Repairing Real-Time Requirements*

Reiya Noguchi[1,2], Ocan Sankur[3]
Thierry Jéron[3], Nicolas Markey[3], and David Mentré[2]

[1] Mitsubishi Electric Corporation, Tokyo (Japan)
lastname.firstname@ah.MitsubishiElectric.co.jp
[2] Mitsubishi Electric R&D Centre Europe, Rennes (France)
initial-of-firstname.lastname@fr.merce.mee.com
[3] Univ Rennes, Inria, CNRS, Rennes (France)
firstname.lastname@inria.fr

Abstract. We consider the problem of repairing inconsistent real-time requirements with respect to two consistency notions: non-vacuity, which means that each requirement can be realized without violating other ones, and rt-consistency, which means that inevitable violations are detected immediately. We provide an iterative algorithm, based on solving SMT queries, to replace designated parameters of real-time requirements with new Boolean expressions and time constraints, so that the resulting set of requirements becomes consistent.

1 Introduction

Requirements play an important role in the design of real-time systems. These allow one to specify desired properties for the system under development at an early stage, and can be used to guide testing and formal verification [26]. While basic requirements focus on the relation between the inputs and outputs of the system, extrafunctional properties such as timing constraints are crucial for describing the behaviors of real-time systems.

It is thus important to design formal requirements that are *consistent*, that is, that avoid contradictions and admit implementations. Several works have focused on providing tools to define, combine, and study specifications [8]; others have defined various notions of *consistency*, *e.g.* [14, 27, 1, 28], which are used to detect conflictual requirements that are impossible to satisfy in an implementation according to given criteria.

While several works have focused on checking the consistency of requirement sets, or applying formal verification on requirements, we are interested in *repairing* a given requirement set that is inconsistent, in order to turn it into a consistent set. Repairing an unsatisfactory model or program is an active research area. It consists in building expressions that fit a given data set to fill unknown expressions in programs. Various techniques such as constraint solving, decision tree learning, or search algorithms are used for repairing programs [3, 16]. We believe that

* This work was partially funded by ANR project Ticktac (ANR-18-CE40-0015).

requirements are a good target for repair algorithms as they can assist the user in correcting unsatisfactory requirement sets in an early stage. In this paper, we provide repair algorithms tailored for the consistency of real-time requirements.

We consider two consistency notions from the literature. The first one is the *non-vacuity* of a requirement set, studied in temporal logic model checking [22] but also in requirement verification [28]. This line of work was inspired by the observation that formulas of the form $a \rightarrow b$ might hold in a given model simply because a never becomes true. Thus, such a formula is *vacuously* satisfied, which indicates an error, either in the design of the model or in the specification. Intuitively, when all requirements are such implications, a requirement set is *non-vacuous* if the premise of each requirement is satisfied by some execution which does not fail the other requirements.

We consider requirements expressed as Simplified Universal Patterns (SUPs for short) [9, 29], which are patterns defining real-time temporal properties, and are in the form of a logical implication with time constraints: in each requirement, completing a given *trigger* phase implies the realization of a corresponding *action* phase. Moreover, the action phase must start after a given time interval following the trigger, and phases are given durations with time intervals. Due to this form, non-vacuity is easy to define and to interpret: the trigger phase of each requirement must be realized by some execution which does not fail other requirements. SUPs can be expressed as timed automata, and our algorithms can be easily extended to general timed automata [2] as in [18]. We do focus on SUPs here for their simplicity, and because non-vacuity can be defined naturally due to their form. They are expressive enough to write complex specifications, including the benchmarks we consider in Section 4.

The second consistency notion we consider is *rt-consistency* [27]. This requires that all finite executions that satisfy all requirements (*i.e.*, do not violate any of them) admit infinite extensions that still satisfy all the requirements. Put differently, this means that if an implementation produces a finite execution whose all continuations necessarily lead to the failure of some requirement, then there must be a requirement that already fails at the said finite execution: the inevitability of an error must be anticipated by the set of requirements. It can be shown that rt-consistency is not a linear-time property; it was expressed using a CTL formula in [18]. It can be observed that adding a requirement to the set can remove rt-inconsistencies, since, intuitively, the new requirement can be made to imediately fail whenever the error is inevitable in the future. However, this must be done with care since adding a requirement might also introduce new rt-inconsistencies and render some other requirements vacuous.

Our main result is an algorithm that, given a requirement set and some designated parameter set M (time constraints and/or Boolean expressions that appear in requirements), attempts to compute new values for the parameters in M such that the new requirement set is rt-consistent and non-vacuous. Our algorithm is iterative: at each iteration, we solve an SMT query to compute candidate values for the parameters, and check whether non-vacuity and rt-consistency hold. When this is not the case, we derive a new constraint to add to the SMT

query and start again. The new constraint either forces one of the requirements to be satisfied non-vacuously, or it excludes a counterexample to rt-consistency.

We apply our algorithm to several benchmarks including four case studies that have appeared in the literature, and anonymized benchmarks from [23]. In each case, we considered manually-introduced rt-inconsistencies and focused on two uses: repairing the requirement set by adding a fresh requirement; and repairing the set by modifying the parameters of a designated requirement.

Related Works. Verification algorithms for non-vacuity and rt-consistency were given in [23] based on a reduction of the problems to a safety verification problem, and using a software model checker. Due to efficiency constraints, the presented results are obtained using a partial check: the rt-consistency is checked only for pairs of real-time requirements; nonetheless, the method can also be applied to consider the whole set.

Our approach is similar to program repair [3, 16] where some techniques are also based on using solvers to find expressions subject to given constraints. The main difference of such lines of work with ours is that correctness is defined based on non-vacuity and rt-consistency rather than on the acceptance of given test cases, or on the model checking of the program w.r.t. a specification.

Repairing real-time systems has been considered recently. In [19, 20], the authors provide an iterative algorithm that finds a timed diagnostic trace in a timed automaton using a model checker, and use an SMT solver to compute modifications in the guards of the automaton. To ensure that the new automaton is satisfactory, they check for untimed language equivalence (which is EXPSPACE-complete [13]). Their tool enumerates all possible repairs until one passes this equivalence test. In [4], the authors use parameter synthesis to find new values of guards and validate with testing. Guard relaxation for ensuring a reachability property is studied in [7].

Several algorithms for temporal logics rely on a given labeling of input signals: the goal is to compute parameter values so as to reject some set of inputs signals, and accept some others; see [21, 11, 15] for signal temporal logic. The problem of synthesizing parameters for metric temporal logic formulas for a given hybrid system was studied in [30]; see also [5] for a statistical learning procedure. In [25], the goal is to compute a formula that accepts a given set of positive traces, and rejects given negative traces. The algorithm also uses a SAT solver to guess the formula as a DAG of size n, and increases n until a solution is found. In our case, we restrict to requirements with propositional formulas in conjunctive form, which simplifies their encodings.

2 Preliminaries

Traces. We fix a set AP of atomic propositions that represent Boolean inputs and outputs of the system. A *valuation* of AP is a mapping $v_{\mathsf{AP}} \colon \mathsf{AP} \to \{\top, \bot\}$ (or equivalently an element of 2^{AP}). We write $\mathcal{B}(\mathsf{AP})$ for the set of Boolean combinations of atomic propositions in AP. That a valuation v_{AP} satisfies a formula $\phi \in \mathcal{B}(\mathsf{AP})$, denoted by $v_{\mathsf{AP}} \models \phi$, is defined in the usual way.

A (finite) *trace* σ is a sequence of valuations, and its length is denoted by $|\sigma|$. Traces are seen as elements of $(2^{\mathsf{AP}})^*$. The *prefix of length* i of the trace $\sigma = \sigma_1\sigma_2\ldots\sigma_n$ is denoted by $\sigma_{1\ldots i} = \sigma_1\sigma_2\ldots\sigma_i$.

Timed automata. We use timed automata (here with a discrete-time semantics) to model and reason about timed requirements.

Let $\mathcal{X} = \{c_i \mid 1 \leq i \leq k\}$ be a set of variables called *clocks*. We consider integer-valued clocks. For a valuation $v_{\mathcal{X}} \colon \mathcal{X} \to \mathbb{N}$ (equivalently an element of $\mathbb{N}^{\mathcal{X}}$), an integer $d \in \mathbb{N}$, and a subset of clocks $R \subseteq \mathcal{X}$, we define $v_{\mathcal{X}} + d$ as the valuation $(v_{\mathcal{X}} + d)(c) = v_{\mathcal{X}}(c) + d$ for all $c \in \mathcal{X}$, and $v_{\mathcal{X}}[R \leftarrow 0]$ as $v_{\mathcal{X}}[R \leftarrow 0](c) = 0$ if $c \in R$, and $v_{\mathcal{X}}[R \leftarrow 0](c) = v_{\mathcal{X}}(c)$ otherwise. Let $\mathbf{0}$ be the valuation mapping all variables to 0.

The set of *clock constraints* over $\mathcal{X}$ 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 over $\mathcal{X}$. The semantics of clock constraints is defined in the expected way: given a clock valuation $v_{\mathcal{X}} \colon \mathcal{X} \to \mathbb{N}$, a constraint $g \in \mathcal{C}(\mathcal{X})$ is true at $v_{\mathcal{X}}$, denoted $v_{\mathcal{X}} \models g$, if the formula obtained by replacing each occurrence of c in g by $v_{\mathcal{X}}(c)$ holds.

We consider timed automata over the alphabet 2^{AP} of valuations of AP, thereby generating (discrete-time) traces. Transitions are labelled with Boolean constraints on AP.

A *timed automaton* (TA) is a tuple $\mathcal{A} = \langle S, s_0, \mathsf{AP}, \mathcal{X}, T, F\rangle$ where S is a finite set of states, $s_0 \in S$ is the initial state, AP is a finite set of atomic propositions, $\mathcal{X}$ is a finite set of clocks, $T \subseteq S \times \mathcal{B}(\mathsf{AP}) \times \mathcal{C}(\mathcal{X}) \times 2^{\mathcal{X}} \times S$ is a finite set of transitions, and $F \subseteq S$ is the set of accepting states.

We endow timed automata with a discrete-time semantics, as follows. With a timed automaton $\mathcal{A}$, we define the infinite-state automaton $\mathcal{S}(\mathcal{A}) = \langle Q, q_0, D, Q_F\rangle$ over 2^{AP} where $Q = S \times \mathbb{N}^{\mathcal{X}}$, $q_0 = (s_0, \mathbf{0})$, $Q_F = F \times \mathbb{N}^{\mathcal{X}}$ is the set of accepting configurations, and transitions in D are combinations of a transition of the TA and a one-time-unit delay. Formally, given a valuation $v_{\mathsf{AP}} \in 2^{\mathsf{AP}}$ and two configurations $(s, v_{\mathcal{X}})$ and $(s', v'_{\mathcal{X}})$, there is a transition $((s, v_{\mathcal{X}}), v_{\mathsf{AP}}, (s', v'_{\mathcal{X}}))$ in D if, and only if, there is a transition (s, ϕ, g, r, s') in T such that $v_{\mathsf{AP}} \models \phi$ and $v_{\mathcal{X}} \models g$, and $v'_{\mathcal{X}} = (v_{\mathcal{X}}[r \leftarrow 0]) + 1$;

Our semantics thus makes it compulsory to take a transition of the TA (possibly a self-loop) at each time unit. This can be used to emulate invariants in states. The automaton $\mathcal{S}(\mathcal{A})$ can be rendered finite by bounding the clocks since the exact values of clock variables above a threshold do not matter (see [2]).

A *run* of $\mathcal{A}$ is a run of its associated infinite-state automaton $\mathcal{S}(\mathcal{A})$. It can be represented as a sequence along which configurations and actions alternate: $(s_0, v_0) \cdot \sigma_1 \cdot (s_1, v_1) \cdot \sigma_2 \cdots (s_n, v_n) \cdots$. A finite run is *accepting* if it ends in Q_F. A trace $\sigma = (\sigma_i)_{1 \leq i \leq n}$ is accepted by $\mathcal{A}$ if there is an accepting run $(s_0, v_0) \cdot \sigma_1 \cdot (s_1, v_1) \cdot \sigma_2 \cdots (s_n, v_n)$ in $\mathcal{A}$.

We only consider *safety* TAs, *i.e.*, TAs in which there are no transitions from $S \setminus F$ to F. Under such a condition, a run is accepting if, and only if, it never visits any non-accepting state. This simplifies the presentation but a richer set of properties could be handled as in [18].

Simplified Universal Patterns. Simplified Universal Patterns (SUPs) [9, 29] are a simple and convenient formalism for expressing requirements. They are more intuitive, but less expressive, than TAs.

An SUP requirement has the following form:

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

where $\mathcal{F}_b = \{\text{TSE}, \text{TC}, \text{TEE}, \text{ASE}, \text{AC}, \text{AEE}\}$ is the set of *Boolean parameters* (See Figure 1 for the meaning of acronyms), which are Boolean formulas on AP, and $\mathcal{F}_t = \{\text{Lmin}, \text{Lmax}, \text{Amin}, \text{Amax}, \text{Tmin}, \text{Tmax}\}$ is the set of *time parameters*, which are integer time bounds. Their union is $\mathcal{F} = \mathcal{F}_b \cup \mathcal{F}_t$. We only consider bounded intervals.

Figure 1 illustrates the intuitive semantics of SUPs. A *trigger phase* (left) is realized, if TSE occurs and 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 *succeed*, 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 AC must hold until AEE occurs within [Amin, Amax] time units. Otherwise, the SUP is *failed*.

The semantics of (generic) SUPs can be encoded using timed automata [6]. These automata are defined over states $Q_{\text{SUP}} = \{\mathsf{init}, \mathsf{trig}, \mathsf{delay}, \mathsf{act}, \mathsf{err}\}$ with err the only state not in F. Intuitively, the execution starts at init, it is at trig if the trigger phase is being checked; at delay if the trigger was realized but the subsequent action has not started yet; at act if the action phase is being checked; from delay or act, either err is reached and the SUP is failed, or init is reached and the SUP succeeds. Note that similar automata definitions were previously given [6].

An SUP instance can be defined as a valuation P of parameters in $\mathcal{F}$, *i.e.*, a valuation of each Boolean parameter of $\mathcal{F}_b$ by a formula in $\mathcal{B}(\mathsf{AP})$, and each time parameter of $\mathcal{F}_t$ by an integer. We then write $\text{SUP}(P)$ for the SUP with parameters defined by P, and $\mathcal{A}_{\text{SUP}(P)}$ for the timed automaton corresponding to $\text{SUP}(P)$. Given such a P and $f \in \mathcal{F}$, P_f refers to the value of the parameter f in P.

The sets of SUP requirements we consider will always be assumed to be indexed, and will be written in the form $(\text{SUP}(P^i))_{1 \leq i \leq n}$. Thus P^i_f will refer

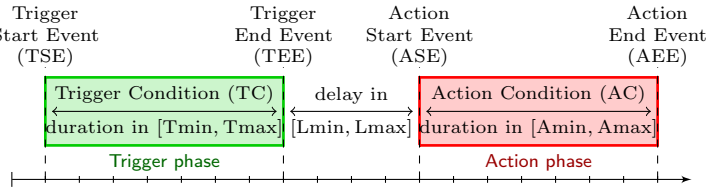

Fig. 1. Intuitive semantics of SUPs

to the value of the parameter f in P^i. We will also consider subsets of *indexed parameters* $\{(f,i) \mid f \in \mathcal{F}, 1 \leq i \leq n\}$ to refer to a subset of the parameters.

Example 1. Consider a flashing light which can blink with a period of 20 time units. The variable `blink` determines whether the blinking mode is active, and `on` indicates that the light is currently on.

$$R_1: \quad (\texttt{on}, \texttt{true}, \texttt{true})[0,0] \xrightarrow{[0,0]} (\texttt{true}, \texttt{on}, \neg\texttt{on})[10,10]$$

$$R_2: \quad (\neg\texttt{on}, \neg\texttt{on}, \neg\texttt{on} \wedge \texttt{blink})[9,9] \xrightarrow{[1,1]} (\texttt{true}, \texttt{true}, \texttt{on})[0,0]$$

R_1 means that when the light turns on, it will remain on for 10 time units, and then turn off. R_2 states that if the light has been off for 9 time units, and the blinking mode is active, then it should turn on at the next time unit.

If we write $R_1 = \mathrm{SUP}(P^1)$ and $R_2 = \mathrm{SUP}(P^2)$, then for instance, P^2_{TSE} is the formula $\neg\texttt{on}$, and $P^1_{\mathrm{Amax}} = 10$.

In the rest of the paper, we only consider timed automata that correspond to SUPs; and the term *requirement* interchangeably refers to an SUP or to its timed-automaton representation.

A trace σ is said to *trigger* an SUP requirement $\mathrm{SUP}(P)$, if the trigger phase is realized by reading σ, that is, if TSE is observed and, within a period in [Tmin, Tmax], TC holds until a point where TEE is true.

A finite or infinite trace σ *satisfies* the SUP if the state err is never reached in $\mathcal{A}_{\mathrm{SUP}(P)}$ by reading σ; this is denoted by $\sigma \models \mathrm{SUP}(P)$. If err is reached, then σ *fails* $\mathrm{SUP}(P)$ and we write $\sigma \not\models \mathrm{SUP}(P)$. For a set of requirements $\mathcal{R} = (\mathrm{SUP}(P^i))_{1 \leq i \leq n}$, we write $\sigma \models \mathcal{R}$ if σ satisfies *all* requirements in $\mathcal{R}$. Symmetrically, we write $\sigma \not\models \mathcal{R}$ if σ fails *at least one of* the requirements in $\mathcal{R}$. Note that since we consider bounded time intervals, when a trace triggers a requirement R, all extensions will eventually either realize the action phase or fail R.

RT-Consistency. We recall rt-consistency, introduced in [27] and further studied in [18]. Put simply, a set $\mathcal{R}$ of requirements is rt-consistent if all finite traces that do not fail $\mathcal{R}$ admit infinite continuations that satisfy $\mathcal{R}$. In other terms, at any finite trace where failure is inevitable, some requirement must already be failed.

For a requirement set $\mathcal{R}$, and trace σ, we write σ **I-fails** $\mathcal{R}$ if for all infinite traces σ', $\sigma \cdot \sigma' \not\models \mathcal{R}$. RT-consistency can then be expressed as follows:

Definition 1 (RT-consistency). *A set $\mathcal{R}$ of requirements is* rt-consistent *if, for any finite trace σ, if σ* **I-fails** *$\mathcal{R}$, then $\sigma \not\models \mathcal{R}$. A* witness to rt-inconsistency *then is a finite trace σ such that σ* **I-fails** *$\mathcal{R}$ and $\sigma \models \mathcal{R}$.*

Thus a witness is a finite trace that satisfies all requirements but whose all infinite continuations fail some of the requirements.

A simpler characterization of rt-inconsistency was proven in [18]:

Theorem 1 ([18]). *A set $\mathcal{R}$ of requirements is* rt-inconsistent *if, and only if there exists a trace σ such that $\sigma \models \mathcal{R}$, and for any valuation $a \in 2^{\mathsf{AP}}$, $\sigma a \not\models \mathcal{R}$.*

Example 2. We consider the requirements R_1 and R_2 from Example 1. We add an atomic proposition `lowBattery`, and consider the new requirement

$$R_3 \colon (\texttt{lowBattery}, \texttt{true}, \texttt{true})[0,0] \xrightarrow{[0,0]} (\texttt{true}, \neg\texttt{on}, \texttt{true})[50,50],$$

which requires to switch off the lights for 50 time units if the battery is detected to be low. The set $\mathcal{R} = \{R_1, R_2, R_3\}$ of requirements is rt-inconsistent. In fact, the finite trace $\sigma = \{\texttt{lowBattery}\} \cdot \emptyset \cdot \ldots \cdot \emptyset \cdot \{\texttt{blink}\}$ of length 9 does not fail any of the requirements, so $\sigma \models \mathcal{R}$. But all extensions of σ fail $\mathcal{R}$. In fact, by R_2, `on` must be true in the next state; while by R_3, `on` must be false. Thus, σ is a witness to the rt-inconsistency of $\mathcal{R}$. One could repair this rt-inconsistency by forcing the value of `blink` to false for 50 time units whenever `lowBattery` is true: $R_4 \colon (\texttt{lowBattery}, \texttt{true}, \texttt{true})[0,0] \xrightarrow{[0,0]} (\texttt{true}, \neg\texttt{blink}, \texttt{true})[50,50]$, so that $\mathcal{R} \cup \{R_4\}$ is rt-consistent.

Non-vacuity. We define the *non-vacuity* of a set of requirements, which states that each requirement must be triggered by some trace without failing any requirement. This notion is closely related to non-vacuity in temporal logic, where an implication of the form $a \rightarrow b$ is said to be satisfied *vacuously* if a is never satisfied in the given system [22]. SUP requirements are similar to implications since the realization of a trigger phase implies the non-violation of the action phase. Intuitively, a requirement that is impossible to trigger points to a bug in the set of requirements, and a good set of requirements must be non-vacuous.

For $R \in \mathcal{R}$, we say that R is *non-vacuous in* $\mathcal{R}$ if there exists a trace that satisfies $\mathcal{R}$ and triggers R; otherwise R is *vacuous* in $\mathcal{R}$.

Definition 2 (Non-vacuity). *A set $\mathcal{R}$ of SUP requirements is* non-vacuous *if for each $R \in \mathcal{R}$, there exists a trace that satisfies $\mathcal{R}$ and triggers R.*

Example 3. Consider again requirements R_1, R_2. Assume that the designer wants to allow a user to maintain the light on manually by pushing a button, but wants blinking to be deactivated if the user has been pushing the button for 20 time units, expressed as

$$R_3' \colon (\texttt{on}, \texttt{on}, \texttt{on})[20,20] \xrightarrow{[0,0]} (\texttt{true}, \texttt{true}, \neg\texttt{blink})[0,0].$$

However, the set $\{R_1, R_2, R_3'\}$ is vacuous: in fact, R_3' can never be triggered since according to R_1, maintaining `on` for 10 time units switches the light off. Thus, R_3' is useless. To fix this issue, the designer can introduce a predicate `button` determining whether the button is being pushed, require R_1, R_2 under condition $\neg\texttt{button}$, and trigger R_3 if $\texttt{button} \wedge \texttt{on}$ has been true for 20 time units.

Conjunctive Formulas and Substitutions. Although we allow the parameters of SUP requirements to be arbitrary Boolean expressions, we will only synthesize parameters that are conjunctive formulas when repairing requirements. We show here how synthesizing a conjunctive formula can be seen as choosing an integer

valuation for a set of fresh variables. This will allow us to use an SMT solver for finding repairs.

Let us fix a requirement set $\mathcal{R} = (\mathrm{SUP}(P^i))_{1 \le i \le n}$, and a subset of *modifiable* indexed parameters $M \subseteq \{(f,i) \mid 1 \le i \le n, f \in \mathcal{F}\}$. For $(f,i) \in M$, define $\mathsf{AP}_{f,i}$ as the set of fresh integer variables $x_{f,i}$ for $x \in \mathsf{AP}$ as follows:

- For $f \in \mathcal{F}_b$, the value of $x_{f,i}$ encodes how x should appear in the conjunctive formula for f in P^i: as a positive literal (1), as a negative literal (-1), or absent (0). We define the *template* for (f,i) as

$$\mathsf{tmp}(f,i) = \wedge_{x \in \mathsf{AP}}([x_{f,i} = 1] \Rightarrow x) \wedge ([x_{f,i} = -1] \Rightarrow \neg x).$$

  A substitution $\xi : \mathsf{AP}_{f,i} \to \{-1, 0, 1\}$ simplifies this formula into a conjunctive formula over AP, so looking for such a conjunctive formula is reduced to looking for a valuation over the variables in $\mathsf{AP}_{f,i}$. The conjunctive formula thus obtained is denoted $\mathsf{tmp}(f,i)[\xi]$. Conversely, any conjunctive formula over AP can be obtained from a template formula by such a substitution.
- For $f \in \mathcal{F}_t$, we define $\mathsf{tmp}(f,i) = x_{f,i}$, and consider substitutions which replace variables $x_{f,i}$ with natural numbers.

Let us define $\mathsf{AP}_M = \bigcup_{\{(f,i) \in M\}} \mathsf{AP}_{f,i}$. Given $\mathcal{R}$ and M, a substitution will refer to a function that is the union of substitutions for all parameters in M (including both timed and Boolean). We denote by $\mathsf{tmp}_M(\mathcal{R})$ the *template requirement set* in which each parameter value P^i_f with $(f,i) \in M$ is replaced with $\mathsf{tmp}(f,i)$; and for a substitution ξ, $\mathsf{tmp}_M(\mathcal{R})[\xi]$ denotes the requirement set obtained by applying the given subtitution to all templates.

Example 4. Consider the following requirements $\mathcal{R} = \{R_1, R_2\}$.

$$R_1: \quad (\mathtt{on}, \mathtt{true}, \mathtt{true})[0, \square] \xrightarrow{[0,0]} (\mathtt{true}, \mathtt{on}, \square)[10, 10]$$

$$R_2: \quad (\neg\mathtt{on}, \neg\mathtt{on}, \square)[9, 9] \xrightarrow{[1,1]} (\mathtt{true}, \mathtt{true}, \mathtt{on})[0, 0]$$

with $\mathsf{AP} = \{\mathtt{on}, \mathtt{blink}\}$, and consider $M = \{(\mathrm{AEE}, 1), (\mathrm{TEE}, 2), (\mathrm{Tmax}, 1)\}$ (*i.e.*, AEE and Tmax in R_1 and TEE in R_2). Placeholders for parameters in M are shown as $\square$. We have, for instance,

$$\begin{aligned}\mathsf{tmp}(\mathrm{AEE}, 1) = {} & ([\mathtt{on}_{\mathrm{AEE},1} = 1] \Rightarrow \mathtt{on}) \wedge ([\mathtt{on}_{\mathrm{AEE},1} = -1] \Rightarrow \neg\mathtt{on}) \\ & \wedge ([\mathtt{blink}_{\mathrm{AEE},1} = 1] \Rightarrow \mathtt{blink}) \wedge ([\mathtt{blink}_{\mathrm{AEE},1} = -1] \Rightarrow \neg\mathtt{blink})\end{aligned}$$

The substitution defined by $\xi(\mathtt{on}_{\mathrm{AEE},1}) = -1$, $\xi(\mathtt{blink}_{\mathrm{AEE},1}) = 0$, $\xi(\mathtt{on}_{\mathrm{TEE},2}) = -1$, $\xi(\mathtt{blink}_{\mathrm{TEE},2}) = 1$, and $\xi(t_{\mathrm{Tmax},1}) = 0$, yields $\mathsf{tmp}(\mathrm{AEE}, 1)[\xi] = \neg\mathtt{on}$ and $\mathsf{tmp}(\mathrm{TEE}, 2)[\xi] = \neg\mathtt{on} \wedge \mathtt{blink}$ and $\mathsf{tmp}(\mathrm{Tmax}, 1)[\xi] = 0$. . Thus, $\mathsf{tmp}_M(\mathcal{R})[\xi]$ is the following:

$$R_1: \quad (\mathtt{on}, \mathtt{true}, \mathtt{true})[0, 0] \xrightarrow{[0,0]} (\mathtt{true}, \mathtt{on}, \neg\mathtt{on})[10, 10]$$

$$R_2: \quad (\neg\mathtt{on}, \neg\mathtt{on}, \neg\mathtt{on} \wedge \mathtt{blink})[9, 9] \xrightarrow{[1,1]} (\mathtt{true}, \mathtt{true}, \mathtt{on})[0, 0]$$

3 Repair Algorithm

Let $\mathcal{R} = (\mathrm{SUP}(P^i))_{1 \le i \le n}$ denote a set of SUP requirements and suppose that it is either vacuous or rt-inconsistent. Given $M \subseteq \{(f,i) \mid 1 \le i \le n,\ f \in \mathcal{F}\}$ of indexed parameters of $\mathcal{R}$, we want to render $\mathcal{R}$ rt-consistent *and* non-vacuous by replacing the parameters in M by fresh conjunctive formulas or time bounds.

Definition 3 (ReqFix). *Given a set $\mathcal{R} = (\mathrm{SUP}(P^i))_{1 \le i \le n}$ of requirements, and a subset $M \subseteq \{(f,i) \mid 1 \le i \le n, f \in \mathcal{F}\}$, find a substitution ξ such that $\mathcal{R}' = \mathsf{tmp}_M(\mathcal{R})[\xi]$ is non-vacuous and rt-consistent.*

Thus, our goal is to repair the given requirements by modifying the allowed set M of parameters. The most general use of the algorithm is to let the user identify the set M. This can be based on their expertise, while we discuss automatizing the choice of M using rt-inconsistency or vacuity proofs in Section 5.

We will also consider a particular use of the algorithm. Notice that some rt-inconsistencies can be repaired by adding a new requirement as we saw in Example 2. The ReqFix problem can be instantiated to add a new requirement as follows. Let trivial denote the SUP requirement where all Boolean parameters are $\top$, and all time parameters are 0. This requirement is trivially satisfied. We add the trivial requirement to $\mathcal{R}$, and let M be the set of all parameters of trivial. Note however that vacuity cannot be repaired by a new requirement, so this only applies to rt-inconsistency.

3.1 Checking Non-Vacuity and rt-Consistency

For a finite trace $\sigma \in (2^{\mathsf{AP}})^*$, let $\mathsf{trig}_\sigma(R)$ denote a propositional formula that is true if, and only if, σ has triggered R. This formula guesses the execution of the SUP automaton on the trace σ and constrains it to visit the state delay. Similarly, a propositional formula can be built for $\sigma \models \mathcal{R}$ (as well as for $\sigma \not\models \mathcal{R}$) by guessing an execution on the automata corresponding to each $R \in \mathcal{R}$ and constraining these to end outside of err (resp. at err).

We perform non-vacuity checking for a requirement $R \in \mathcal{R}$ as a bounded search for a trace that triggers R without failing $\mathcal{R}$.

Definition 4. *For a given set of requirements $\mathcal{R}$, $R \in \mathcal{R}$, and bound $\alpha > 0$, define* $\mathsf{nonvac}(R, \mathcal{R})$ *as* $\exists \sigma \in (2^{\mathsf{AP}_1} \cdot \ldots \cdot 2^{\mathsf{AP}_\alpha}).\ \mathsf{trig}_\sigma(R) \wedge \sigma \models \mathcal{R}$.

Notice that each 2^{AP_i} defines the valuation at the i-th step. This is thus a partial check since the bound α needs to be fixed. Notice that even though σ triggers R and $\sigma \models \mathcal{R}$, it might be that no infinite extensions of such a σ satisfy $\mathcal{R}$; nonetheless, since we also ensure that $\mathcal{R}$ is rt-consistent, such an extension will be guaranteed to exist. If $\mathsf{nonvac}(R, \mathcal{R})$ is true, then one can query the solver for a witness trace σ triggering R and satisfying $\mathcal{R}$.

We will use template variants of the above formulas: $\mathsf{trig}_\sigma(\mathsf{tmp}_M(R))$, $\sigma \models \mathsf{tmp}_M(\mathcal{R})$, $\sigma \not\models \mathsf{tmp}_M(\mathcal{R})$, $\mathsf{nonvac}(\mathsf{tmp}_M(R), \mathsf{tmp}_M(\mathcal{R}))$. These simply consist in replacing formulas corresponding to parameters in M by templates. The set of free

variables of the latter formulas is AP_M. As in Section 2, applying a substitution for AP_M determines the truth value of each formula.

This allows us to constrain substitutions ξ we want to compute. For instance, if we want ξ to define a new requirement set $\mathsf{tmp}_M(\mathcal{R})[\xi]$ that is satisfied by a given trace σ, and in which $R \in \mathcal{R}$ is non-vacuous, we can check the satisfiability of $\sigma \models \mathsf{tmp}_M(\mathcal{R}) \wedge \mathsf{nonvac}(\mathsf{tmp}_M(R), \mathsf{tmp}_M(\mathcal{R}))$, and choose ξ as a model of this formula. We generalize this idea into an algorithm in the next section.

To check rt-consistency, one can use, as a black box, any algorithm given in [27, 23, 18]. Here, we consider a bounded model checking approach and look for an rt-inconsistency witness of bounded length using an SMT solver, following the formulation of Theorem 1. This approach only gives partial guarantees, it improves the performance while ruling out any counterexample of a given length. A sound and complete algorithm from [23, 18] can be used instead to make the check complete.

3.2 Algorithm for ReqFix

Consider $\mathcal{R} = \mathrm{SUP}(P_i)_{1 \leq i \leq n}$ and a subset M of indexed parameters. Let $\mathcal{R}_M \subseteq \mathcal{R}$ be the subset of requirements with parameters in M, and $\overline{\mathcal{R}}_M = \mathcal{R} \setminus \mathcal{R}_M$. That is, only $\mathcal{R}_M$ has modifiable parameters.

The algorithm consists in guessing conjunctive formulas for parameters in M, that is, a substitution ξ that satisfies a set of constraints $\mathcal{C}$ that we iteratively build. If the guessed substitution ξ yields a non-vacuous and rt-consistent requirement set, then we return $\mathsf{tmp}_M(\mathcal{R})[\xi]$ as the new requirement set. Otherwise, the algorithm derives new constraints to add to $\mathcal{C}$ and iterates.

Assume that $\mathcal{R}$ is vacuous, that is, there exists $R \in \mathcal{R}$ which cannot be triggered. Then, the substitution ξ we are looking for must be such that $\mathsf{tmp}_M(R)[\xi]$ is non-vacuous in $\mathsf{tmp}_M(\mathcal{R})[\xi]$, that is, we must add the following formula to $\mathcal{C}$: $\mathsf{nonvac}(\mathsf{tmp}_M(R)[\xi], \mathsf{tmp}_M(\mathcal{R})[\xi])$.

Assume that $\mathcal{R}$ is rt-inconsistent, σ is an rt-inconsistency witness.

1. If σ is an rt-inconsistency witness for $\overline{\mathcal{R}}_M$, since we can only modify $\mathcal{R}_M$, then we need $\mathsf{tmp}_M(\mathcal{R}_M)[\xi]$ to rule out σ, that is, σ must fail $\mathsf{tmp}_M(\mathcal{R}_M)[\xi]$. We thus add $\sigma \not\models \mathsf{tmp}_M(\mathcal{R}_M)$ to the constraint set $\mathcal{C}$.
2. If σ is not an rt-inconsistency witness for $\overline{\mathcal{R}}_M$, then σ can be extended without failing $\overline{\mathcal{R}}_M$, but these extensions lead to failure in $\mathcal{R}_M$. In order to rule out the witness σ, $\mathsf{tmp}_M(\mathcal{R}_M)[\xi]$ must be such that either σ is rejected (*i.e.*, $\sigma \not\models \mathsf{tmp}_M(\mathcal{R}_M)[\xi]$), or σ admits a one-step extension that satisfies $\mathsf{tmp}_M(\mathcal{R})[\xi]$. This constraint on ξ is written as $\sigma \not\models \mathsf{tmp}_M(\mathcal{R}_M) \vee \mathsf{ext}_\sigma(\mathsf{tmp}_M(\mathcal{R}))$, where $\mathsf{ext}_\sigma(\mathsf{tmp}_M(\mathcal{R})) = \exists a \in 2^{\mathsf{AP}}.\ \sigma \cdot a \models \mathsf{tmp}_M(\mathcal{R})$.

The following lemma shows that the constraints added in the two cases described above are necessary in order to rule out the rt-inconsistency witness.

Lemma 1. *Let σ be an rt-inconsistency witness for $\mathcal{R}$.*

1. *If σ is an rt-inconsistency witness in $\overline{\mathcal{R}}_M$, then for all requirement sets $\mathcal{R}'$ with $\sigma \models \mathcal{R}'$, σ is an rt-inconsistency witness in $\overline{\mathcal{R}}_M \cup \mathcal{R}'$.*

2. *If σ is not an rt-inconsitency witness in $\overline{\mathcal{R}}_M$, then for all requirement sets $\mathcal{R}'$ with $\sigma \models \mathcal{R}' \wedge \neg \mathsf{ext}_\sigma(\overline{\mathcal{R}}_M \cup \mathcal{R}')$, σ is an rt-inconsistency witness in $\overline{\mathcal{R}}_M \cup \mathcal{R}'$.*

Algorithm. The full procedure is described in Algorithm 1. Its inputs are a set $\mathcal{R}$ of requirements, and a subset M of indexed parameters of $\mathcal{R}$. For any propositional formula Φ, we denote by $\mathsf{SAT}(\Phi)$ the satisfiability check which returns either true and a model for Φ, or false.

The algorithm starts with a vacuity check inside the set $\overline{\mathcal{R}}_M$ on line 2: if $\overline{\mathcal{R}}_M$ itself is vacuous, then $\mathcal{R}$ cannot be repaired and the algorithm rejects. We maintain a set of constraints $\mathcal{C}$, which contains non-vacuity constraints of the form $\mathsf{nonvac}(\mathsf{tmp}_M(R), \mathsf{tmp}_M(\mathcal{R}))$ and constraints of the forms $\sigma \not\models \mathsf{tmp}_M(\mathcal{R}_M)$, $\sigma \models \mathsf{tmp}_M(\mathcal{R}_M)$ and $\sigma \not\models \mathsf{tmp}_M(\mathcal{R}_M) \vee \mathsf{ext}_\sigma(\mathsf{tmp}_M(\mathcal{R}))$. Recall that the set of free variables of these formulas is AP_M, so a model for the query on line 5 defines a substitution ξ, and thus a new requirement set $\mathsf{tmp}_M(\mathcal{R})[\xi]$.

On line 7, we check if $\overline{\mathcal{R}}_M \cup \mathcal{R}'_M$ is vacuous, and then identify a requirement R that cannot be triggered without violating $\overline{\mathcal{R}}_M \cup \mathcal{R}'_M$. We necessarily have $R \in \overline{\mathcal{R}}_M$, since all requirements in $\mathcal{R}'_M$ are non-vacuous as they satisfy $\mathcal{C}$. We find a trace σ that triggers R while satisfying $\overline{\mathcal{R}}_M$. Such a trace σ exists by line 2, but necessarily violates $\mathcal{R}'_M$. We add $\sigma \models \mathsf{tmp}_M(\mathcal{R}_M)$ to $\mathcal{C}$, which ensures that subsequent iterations will make sure that σ triggers R without violating $\mathcal{R}'_M$.

If $\overline{\mathcal{R}}_M \cup \mathcal{R}'_M$ is non-vacuous, then we check its rt-consistency. If it is rt-consistent, then the algorithm has succeeded, and we return $\overline{\mathcal{R}}_M \cup \mathcal{R}'_M$. Otherwise, we consider a witness σ to rt-inconsistency. We distinguish two cases as above: On line 13, we check if σ is already a witness to the rt-inconsistency of $\overline{\mathcal{R}}_M$, in which case we add the constraint $\sigma \not\models (\mathsf{tmp}_M(\mathcal{R}_M))$. Otherwise, we add $\sigma \not\models \mathsf{tmp}_M(\mathcal{R}_M) \vee \mathsf{ext}_\sigma(\overline{\mathcal{R}}_M \cup \mathsf{tmp}_M(\mathcal{R}))$.

Observe that if the query on line 5 is unsatisfiable, then the algorithm returns "Unknown", in which case the result is inconclusive. In fact, since the choice of the non-vacuity constraints on Line 8 is arbitrary, the unsatisfiability of the query does not imply the absence of solution. The algorithm could be rendered complete using backtracking although we have not explored this direction.

Minimizing Distance. It may be desirable to compute a solution $\mathsf{tmp}_M(\mathcal{R})[\xi]$ that is syntactically close to $\mathcal{R}$, so as to make a minimal number of changes during the repair. To formalize this, let us define a distance between conjunctive formulas. Let

$$d(\ell_1 \wedge \ldots \wedge \ell_m, \ell'_1 \wedge \ldots \wedge \ell'_n) = |\mathsf{Supp}(\{\ell_1, \ldots, \ell_m\} \oplus \{\ell'_1, \ldots, \ell'_n\})|,$$

where $\oplus$ denotes the symmetric difference, and Supp is the set of variables appearing in the given set of literals. For instance, $d(\neg \mathtt{on}, \mathtt{on}) = 1$, and $d(\mathtt{on} \wedge \mathtt{blink}, \mathtt{on}) = 1$. For two time bounds T, T', we extend this definition to $d(T, T') = |T - T'|$. The distance between two SUPs with parameters P and P' is the weighted sum of the distances of their parameters: $d(P, P') = w_b \cdot \sum_{f \in \mathcal{F}_b} d(P_f, P'_f) + w_t \cdot \sum_{f \in \mathcal{F}_t} d(P_f, P'_f)$ for given weights $w_b, w_t \geq 0$. Furthermore, given two SUP requirement sets of the same size, $\mathcal{R} = (\mathrm{SUP}(P^i))_{1 \leq i \leq n}$ and $\mathcal{R}' = (\mathrm{SUP}(P'^i))_{1 \leq i \leq n}$, define $d(\mathcal{R}, \mathcal{R}') = \sum_{i=1}^{n} d(P^i, P'^i)$.

Input: A set $\mathcal{R}$ of SUP requirements, and parameter set M

1 Let $\mathcal{R}_M \subseteq \mathcal{R}$ the set of those requirements that contain parameters in M, and $\overline{\mathcal{R}}_M = \mathcal{R} \setminus \mathcal{R}_M$
2 **if** $\exists R \in \overline{\mathcal{R}}_M.\ \neg\mathsf{nonvac}(R, \overline{\mathcal{R}}_M)$ **then**
3 | **return** Reject
4 $\mathcal{C} \leftarrow \bigwedge_{R \in \mathcal{R}_M} \mathsf{nonvac}(\mathsf{tmp}_M(R), \mathsf{tmp}_M(\mathcal{R}))$
5 **while** $\mathsf{SAT}(\bigwedge_{\phi(M) \in \mathcal{C}} \phi(M))$ **do**
6 | Let ξ be a model of this formula, and let $\mathcal{R}'_M = \mathsf{tmp}_M(\mathcal{R}_M)[\xi]$
7 | **if** $\overline{\mathcal{R}}_M \cup \mathcal{R}'_M$ *is vacuous* **then**
8 | | Choose $R \in \overline{\mathcal{R}}_M$ which cannot be triggered
9 | | Let σ be a trace that triggers R and satisfies $\overline{\mathcal{R}}_M$
10 | | $\mathcal{C} \leftarrow \mathcal{C} \cup \{\sigma \models \mathsf{tmp}_M(\mathcal{R}_M)\}$
11 | **else if** $\overline{\mathcal{R}}_M \cup \mathcal{R}'_M$ *is rt-inconsistent* **then**
12 | | Let σ be an rt-inconsistency witness
13 | | **if** σ *is an rt-inconsistency witness for* $\overline{\mathcal{R}}_M$ **then**
14 | | | $\mathcal{C} \leftarrow \mathcal{C} \cup \{\sigma \not\models \mathsf{tmp}_M(\mathcal{R}_M)\}$
15 | | **else**
16 | | | $\mathcal{C} \leftarrow \mathcal{C} \cup \{\sigma \not\models \mathsf{tmp}_M(\mathcal{R}_M) \vee \mathsf{ext}_\sigma(\overline{\mathcal{R}}_M \cup \mathsf{tmp}_M(\mathcal{R}_M))\}$
17 | **else**
18 | | **return** $\overline{\mathcal{R}}_M \cup \mathcal{R}'_M$
19 **return** Unknown

Algorithm 1: Algorithm for ReqFix.

In order to find the substitution that minimizes the distance between the original requirement set and the new one, we use MaxSMT [10]. The query on line 5 is considered a *hard* formula (that must be satisfied), and the following are *soft* formulas (that may be satisfied or violated):

$$w_b \cdot \sum_{(f,i) \in M : f \in \mathcal{F}_b} (\mathsf{if}\ (x_{f,i} \neq \bar{x}_{f,i})\ \mathsf{then}\ 1\ \mathsf{else}\ 0) + w_t \cdot \sum_{(f,i) \in M : f \in \mathcal{F}_t} |x_{f,i} - \bar{x}_{f,i}| \leq k,$$

for all $0 \leq k \leq m$, for an appropriately chosen m, and weights $w_b, w_t \geq 0$, where $x_{f,i} \in \mathsf{AP}_M$ and the $\bar{x}_{f,i}$ are constant values defining the substitution that yields the original requirement set $\mathcal{R}$. The MaxSMT solver returns a model that satisfies the hard formulas, and satisfies a maximal number of soft formulas; which means minimizing $d(\mathcal{R}, \mathsf{tmp}_M(\mathcal{R})[\xi])$.

4 Experiments

We implemented our techniques in Python and applied it to four case studies from the literature [12, 17, 24] as well as to a set of anonymized benchmarks from [23]. We manually introduced rt-inconsistencies by removing a requirement, or by modifying the parameters of a requirement. The summary of the results are shown in Table 1. We considered two applications of our algorithm. In the first case, starting from an rt-inconsistent set, we looked for a repair by *generating* a

new requirement with the minimal number of literals and the least time bounds. We call this the *generation variant* of our program. Notice that this consists in minimizing the distance of the generated requirement to the trivial requirement. In the second case, we selected the parameters of a requirement as the set M to be *modified*, and looked for a repair that minimizes the distance of the new requirement with the old one. We call this the *modification variant* of our program. While the generation variant allowed us to find very simple repairs, these were not always satisfactory. The second one yields repairs that are syntactically very similar to the initial requirement, and were closer to the intended behavior in the considered case studies. We provide a focus on two case studies below.

Blinking system. This case study corresponds to the behaviour of the turning light indicator in a car [17]. The pitman arm can be moved up or down to a first position (5 degrees) to turn the indicator on for only 3 cycles; in each direction (up and down), it can also be moved to a second position (7 degrees) where the indicator remains on until the arm is moved back. We analyzed a set of 7 requirements, including the following one.

$$R : (\texttt{down5}, \texttt{down5}, \neg\texttt{down5})[0,3] \xrightarrow{[5,5]} (\neg\texttt{down5} \wedge \neg\texttt{down7}, \texttt{true}, \texttt{true})[0,0],$$

which states that if the pitman arm is maintained down for less than 3 time units, then it will automatically be on neutral position 5 time units later. We modified this, by introducing a typo, into the following requirement in order to introduce an rt-inconsistency:

$$R' : (\texttt{down5}, \texttt{down5}, \neg\texttt{down5})[0,3] \xrightarrow{[5,5]} (\neg\texttt{down5}, \texttt{true}, \texttt{true})[0,0],$$

and ran the modification variant of the algorithm to find a repair by modifying the parameters of R'. A solution was found after 3 iterations and 12 seconds:

$$R_{\text{fix}} : (\texttt{down5}, \texttt{down5}, \neg\texttt{down5})[0,3] \xrightarrow{[5,5]} (\neg\texttt{down5}, \texttt{true}, \neg\texttt{down7})[0,0],$$

which is semantically equivalent to R.

When we ran the generation variant of the algorithm to find a repair to the set obtained by removing R altogether, we obtained the following requirement:

$$R'_{\text{fix}} : (\texttt{true}, \texttt{true}, \texttt{true})[0,0] \xrightarrow{[0,0]} (\texttt{true}, \texttt{down7}, \neg\texttt{blink})[0,6].$$

This requirement enforces that the blinking must be disabled every 6 time units, while in the meantime, the pitman arm kept down by 7 degrees. While this intuitively does not correspond to a desirable requirement, it does ensure the rt-consistency and non-vacuity. In practice, one would perhaps need to allow the user to inspect the repair and accept or reject, add constraints and ask for a new repair. This could yield more satisfactory repairs for the generation variant.

Carriage line control. This example from [24, Appendix 4.20] represents a carriage in charge of bringing a piece of material from a container to a conveyor. When the carriage receives the piece of material, it moves forward to a place where an

Case study	Size	Modification		Generation	
		time	#iter.	time	#iter.
Carriage line[24]	12	24s	4	38s	11
Landing gear[12]	10	14s	2	21s	6
Car light blink.[17]	6	13s	4	13m47s	44
Cruise ctrl.[17]	7	9s	4	12s	6
part1-04	13	29s	11	21s	9
part1-05	14	19s	4	47s	17
part1-06	16	17s	4	21s	10
part2-06	18	32s	11	48s	16
part2-07	24	43s	5	58s	12
part2-08	27	51s	4	1m8s	13
part2-10	80	3m47s	2	2m39s	1
part3-02	26	45s	5	1m3s	13
part3-04	13	24s	8	21s	8
part3-05	26	1m6s	9	1m7s	13
part3-08	27	1m52s	10	1m32s	19
part3-14	24	3m8s	3	10m55s	18
part3-16	22	TO	-	TO	-

Table 1. Results of the benchmarks for two variants of the algorithm: the generation variant repairs requirement sets by adding a fresh requirement; the modification requirement repairs by modifying the parameters of a designated requirement. The former minimizes the number of literals and the size of the time bounds introduced, while the latter minimizes the distance between the designated requirement and the new one. The size column shows the number of requirements; the time column is execution time, and %iter. column shows the number of iterations. A bound of $\alpha = 30$ was used for non-vacuity and rt-consistency checks.

arm will push the piece onto the conveyor, and then moves back to its original location.

We described the behaviour of this system using 12 SUP requirements, of which 6 involved timing constraints. As an example, we have the following SUP: $R : (\texttt{fwd}, \texttt{true}, \texttt{true})[0,0] \xrightarrow{[1,1]} (\neg\texttt{bckwd}, \neg\texttt{bckwd}, \neg\texttt{bckwd} \wedge \texttt{right})[0,20)$, stating that when the carriage is at its forward position, then at the next step, it should not be at the backward position until it starts moving right. This requirement is used to model the physical environment: the carriage cannot be both on the forward and backward limits, and it must start moving right before it can reach the backward limit.

Modifying this requirement by introducing a typo as follows leads to an rt-inconsistency. $R' : (\texttt{fwd}, \texttt{true}, \texttt{true})[0,0] \xrightarrow{[1,1]} (\neg\texttt{bckwd}, \neg\texttt{bckwd}, \texttt{true})[0,20)$,

The modification variant of our tool computed the following repair: $R'_{\text{fix}} : (\texttt{fwd}, \texttt{true}, \texttt{true})[0,0] \xrightarrow{[1,1]} (\neg\texttt{bckwd}, \neg\texttt{bckwd}, \neg\texttt{push})[0,20)$, which says that the carriage cannot be in the backward position until the arm stops pushing the object. This is slightly different than the original requirement R but it does constrain the environment in a similar way. In fact, the idea of the system is that

the carriage must move right when the arm stops pushing, and R'_{fix} says that only then can the carriage reach the backward limit.

5 Conclusion

We believe that the practical application of requirement repair would be a tool that assists the designer by suggesting repairs. The designer should be able to either pick a suggested repair, suggest additional constraints and request different repairs. Our program is currently a proof of concept and many additional features would be required to turn it into such a tool.

One of the possible directions is to be able to choose the set M automatically. This is possible in some cases, for instance, if $\mathsf{nonvac}(R, \mathcal{R})$ is not true, then one can determine the set of parameters involved in its unsatisfiability proof, which can be included in M (we know that at least one such parameter must be in M). The choice of M with a similar method is less obvious for rt-consistency and will be the subject of future work.

Another important direction would be the computation of solutions that are close to the original requirement set *semantically*, for instance, minimizing the number of traces that are accepted by one but not the other set.

References

[1] B. K. Aichernig *et al.* Require, test, and trace it. *Int. Journal Software Tools for Technology Transfer*, 19(4):409–426, Aug 2017.

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

[3] R. Alur *et al.* Search-based program synthesis. *Commun. ACM*, 61(12):84–93, Nov. 2018.

[4] É. André *et al.* Repairing timed automata clock guards through abstraction and testing. In *Tests and Proofs*, p. 129–146. Springer, 2019.

[5] E. Bartocci, L. Bortolussi, and G. Sanguinetti. Data-driven statistical learning of temporal logic properties. In *Formal Modeling and Analysis of Timed Systems (FORMATS)*, p. 23–37. Springer, 2014.

[6] J. S. Becker. Analyzing consistency of formal requirements. In *Automated Verification of Critical Systems (AVoCS)*, 2019.

[7] J. Bendík *et al.* Timed automata relaxation for reachability. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 12651, p. 291–310. Springer, 2021.

[8] A. Benveniste *et al.* Contracts for system design. *Foundations and Trends in Electronic Design Automation*, 12(2-3):124–400, 2018.

[9] T. Bienmüller *et al.* Modeling requirements for quantitative consistency analysis and automatic test case generation. In *Workshop on Formal and Model-Driven Techniques for Developing Trustworthy Systems*, 2016.

[10] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability.* IOS press, 2009.

[11] G. Bombara *et al.* A Decision Tree Approach to Data Classification using Signal Temporal Logic. In *Hybrid Systems: Computation and Control (HSCC)*, p. 1–10, Vienna, Austria, April 2016.

[12] F. Boniol and V. Wiels. Landing gear system, 2014. `https://www.irit.fr/ABZ2014/landing_system.pdf`.

[13] R. Brenguier, S. Göller, and O. Sankur. A comparison of succinctly represented finite-state systems. In *Int. Conf. Concurrency Theory (CONCUR)*, LNCS 7454, p. 147–161, Newcastle, UK, Sept. 2012. Springer.

[14] C. Ellen, S. Sieverding, and H. Hungar. Detecting consistencies and inconsistencies of pattern-based functional requirements. In *Formal Methods for Industrial Critical Systems (FMICS)*, p. 155–169. Springer, 2014.

[15] M. Ergurtuna, B. Yalcinkaya, and E. A. Gol. An automated system repair framework with signal temporal logic. *Acta Informatica*, p. 1–27, 2021.

[16] C. L. Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Commun. ACM*, 62(12):56–65, Nov. 2019.

[17] F. Houdek and A. Raschke. Adaptive exterior light and speed control system, 2021. `https://abz2021.uni-ulm.de/resources/files/casestudyABZ2020v1.17.pdf`.

[18] T. Jéron *et al.* Incremental methods for checking real-time consistency. In *Int. Conf. Formal Modeling and Analysis of Timed Systems (FORMATS)*, LNCS 12288, 2020.

[19] M. Kölbl, S. Leue, and T. Wies. Clock bound repair for timed systems. In *Int. Conf. Computer Aided Verification (CAV)*, p. 79–96. Springer, 2019.

[20] M. Kölbl, S. Leue, and T. Wies. Tartar: A timed automata repair tool. In *Int. Conf. Computer Aided Verification (CAV)*, p. 529–540. Springer, 2020.

[21] Z. Kong *et al.* Temporal logic inference for classification and prediction from data. In *17th Int. Conf. Hybrid Systems: Computation and Control (HSCC)*, p. 273–282, New York, NY, USA, 2014. ACM.

[22] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *Int. J. Softw. Tools Technol. Transf.*, 4(2):224–233, 2003.

[23] V. Langenfeld *et al.* Scalable analysis of real-time requirements. In *Int. Requirements Engineering Conf (RE)*, p. 234–244. IEEE, 2019.

[24] Mitsubishi Electric Corporation. Mitsubishi programmable controller – Training manual, 2012. `https://dl.mitsubishielectric.com/dl/fa/document/manual/school_text/sh081123eng/sh081123enga.pdf`.

[25] D. Neider and I. Gavran. Learning linear temporal properties. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, p. 1–10. IEEE, 2018.

[26] K. Pohl. *Requirements engineering: fundamentals, principles, and techniques*. Springer, 2010.

[27] A. Post, J. Hoenicke, and A. Podelski. rt-inconsistency: a new property for real-time requirements. In *Fundamental Approaches to Software Engineering (FASE)*, LNCS 6603. Springer, 2011.

[28] A. Post, J. Hoenicke, and A. Podelski. Vacuous real-time requirements. In *IEEE Int. Requirements Engineering Conf. (RE)*, p. 153–162, Aug 2011.

[29] T. Teige, T. Bienmüller, and H. J. Holberg. Universal pattern: Formalization, testing, coverage, verification, and test case generation for safety-critical requirements. In *19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'16)*, p. 6–9. Albert-Ludwigs-Universität Freiburg, 2016.

[30] H. Yang, B. Hoxha, and G. Fainekos. Querying parametric temporal logic properties on embedded systems. In *Testing Software and Systems*, p. 136–151. Springer, 2012.