Since $s_i \models \mathbf{EX\ Error}_{R_1} \wedge \mathbf{EX\ Error}_{R_2}$, it satisfies $\mathbf{AF}_{\alpha+\beta-k}(\mathbf{Error}_{\mathcal{R}'} \vee \mathbf{Error}_{R_1} \vee \mathbf{Error}_{R_2})$. If additionally σ_i satisfies $\neg(\sigma_i$ **fails** $\mathcal{R})$, since $\mathbf{AF}_{\alpha+\beta-k}(\mathbf{Error}_{R'} \vee \mathbf{Error}_{R_1} \vee \mathbf{Error}_{R_2})$ implies $\mathbf{AF\ Error}_{\mathcal{R}}$, then s_i satisfies $\neg\mathbf{Error}_{\mathcal{R}} \wedge \mathbf{AF\ Error}_{\mathcal{R}}$, thus σ_i is a witness for rt-inconsistency. □

Lemma 5. *Let $\mathcal{R}$ be a set of requirements, and σ be a finite trace returned by Algorithm 2. Then σ is a witness for rt-inconsistency for $\mathcal{R}$.*

Proof. Assume that the algorithm returned a counterexample trace $\sigma \in \Sigma^*$ for the outer iteration with $R_1, R_2 \in \mathcal{R}$, and inner iteration $\mathcal{R}' \subseteq \mathcal{R}$. The algorithm ensures that $\neg(\sigma$ **fails** $\mathcal{R})$ (line 6). We then use Lemma 4 to conclude that σ is a witness for rt-inconsistency for $\mathcal{R}$. □