*Fundamenta Informaticae 178 (2020) 1–27*
*DOI 10.3233/FI-2020-1950*
*IOS Press*



# Verification and Parameter Synthesis for Real-Time Programs using Refinement of Trace Abstraction*

**Franck Cassez**†
*Department of Computing*
*Macquarie University, Sydney, Australia*
*franck.cassez@mq.edu.au*

**Peter Gjøl Jensen**
*Department of Computer Science*
*Aalborg University, Denmark*
*pgj@cs.aau.dk*

**Kim Guldstrand Larsen**
*Department of Computer Science*
*Aalborg University, Denmark*
*kgl@cs.aau.dk*

**Abstract.** We address the safety verification and synthesis problems for real-time systems. We introduce real-time programs that are made of instructions that can perform assignments to discrete and real-valued variables. They are general enough to capture interesting classes of timed systems such as timed automata, stopwatch automata, time(d) Petri nets and hybrid automata. We propose a semi-algorithm using refinement of trace abstractions to solve both the reachability verification problem and the parameter synthesis problem for real-time programs. All of the algorithms proposed have been implemented and we have conducted a series of experiments, comparing the performance of our new approach to state-of-the-art tools in classical reachability, robustness analysis and parameter synthesis for timed systems. We show that our new method provides solutions to problems which are unsolvable by the current state-of-the-art tools.

*A preliminary version of this work appeared in [1].

†Address for correspondence: Department of Computing, Macquarie University, Sydney, Australia

# 1. Introduction

Model-checking is a widely used formal method to assist in verifying software systems. A wide range of model-checking techniques and tools are available and there are numerous successful applications in the safety-critical industry and the hardware industry – in addition the approach is seeing an increasing adoption in the general software engineering community. The main limitation of this formal verification technique is the so-called *state explosion problem*. *Abstraction refinement techniques* were introduced to overcome this problem. The most well-known technique is probably the *Counter Example Guided Abstraction Refinement* (CEGAR) method pioneered by Clarke *et al.* [2]. In this method the state space is abstracted with predicates on the concrete values of the program variables. The (counter-example guided) *trace abstraction refinement* (TAR) method was proposed later by Heizmann *et al.* [3, 4] and is based on abstracting the set of traces of a program rather than the set of states. These two techniques have been widely used in the context of software verification. Their effectiveness and versatility in verifying *qualitative* (or functional) properties of C programs is reflected in the most recent *Software Verification* competition results [5].

**Analysis of timed systems.** Reasoning about *quantitative* properties of programs requires extended modeling features like real-time clocks. *Timed Automata* [6] (TA), introduced by Alur and Dill in 1989, is a very popular formalism to model real-time systems with dense-time clocks. Efficient symbolic model-checking techniques for TA are implemented in the real-time model-checker UPPAAL [7]. Extending TA, e.g., with the ability to stop and resume clocks (stopwatches), leads to undecidability of the reachability problem [8, 9] which is the basic verification problem. As a result, semi-algorithms have been designed to verify extended classes of TA e.g., *hybrid automata*, and are implemented in a number of dedicated tools [10, 11, 12]. However, a common difficulty with the analysis of quantitative properties of timed automata and extensions thereof is that specialized data-structures are needed for each extension and each type of problem. As a consequence, the analysis tools have special-purpose efficient algorithms and data-structures suited and optimized only towards their specific problem and extension.

In this work we aim to provide a uniform solution to the analysis of timed systems by designing a generic semi-algorithm to analyze real-time programs which semantically captures a wide range of specification formalisms, including hybrid automata. We demonstrate that our new method provides solutions to problems which are unsolvable by the current state-of-the-art tools. We also show that our technique can be extended to solve specific problems like robustness and parameter synthesis.

**Related work.** The *trace abstraction refinement* (TAR) technique was proposed by Heizmann *et al.* [3, 4]. Wang *et al.* [13] proposed the use of TAR for the analysis of timed automata. However, their approach is based on the computation of the standard *zones* which comes with usual limitations: it is not applicable to extensions of TA (e.g., stopwatch automata) and can only discover predicates that are zones. Moreover, their approach has not been implemented and it is not clear whether it can outperform state-of-the-art techniques e.g., as implemented in UPPAAL.

Several works have investigated CEGAR techniques in both timed and hybrid settings [14, 15, 16, 17]. The CEGAR technique has also been extended to parameter-synthesis [16]. As proved by Heizmann *et al.* [3], such methods are special cases of the TAR framework.

The *IC3* [18] verification approach has also been deployed for the verification of hybrid systems [19] but relies on a fix-point computation over a combined encoding of the transition-function, rather than a trace-subtraction approach. IC3 approaches and the likes have also been used for parameter synthesis [20, 19, 21]. While similar fundamental techniques are leveraged in these approaches (e.g. [20] utilizes Fourier-Motzkin-elimination), we note that our refinement method (TAR) is radically different in nature. IC3 is an iterative fix-point computation over an up-front and complete encoding of the transition-function.

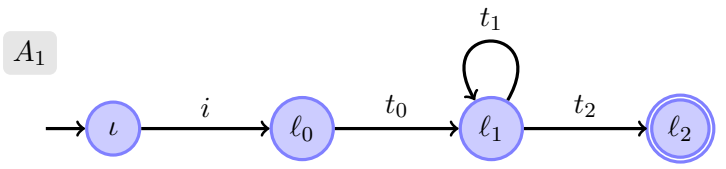
Since the publication of a preliminary version of this paper [1], Kafle *et al.* [22] have demonstrated a novel method of parameter synthesis for timed systems via Constrained Horn Clauses (CHC). While their approach shows promising results for the Fischers parameter synthesis examples from [1], it currently relies on manual translation of a given problem into CHC format, hindering its applicability to large systems.

As mentioned earlier, our technique allows for a unique and logical (predicates) representation of sets of states accross different models (timed, hybrid automata) and problems (reachability, robustness, parameter synthesis), which is in contrast to state-of-the-art tools such as UPPAAL [7], SPACEEX [11], HYTECH [12], PHAVER [10], VERIFIX [23], SYMROB [24] and IMITATOR [25] that rely on special-purpose polyhedra libraries to realize their computation.

We propose a new technique which is radically different to previous approaches and leverages the power of SMT-solvers to discover non-trivial invariants for a large class of real-time systems including the class of hybrid automata. All the previous analysis techniques compute, reduce and check the state-space either up-front or on-the-fly, leading to the construction of significant parts of the state-space. In contrast our approach is an abstraction refinement method and the refinements are built by discovering non-trivial program invariants that are not always expressible using zones, or polyehdra. For instance they can express constraints that combine discrete and continuous variables of the system. This enables us to use our algorithm on non-decidable classes like stopwatch automata, and successfully (i.e., the algorithm terminates) check instances of these classes. A simple example is discussed in Section 2.

**Our contribution.** We propose a variant of the trace abstractions refinement (TAR) technique to solve the reachability problem and the parameter synthesis problem for real-time programs. Our approach combines an automata-theoretic framework and state-of-the-art Satisfiability Modulo Theory (SMT) techniques for discovering program invariants. We demonstrate on a number of case-studies that this new approach can compute answers to problems unsolvable by special-purpose tools and algorithms in their respective domain.

This paper is an extended version of [1] in which we first introduced TAR for real-time programs. In this extended version, we provide a comprehensive introduction illustrated by more examples, extensions of the original algorithms from [1] and the proofs of theorems and lemmas.




Discrete Instructions

| Edge | Guard | Update |
|---|---|---|
| $i$ | *True* | `x:=y:=z:=0` |
| $t_0$ | *True* | `z:=0` |
| $t_1$ | `x==1` | `x:=0` |
| $t_2$ | `x-y>=1 and z<1` | - |

Continuous Instructions

| Location | Rate |
|---|---|
| $\iota$ | `dy/dt=1` |
| $\ell_0$ | `dy/dt=1` |
| $\ell_1$ | `dy/dt=0` |
| $\ell_2$ | `dy/dt=0` |

Figure 1: Real-time program $P_1$: CFG $A_1$ of $P_1$ (top) with the accepting location $\ell_2$ and its instructions (bottom).

## 2. Motivations

***Real-Time Programs.*** Figure 1 is an example of a real-time program $P_1$. It is defined by a finite automaton $A_1$ (Figure 1, top) which is the *control flow graph (CFG)* of $P_1$, and some *continuous and discrete instructions* (bottom). The control flow graph $A_1$ accepts the regular language $\mathcal{L}(A_1) = i.t_0.t_1^*.t_2$: the program starts in (control) location $\iota$ and is completed when $\ell_2$ (accepting location) is reached. The program variables are $x, y, z$ which are real numbers. This real-time program is the specification of a stopwatch automaton with 2 clocks, $x$ and $z$, and one stopwatch $y$. The variables are updated according to the following rules:

- Each edge's label defines a *guard* $g$ (a condition on the variables) for which the edge is enabled, and an *update* $u$ which is an assignment to the variables when the edge is taken. For instance the edge $t_1$ can be taken when the valuation of the variable $x$ is 1 and when it is taken, $x$ is reset. This corresponds to a *discrete* transition of the program.

- Each location is associated with a *rate vector* $r$ that defines the derivatives of the variables. The default derivative for a variable is 1 (we omit the rates for $x, z$ in the Figure). For instance in location $\ell_0$ the derivatives are $(1, 0, 1)$ (order is $x, y, z$). When the program is in location $\ell_0$ the variables $x, y, z$ increase at a rate defined by their respective derivatives: $x, z$ increase by 1 each time unit, and $y$ is frozen (derivative is 0). This corresponds to a *continuous* transition of the program.

A sequence of program instructions $w = a_0.a_1.\cdots.a_n \in \mathcal{L}(A_1)$ defines a (possibly empty) set of *timed words*, $\tau(w)$, of the form $(a_0, \delta_0).\cdots.(a_n, \delta_n)$ where $\delta_i \geq 0, i \in [0..n]$ is the time elapsed between two discrete transitions. For instance, the timed words associated with $i.t_0.t_2$ are of the form $(i, \delta_0).(t_0, \delta_1).(t_2, \delta_2)$, for all $\delta_i \in \mathbb{R}_{\geq 0}, i \in \{0, 1, 2\}$ such that the following constraints (predicates

that define that each transition can be fired after $\delta_i$ time units) can be satisfied[1]:

$$\underbrace{x_0 = y_0 = z_0 = \delta_0 \wedge \delta_0 \geq 0}_{\text{Time elapsing } \delta_0 \text{ in } \iota} \quad \wedge \quad \underbrace{\mathit{True}}_{\text{Guard of } i} \tag{$C_0$}$$

$$\underbrace{x_1 = y_1 = z_1 = 0 + \delta_1 \wedge \delta_1 \geq 0}_{\text{Update of } i \text{ and time elapsing } \delta_1 \text{ in } \ell_0} \quad \wedge \quad \underbrace{\mathit{True}}_{\text{Guard of } t_0} \tag{$C_1$}$$

$$\underbrace{x_2 = x_1 + \delta_2 \wedge y_2 = y_1 \wedge z_2 = 0 + \delta_2 \wedge \delta_2 \geq 0}_{\text{Update of } t_0 \text{ and time elapsing } \delta_2 \text{ in } \ell_1} \wedge \underbrace{x_2 - y_2 \geq 1 \wedge z_2 < 1}_{\text{Guard of } t_2} \tag{$C_2$}$$

These constraints encode the following semantics: $i$ is taken after $\delta_0$ time units and at that time $x, y, z$ are equal to $\delta_0$ and hence $x_0, y_0, z_0$ are the values of the variables when location $\ell_0$ is entered. The program remains in $\ell_0$ for $\delta_1$ time units. When $t_0$ is taken after $\delta_1$ time units, the values of $x, y, z$ is given by $x_1, y_1, z_1$. Finally the program remains $\delta_2$ time units in $\ell_1$ and $t_2$ is taken to reach $\ell_2$ which is the end of the program. It follows that the program can execute $i.t_0.t_2$ (or in other words, $i.t_0.t_2$ is feasible) if and only if we can find $\delta_0, \delta_1, \delta_2$ such that $C_0 \wedge C_1 \wedge C_2$ is satisfiable. Hence the set of timed words associated with $i.t_0.t_2$ is not empty iff $C_0 \wedge C_1 \wedge C_2$ is satisfiable.

***Language Emptiness Problem.*** The *timed language*, $\mathcal{TL}(P_1)$, accepted by $P_1$ is the set of timed words associated with all the (untimed) words $w$ accepted by $A_1$ i.e., $\mathcal{TL}(P_1) = \cup_{w \in \mathcal{L}(A_1)} \tau(w)$.

The *language emptiness problem* is a standard problem in Timed Automata theory [6] and is stated as follows for real-time programs:

**given a real-time program $P$, is $\mathcal{TL}(P)$ empty?**

It is known that the emptiness problem is decidable for some classes of real-time programs like Timed Automata [6], but undecidable for more expressive classes like Stopwatch Automata [9]. It is usually possible to compute symbolic representations of sets of *reachable* valuations after a sequence of transitions. However, to compute the set of reachable valuations we may need to explore an arbitrary and unbounded number of sequences. Hence only semi-algorithms exist to compute the set of reachable valuations. For instance, using PHAVER to compute the set of reachable valuations for $P_1$ does not terminate (Table 1). To force termination, we can compute an over-approximation of the set of reachable valuations. Computing an over-approximation is sound (if we declare an over-approximation of a timed language to be empty the timed language is empty) but incomplete i.e., it may result in *false positives* (we declare a timed language non empty whereas it is empty). This is witnessed by the column "UPPAAL" in Table 1 where UPPAAL over-approximates sets of valuations in the program $P_1$ using DBMs. After $i.t_0$, the over-approximation is $0 \leq y \leq x \wedge 0 \leq z \leq x$ (this is the smallest DBMs that contains the actual set of valuations reachable after $i.t_0$). This over-approximation intersects the guard $x - y \geq 1 \wedge z < 1$ of $t_2$ which enables $t_2$. Using this over-approximate set of valuations we would declare that $\ell_2$ is reachable in $P_1$ but this is an artifact of the over-approximation.[2] Neither UPPAAL nor PHAVER can prove that $\mathcal{TL}(P_1) = \varnothing$.

[1] We assume the program starts in $\iota$ and all the variables are initially zero.

[2] UPPAAL terminates with the result "the language **may** not be empty".

Table 1: Symbolic representation of reachable states after a sequence of instructions. UPPAAL concludes that $\mathcal{TL}(A_1) \neq \varnothing$ due to the over-approximation using DBMs. PHAVER does not terminate.

| Sequence | PHAVER | UPPAAL |
|---|---|---|
| $i.t_0$ | $z = x - y \wedge 0 \leq z \leq x$ | $0 \leq y \leq x \wedge 0 \leq z \leq x$ |
| $i.t_0.t_1$ | $z = x - y + 1 \wedge 0 \leq x \leq z \leq x + 1$ | $0 \leq z - x \leq 1 \wedge 0 \leq y$ |
| $i.t_0.(t_1)^2$ | $z = x - y + 2 \wedge 0 \leq x \leq z - 1 \leq x + 1$ | $1 \leq z - x \leq 2 \wedge 0 \leq y$ |
| $i.t_0.(t_1)^3$ | $z = x - y + 3 \wedge 0 \leq x \leq z - 2 \leq x + 1$ | $2 \leq z - x \leq 3 \wedge 0 \leq y$ |
| $\ldots$ | $\ldots$ | $\ldots$ |
| $i.t_0.(t_1)^k$ | $z = x - y + k \wedge 0 \leq x \leq z - k + 1 \leq x + 1$ | $k - 1 \leq z - x \leq k \wedge 0 \leq y$ |
| $\ldots$ | $\ldots$ | $\ldots$ |

***Trace Abstraction Refinement for Real-Time Programs.*** The technique we introduce can discover *arbitrary abstractions and invariants* that enable us to prove $\mathcal{TL}(P_1) = \varnothing$. Our method is a version of the *trace abstraction refinement* (TAR) technique introduced in [3] and is depicted in Figure 2.

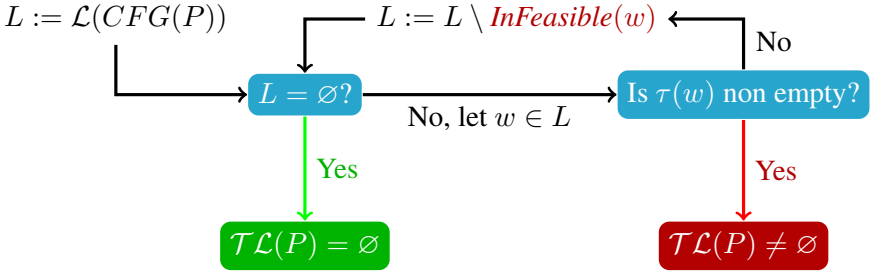



Figure 2: Trace Abstraction Refinement Loop for Real-Time Programs

Let us first introduce how the trace abstraction refinement algorithm (Figure 2) operates on a real-time program $P$:

1. the algorithm starts using the control flow graph of $P$, $CFG(P)$, and initially $L = \mathcal{L}(CFG(P))$.
2. if $L = \varnothing$ then $\mathcal{TL}(P)$ is empty and the algorithm terminates (green block).
3. otherwise, there is $w \in L$. We check whether $\tau(w)$ is empty or not:
   - If it is not empty then $\mathcal{TL}(P)$ is not empty and the algorithm terminates (red block).
   - Otherwise, we can find[3] a regular language over the alphabet of $CFG(P)$, *InFeasible*$(w)$, that satisfies: 1) $w \in$ *InFeasible*$(w)$ and 2) $\forall v \in$ *InFeasible*$(w)$, $\mathcal{TL}(v) = \varnothing$. In the next iteration of the algorithm, we look for a candidate trace in $L \setminus$ *InFeasible*$(w)$, i.e., we refine the trace abstraction $L$ by subtracting *InFeasible*$(w)$ from it.

[3]How this language is built is defined in Section 4.

Assume the algorithm terminates after $k$ iterations. In this case we were able to build a finite number of regular languages $L_1 = \mathit{InFeasible}(w_1), L_2 = \mathit{InFeasible}(w_2), \cdots, L_k = \mathit{InFeasible}(w_k)$ such that $\forall 1 \leq i \leq k, \mathcal{TL}(L_i) = \varnothing$. If we terminate with $\mathcal{TL}(P) = \varnothing$ then $\mathcal{L}(CFG(P)) \subseteq \cup_{i=1}^{k} L_i$. Otherwise if we terminate with $\tau(w_{k+1}) \neq \varnothing$ we found a witness trace $w_{k+1} \in \mathcal{L}(CFG(P)) \setminus \cup_{i=1}^{k} L_i$ such that $\tau(w_{k+1}) \neq \varnothing$ i.e., a feasible timed trace.

***Example 1: Stopwatch Automaton.*** We illustrate the algorithm using our program $P_1$:

- we initially let $L = \mathcal{L}(CFG(P_1))$. Since $w_1 = i.t_0.t_2 \in \mathcal{L}(CFG(P_1))$ and thus $w_1 \in L$ the check $L = \varnothing$ fails. We therefore check whether $\tau(w_1) = \varnothing$ which can be done by encoding the corresponding set of timed traces as described by Equations $(C_0)$–$(C_2)$ and then check whether $C_0 \wedge C_1 \wedge C_2$ is satisfiable (e.g., using an SMT-solver and the theory of Linear Real Arithmetic). $C_0 \wedge C_1 \wedge C_2$ is not satisfiable and this establishes $\tau(w_1) = \varnothing$.

- from the proof that $C_0 \wedge C_1 \wedge C_2$ is not satisfiable, we can obtain an *inductive interpolant* that comprises of two predicates $I_0, I_1$ – one for each conjunction – over the variables $x, y, z$. An example of an inductive interpolant[4] is $I_0 = x \leq y$ and $I_1 = x - y \leq z$. These predicates are *invariants* of any timed word of the untimed word $w_1$, and can be used to annotate the sequence of transitions $w_1$ with pre- and post-conditions (Equation 1), which are Hoare triples of the form $\{C\}\ a\ \{D\}$:

$$\{\mathit{True}\} \quad i \quad \{I_0\} \quad t_0 \quad \{I_1\} \quad t_2 \quad \{\mathit{False}\} \tag{1}$$

  A triple $\{C\}\ a\ \{D\}$ is *valid* if whenever we start in a state $s$ satisfying $C$, and execute instruction $a$, the resulting new state $s'$ is in $D$. $\{C\}\ a\ \{\mathit{False}\}$ means that no state exists after executing $a$ from $C$, i.e., the trace $a$ is infeasible. The inductiveness of the interpolants is due to the fact that each triple $\{C\}\ a\ \{D\}$ in the sequence (1) is a valid Hoare triple. Hoare triples (and validity) generalise to sequences of instructions $\sigma$ in the form $\{C\}\ \sigma\ \{D\}$.

  Because we can also prove that $\{I_1\}\ (t_1)^*\ \{I_1\}$ is a valid Hoare triple, if we combine this fact with Equation 1 we obtain a regular set of traces annotated with pre/post-conditions as per Equation 2.

$$\{\mathit{True}\} \quad i \quad \{I_0\} \quad t_0 \quad \mathbf{\{I_1\}} \quad \mathbf{(t_1)^*} \quad \mathbf{\{I_1\}} \quad t_2 \quad \{\mathit{False}\} \tag{2}$$

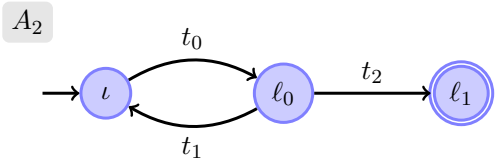
  This implies that the regular set of traces $i.t_0.(t_1)^*.t_2$ does not have any associated timed traces: for each word $w \in i.t_0.(t_1)^*.t_2$, $\tau(w) = \varnothing$ and as $\mathcal{L}(CFG(P_1)) \subseteq i.t_0.(t_1)^*.t_2$ we can conclude that $\mathcal{TL}(A_1) = \varnothing$.

***Example 2: Extended Timed Automaton.*** The following example (Figure 3) illustrates how extensions of timed automata with constraints that mix discrete and real variables can be analyzed. The real-time program $P_2$ (Figure 3) is given by the CFG (left) and the instructions[5] (right): it specifies a timed automaton with 2 clocks $x, y$ (real variables) and one integer variable $i$.

This is an extended version of timed automata as the constraint $y < i$ mixes integer and real variables (clocks) and this is not permitted in the standard definition of timed automata. Initially all the

[4] This is the pair returned by Z3 for $C_0 \wedge C_1 \wedge C_2$.

[5] The rates table is omitted as all the variables are clocks with rate 1.




Discrete Instructions

| Edge | Guard | Update |
|---|---|---|
| $t_0$ | $x \geq 1$ | - |
| $t_1$ | $True$ | `x:= 0; i := i + 1` |
| $t_2$ | $y < i$ | - |

Figure 3: Real-time program $P_2$: CFG $A_2$ of $P_2$ (left) with accepting location $\ell_1$ and its instructions (right).

variables are set to 0. The objective is to prove that location $\ell_1$ is unreachable and thus that $\mathcal{TL}(P_2) = \varnothing$. Note that UPPAAL does allow this specification but is unable to prove that $\ell_1$ is unreachable because $i$ is unbounded.

Our method is able to discover invariants that mix integer and real variables and can prove that $\ell_1$ is unreachable as follows:

1. the first iteration of the TAR algorithm starts with $L = \mathcal{L}(CFG(P_2))$. The check $L = \varnothing$ is negative as $w_1 = t_0.t_2 \in L$. However every timed word in $\tau(w_1)$ must satisfy the following constraints that correspond to taking $t_0$ and then $t_2$:

$$\underbrace{x_0 = y_0 = \delta_0 \wedge \delta_0 \geq 0 \wedge i_0 = 0}_{\text{Time elapsing } \delta_0 \text{ in } \iota} \wedge \underbrace{x_0 \geq 1}_{\text{Guard of } t_0} \qquad (C'_0)$$

$$\underbrace{x_1 = x_0 + \delta_1 \wedge y_1 = y_0 + \delta_1 \wedge i_1 = i_0 \wedge \delta_1 \geq 0}_{\text{Update of } t_0 \text{ and time elapsing } \delta_1 \text{ in } \ell_0} \wedge \underbrace{y_1 < i_1}_{\text{Guard of } t_2} \qquad (C'_1)$$

   $C'_0 \wedge C'_1$ is not satisfiable and hence $\mathcal{TL}(t_0.t_2) = \varnothing$ and thus we can safely remove $w_1$ from $L$. We can extract interpolants from the proof of unsatisfiability of $C'_0 \wedge C'_1$ and we establish the following sequence of valid Hoare triples:

$$\{x = y = i = 0\} \quad t_0 \quad \{x = y \wedge x \geq i\} \quad t_2 \quad \{False\} \qquad (3)$$

2. the second iteration of the TAR algorithm starts with an updated $L = \mathcal{L}(CFG(P_2)) \setminus \{w_1\}$. Again $L$ is not empty and for instance $w_2 = t_0.t_1.t_0.t_2$ is in $L$. The encoding for checking the emptiness of $\tau(w_2)$ is:

$$\underbrace{x_0 = y_0 = \delta_0 \wedge \delta_0 \geq 0 \wedge i_0 = 0}_{\text{Time elapsing } \delta_0 \text{ in } \iota} \wedge \underbrace{x_0 \geq 1}_{\text{Guard of } t_0} \qquad (C''_0)$$

$$\underbrace{x_1 = x_0 + \delta_1 \wedge y_1 = y_0 + \delta_1 \wedge i_1 = i_0 \wedge \delta_1 \geq 0}_{\text{Update of } t_0 \text{ and time elspsing } \delta_1 \text{ in } \ell_0} \wedge \underbrace{True}_{\text{Guard of } t_1} \qquad (C''_1)$$

$$\underbrace{x_2 = 0 + \delta_2 \wedge y_2 = y_1 + \delta_2 \wedge i_2 = i_1 + 1 \wedge \delta_2 \geq 0}_{\text{Time elapsing } \delta_2 \text{ in } \iota} \wedge \underbrace{x_2 \geq 1}_{\text{Guard of } t_0} \qquad (C''_2)$$

$$\underbrace{x_3 = x_2 + \delta_3 \wedge y_3 = y_2 + \delta_3 \wedge i_3 = i_2 \wedge \delta_3 \geq 0}_{\text{Time elapsing } \delta_3 \text{ in } \ell_0} \wedge \underbrace{y_3 < i_3}_{\text{Guard of } t_2} \qquad (C''_3)$$

$C_0'' \wedge C_1'' \wedge C_2'' \wedge C_3''$ is unsatisfiable and hence $\mathcal{TL}(t_0.t_1.t_0.t_2) = \varnothing$. We can extract interpolants from the proof of unsatisfiability and we establish the following sequence of valid Hoare triples.

$$\{x = y = i = 0\} \quad t_0 \quad \{y \geq i\} \quad t_1.t_0 \quad \{y \geq i\} \quad t_2 \quad \{\mathit{False}\} \tag{4}$$

As can be seen as $\{y \geq i\}\ t_1.t_0\ \{y \geq i\}$ holds we can generalize this sequence to an arbitrary number of iterations of $t_0.t_1$:

$$\{x = y = i = 0\} \quad t_0 \quad \boldsymbol{\{y \geq i\}} \quad \boldsymbol{(t_1.t_0)^+} \quad \boldsymbol{\{y \geq i\}} \quad t_2 \quad \{\mathit{False}\} \tag{5}$$

which entails that $\mathcal{TL}(t_0.(t_1.t_0)^+.t_2) = \varnothing$. This implies that we can remove $t_0.(t_1.t_0)^+.t_2$ from $L$.

3. observe that $L = \varnothing$ in the next iteration of TAR as $\mathcal{L}(CFG(P_2))\backslash(\{t_0.t_2\} \cup t_0.(t_1.t_0)^+.t_2) = \varnothing$ given that $\mathcal{L}(CFG(P_2)) = t_0.(t_1.t_0)^*.t_2$. We have thus proved that $\mathcal{TL}(P_2) = \varnothing$ as any word of instructions in $\mathcal{L}(CFG(P_2))$ induces an infeasible trace and the algorithm terminates.

In the rest of the paper, we provide a formal development of the methods we have introduced so far.

# 3. Real-time programs

Our approach is general enough and applicable to a wide range of timed systems called *real-time programs*. As an example, timed, stopwatch, hybrid automata and time Petri nets are special cases of real-time programs.

In this section we formally define *real-time programs*. Real-time programs specify the control flow of *instructions*, just as standard imperative programs do. The instructions can update *variables* by assigning new values to them. Each instruction has a semantics and together with the control flow this precisely defines the semantics of real-time programs.

## 3.1. Notations

A finite automaton over an alphabet $\Sigma$ is a tuple $\mathcal{A} = (Q, \iota, \Sigma, \Delta, F)$ where $Q$ is a finite set of locations s.t. $\iota \in Q$ is the initial location, $\Sigma$ is a finite alphabet of actions, $\Delta \subseteq (Q \times \Sigma \times Q)$ is a finite transition relation, $F \subseteq Q$ is the set of *accepting* locations. A word $w = \alpha_0.\alpha_1.\cdots.\alpha_n$ is a finite sequence of letters from $\Sigma$; we let $w[i] = \alpha_i$ be the $i$-th letter of $w$, $|w|$ be the length of $w$ which is $n+1$. Let $\epsilon$ be the empty word and $|\epsilon| = 0$, and let $\Sigma^*$ be the set of finite words over $\Sigma$. The *language*, $\mathcal{L}(\mathcal{A})$, accepted by $\mathcal{A}$ is defined in the usual manner as the set of words that can lead to $F$ from $\iota$.

Let $V$ be a finite set of real-valued variables. A *valuation* is a function $\nu : V \to \mathbb{R}$. The set of valuations is $[V \to \mathbb{R}]$.

We denote by $\beta(V)$ the set of *constraints* (or Boolean predicates) over $V$ and given $\varphi \in \beta(V)$, we let *Vars*$(\varphi)$ be the set of unconstrained variables in $\varphi$. Given a valuation, we let the truth value of a constraint (Boolean predicate) $\varphi$ be denoted by $\varphi(\nu) \in \{\mathit{True}, \mathit{False}\}$, and write $\nu \models \varphi$ when $\varphi(\nu) = \mathit{True}$ and let $[\![\varphi]\!] = \{\nu \mid \nu \models \varphi\}$.

An *update* $\mu \subseteq [V \to \mathbb{R}] \times [V \to \mathbb{R}]$ is a binary relation over valuations. Given an update $\mu$ and a set of valuations $\mathcal{V}$, we let $\mu(\mathcal{V}) = \{\nu' \mid \exists \nu \in \mathcal{V} \text{ and } (\nu, \nu') \in \mu\}$. We let $\mathcal{U}(V)$ be the set of updates on the variables in $V$.

Similar to the update relation, we define a *rate* function $\rho : V \to \mathbb{R}$ (rates can be negative), i.e., a function from a variable to a real number[6]. A rate is then a vector $\rho \in \mathbb{R}^V$. Given a valuation $\nu$ and a timestep $\delta \in \mathbb{R}_{\geq 0}$ the valuation $\nu + (\rho, \delta)$ is defined by: $(\nu + (\rho, \delta))(v) = \nu(v) + \rho(v) \times \delta$ for $v \in V$.

## 3.2. Real-time instructions

Let $\Sigma = \beta(V) \times \mathcal{U}(V) \times \mathcal{R}(V)$ be a countable set of instructions – and intentionally also the alphabet of the CFG. Each $\alpha \in \Sigma$ is a tuple (*guard, update, rates*) denoted by $(\gamma_\alpha, \mu_\alpha, \rho_\alpha)$. Let $\nu : V \to \mathbb{R}$ and $\nu' : V \to \mathbb{R}$ be two valuations. For each pair $(\alpha, \delta) \in \Sigma \times \mathbb{R}_{\geq 0}$ we define the following transition relation $\xrightarrow{\alpha,\delta}$:

$$\nu \xrightarrow{\alpha,\delta} \nu' \iff \begin{cases} 1. & \nu \models \gamma_\alpha \text{ (guard of } \alpha \text{ is satisfied in } \nu), \\ 2. & \exists \nu'' \text{ s.t. } (\nu, \nu'') \in \mu_\alpha \text{ (discrete update allowed by } \alpha) \text{ and} \\ 3. & \nu' = \nu'' + (\rho_\alpha, \delta) \text{ (continuous update as defined by } \alpha). \end{cases}$$

The semantics of $\alpha \in \Sigma$ is a mapping $[\![\alpha]\!] : [V \to \mathbb{R}] \to 2^{[V \to \mathbb{R}]}$ and for $\nu \in [V \to \mathbb{R}]$

$$[\![\alpha]\!](\nu) = \{\nu' \mid \exists \delta \geq 0, \nu \xrightarrow{\alpha,\delta} \nu'\}. \tag{6}$$

It follows that:

**Fact 3.1.** $\exists \delta \geq 0, \nu \xrightarrow{\alpha,\delta} \nu' \iff \nu' \in [\![\alpha]\!](\nu)$.

This mapping can be straightforwardly extended to sets of valuations $K \subseteq [V \to \mathbb{R}]$ as follows:

$$[\![\alpha]\!](K) = \bigcup_{\nu \in K} [\![\alpha]\!](\nu). \tag{7}$$

## 3.3. Post operator

Let $K$ be a set of valuations and $w \in \Sigma^*$. We inductively define the *(strongest) post operator* $\mathit{Post}(K, w)$ as follows:

$$\begin{aligned} \mathit{Post}(K, \epsilon) &= K \\ \mathit{Post}(K, \alpha.w) &= \mathit{Post}([\![\alpha]\!](K), w) \end{aligned}$$

The post operator extends to logical constraints $\varphi \in \beta(V)$ by defining $\mathit{Post}(\varphi, w) = \mathit{Post}([\![\varphi]\!], w)$. In the sequel, we assume that, when $\varphi \in \beta(V)$, then $[\![\alpha]\!]([\![\varphi]\!])$ is also definable as a constraint in $\beta(V)$. This inductively implies that $\mathit{Post}(\varphi, w)$ can also be expressed as a constraint in $\beta(V)$ for sequences of instructions $w \in \Sigma^*$.

[6] We can allow rates to be arbitrary terms but in this paper we restrict to deterministic rates or bounded intervals.

### 3.4. Timed words and feasible words

A *timed word* (over alphabet $\Sigma$) is a finite sequence $\sigma = (\alpha_0, \delta_0).(\alpha_1, \delta_1).\cdots.(\alpha_n, \delta_n)$ such that for each $0 \leq i \leq n$, $\delta_i \in \mathbb{R}_{\geq 0}$ and $\alpha_i \in \Sigma$. The timed word $\sigma$ is *feasible* if and only if there exists a set of valuations $\{\nu_0, \ldots, \nu_{n+1}\} \subseteq [V \to \mathbb{R}]$ such that:

$$\nu_0 \xrightarrow{\alpha_0,\delta_0} \nu_1 \xrightarrow{\alpha_1,\delta_1} \nu_2 \quad \cdots \quad \nu_n \xrightarrow{\alpha_n,\delta_n} \nu_{n+1}.$$

We let $\mathit{Unt}(\sigma) = \alpha_0.\alpha_1.\cdots.\alpha_n$ be the *untimed* version of $\sigma$. We extend the notion *feasible* to an untimed word $w \in \Sigma^*$: $w$ is feasible iff $w = \mathit{Unt}(\sigma)$ for some feasible timed word $\sigma$.

**Lemma 3.2.** An untimed word $w \in \Sigma^*$ is *feasible* iff $\mathit{Post}(\mathit{True}, w) \neq \mathit{False}$.

**Proof:**
We prove this Lemma by induction on the length of $w$. The induction hypothesis is:

$$\nu_0 \xrightarrow{\alpha_0,\delta_0} \nu_1 \xrightarrow{\alpha_1,\delta_1} \nu_2 \quad \cdots \quad \nu_n \xrightarrow{\alpha_n,\delta_n} \nu_{n+1} \iff \nu_{n+1} \in \mathit{Post}(\{\nu_0\}, \alpha_0.\alpha_1.\cdots.\alpha_n)$$

which is enough to prove the Lemma.

*Base step.* If $w = \epsilon$, then $\mathit{Post}(\{\nu_0\}, \epsilon) = \{\nu_0\}$.

*Inductive step.* Assume $\nu_0 \xrightarrow{\alpha_0,\delta_0} \nu_1 \xrightarrow{\alpha_1,\delta_1} \nu_2 \quad \cdots \quad \nu_n \xrightarrow{\alpha_n,\delta_n} \nu_{n+1} \xrightarrow{\alpha_{n+1},\delta_{n+1}} \nu_{n+2}$. By induction hypothesis, $\nu_{n+1} \in \mathit{Post}(\{\nu_0\}, \alpha_0.\alpha_1.\cdots.\alpha_n)$, and $\nu_{n+2} \in [\![\alpha_{n+1}]\!](\nu_{n+1})$. By definition of *Post* this implies that $\nu_{n+2} \in \mathit{Post}(\{\nu_0\}, \alpha_0.\alpha_1.\cdots.\alpha_n.\alpha_{n+1})$. □

### 3.5. Real-time programs

The specification of a real-time program decouples the *control* (e.g., for Timed Automata, the locations) and the *data* (the clocks or integer variables). A *real-time program* is a pair $P = (A_P, [\![\cdot]\!])$ where $A_P$ is a finite automaton $A_P = (Q, \iota, \Sigma, \Delta, F)$ over the alphabet[7] $\Sigma$, $\Delta$ defines the control-flow graph of the program and $[\![\cdot]\!]$ provides the semantics of each instruction.

A timed word $\sigma$ is *accepted* by $P$ if and only if:

1. $\mathit{Unt}(\sigma)$ is accepted by $A_P$ and,
2. $\sigma$ is feasible.

The *timed language*, $\mathcal{TL}(P)$, of a real-time program $P$ is the set of timed words accepted by $P$, i.e., $\sigma \in \mathcal{TL}(P)$ if and only if $\mathit{Unt}(\sigma) \in \mathcal{L}(A_P)$ and $\sigma$ is feasible.

**Remark 3.3.** We do not assume any particular values initially for the variables of a real-time program (the variables that appear in $I$). This is reflected by the definition of *feasibility* that only requires the existence of valuations without containing the initial one $\nu_0$. When specifying a real-time program, initial values can be explicitly set by regular instructions at the beginning of the program. This is similar to standard programs where the first instructions can set the values of some variables.

[7] $\Sigma$ can be infinite but we require the control-flow graph $\Delta$ (transition relation) of $A_P$ to be finite.

### 3.6. Timed language emptiness problem

The *(timed) language emptiness problem* asks the following:

Given a real-time program $P$, is $\mathcal{TL}(P)$ empty?

**Theorem 3.4.** $\mathcal{TL}(P) \neq \varnothing$ iff $\exists w \in \mathcal{L}(A_P)$ such that *Post*$(True, w) \not\subseteq False$.

**Proof:**
$\mathcal{TL}(P) \neq \varnothing$ iff there exists a feasible timed word $\sigma$ such that *Unt*$(\sigma)$ is accepted by $A_P$. This is equivalent to the existence of a feasible word $w \in \mathcal{L}(A_P)$, and by Lemma 3.2, feasibility of $w$ is equivalent to *Post*$(True, w) \not\subseteq False$. □

### 3.7. Useful classes of real-time programs

*Timed Automata* are a special case of real-time programs. The variables are called clocks. $\beta(V)$ is restricted to constraints on individual clocks or *difference constraints* generated by the grammar:

$$b_1, b_2 ::= True \mid False \mid x - y \bowtie k \mid x \bowtie k \mid b_1 \wedge b_2 \tag{8}$$

where $x, y \in V$, $k \in \mathbb{Q}_{\geq 0}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$[8]. We note that wlog. we omit *location invariants* as for the language emptiness problem, these can be implemented as guards. An update in $\mu \in \mathcal{U}(V)$ is defined by a set of clocks to be *reset*. Each pair $(\nu, \nu') \in \mu$ is such that $\nu'(x) = \nu(x)$ or $\nu'(x) = 0$ for each $x \in V$. The valid rates are fixed to 1, and thus $\mathcal{R}(V) = \{1\}^V$.

*Stopwatch Automata* can also be defined as a special case of real-time programs. As defined in [8], Stopwatch Automata are Timed Automata extended with *stopwatches* which are clocks that can be stopped. $\beta(V)$ and $\mathcal{U}(V)$ are the same as for Timed Automata but the set of valid rates is defined by the functions of the form $\mathcal{R}(V) = \{0, 1\}^V$ (the clock rates can be either 0 or 1). An example of a Stopwatch Automaton is given by the timed system $\mathcal{A}_1$ in Figure 1.

As there exists syntactic translations (preserving timed languages or reachability) that map hybrid automata to stopwatch automata [8], and translations that map time Petri nets [26, 27] and extensions [28, 29] thereof to timed automata, it follows that time Petri nets and hybrid automata are also special cases of real-time programs. This shows that the method we present in the next section is applicable to a wide range of timed systems. What is remarkable as well, is that it is not restricted to timed systems that have a finite number of discrete states but can also accommodate infinite discrete state spaces. For example, the real-time program $P_2$ in Figure 3, page 8 has two clocks $x$ and $y$ and an unbounded integer variable $i$. Even though $i$ is unbounded, our technique discovers the loop invariant $y \geq i$ of the $\iota$ and $\ell_0$ locations – an invariant is over a real-time clock $y$ and the integer variable $i$. It allows us to prove that $\mathcal{TL}(P_2) = \varnothing$ as the guard of $t_2$ never can be satisfied ($y < i$).

[8] While difference constraints are strictly disallowed in most definitions of Timed Automata, the method we propose retain its properties regardless of their presence.

# 4. Trace abstraction refinement for real-time programs

In this section we give a formal description of a semi-algorithm to solve the language emptiness problem for real-time programs. The semi-algorithm is a version of the *refinement of trace abstractions* (TAR) approach [3] for timed systems.

## 4.1. Refinement of trace abstraction for real-time programs

We have already introduced our algorithm in Figure 2, page 6. We now give a precise formulation of the TAR semi-algorithm for real-time programs, in Algorithm 1. It is essentially the same as the semi-algorithm as introduced in [3] – we therefore omit theorems of completeness and soundness as these will be equivalent to the theorems in [3] and are proved in the exact same manner.

**Algorithm 1:** RTTAR – Trace Abstraction Refinement for Real-Time Programs

**Input :** A real-time program $P = (A_P, [\![\cdot]\!])$.
**Result:** $(\mathit{True}, -)$ if $\mathcal{TL}(P) = \varnothing$, and otherwise $(\mathit{False}, w)$ if $\mathcal{TL}(P) \neq \varnothing$ with $w \in \mathcal{L}(A_P)$ and $\mathit{Post}(\mathit{True}, w) \not\subseteq \mathit{False}$ – or non-termination.
**Var :** $R$: a regular language, initially $R = \varnothing$.
$w$: a word in $\mathcal{L}(A_P)$, initially $w = \epsilon$.
$T$: A finite automaton, initially empty.

```
1 while L(A_P) ⊈ R do
2     Let w ∈ L(A_P) \ R;
3     if Post(True, w) ⊈ False then
          /* w is feasible and w is a counter-example */
4         return (False, w);
5     else
          /* w is infeasible, compute an interpolant automaton based on w */
6         Let T = ITA(w);
          /* Add T to refinement and continue */
7         Let R := R ∪ L(T);
8 return (True, −);
```

The input to the semi-algorithm *TAR-RT* is a real-time program $P = (A_P, [\![\cdot]\!])$. An invariant of the semi-algorithm is that the refinement $R$, which is subtracted to the initial set of traces, is either empty or containing infeasible traces only. In the coarsets, initial abstraction, all the words $\mathcal{L}(A_P)$ are potentially feasible. In each iteration of the algorithm, we then chip away infeasible behaviour (via the set $R$) of $A_P$, making the set difference $\mathcal{L}(A_P) \setminus R$ move closer to the set of feasible traces, thereby shrinking the overapproximation of feasible traces ($\mathcal{L}(A_P) \setminus R$).

Initially the refinement $R$ is the empty set. The semi-algorithm works as follows:

**Step 1** line 1, check whether all the (untimed) traces in $\mathcal{L}(A_P)$ are in $R$. If this is the case, $\mathcal{TL}(P)$ is empty and the semi-algorithm terminates (line 8). Otherwise (line 2), there is a sequence $w \in \mathcal{L}(A_P) \setminus R$, goto Step 2;

**Step 2** if $w$ is feasible (line 3) i.e., there is a feasible timed word $\sigma$ such that $\mathit{Unt}(\sigma) = w$, then $\sigma \in \mathcal{TL}(P)$ and $\mathcal{TL}(P) \neq \varnothing$ and the semi-algorithm terminates (line 4). Otherwise $w$ is not feasible, goto Step 3;

**Step 3** $w$ is infeasible and given the reason for infeasibility we can construct (line 6) a finite *interpolant automaton*, $\mathit{ITA}(w)$, that accepts $w$ and other words that are infeasible for the same reason. How $\mathit{ITA}(w)$ is computed is addressed in the sequel. The automaton $\mathit{ITA}(w)$ is added (line 7) to the previous refinement $R$ and the semi-algorithm starts a new round at Step 1 (line 1).

In the next paragraphs we explain the main steps of the algorithms: how to check feasibility of a sequence of instructions and how to build $\mathit{ITA}(w)$.

## 4.2. Checking feasibility

Given a arbitrary word $w \in \Sigma^*$, we can check whether $w$ is feasible by encoding the side-effects of each instruction in $w$ using linear arithmetic as demonstrated in Examples 1 and 2.

We now define a function *Enc* for constructing such a constraint-system characterizing the feasibility of a given trace. We first show how to encode the side-effects and feasibility of a single instruction $\alpha \in \Sigma$. Recall that $\alpha = (\gamma, \mu, \rho)$ where the three components are respectively the guard, the update, and the rates. Assume that the variables[9] in $\alpha$ are $X = \{x_1, x_2, \cdots, x_k\}$. We can define the semantics of $\alpha$ using the standard unprimed[10] and primed variables ($X'$). We assume that the guard and the updates can be defined by predicates and write $\alpha = (\varphi(\overline{x}), \mu(\overline{x}, \overline{x}'), \rho(\overline{x}))$ with:

- $\varphi(\overline{x}) \in \beta(X)$ is the guard of the instruction,
- $\mu(\overline{x}, \overline{x}')$ a set of constraints in $\beta(X \cup X')$,
- $\rho : X \to \mathbb{Q}$ defines the rates of the variables.

The effect of $\alpha$ from a valuation $\overline{x}''$, which is composed of 1) discrete step if the guard is true followed by the updates leading to a new valuation $\overline{x}'$, and 2) continuous step i.e., time elapsing $\delta$, leading to a new valuation $\overline{x}$, can be encoded as follows:

$$\mathit{Enc}(\alpha, \overline{x}'', \overline{x}', \overline{x}, \delta) = \varphi(\overline{x}'') \wedge \mu(\overline{x}'', \overline{x}') \wedge \overline{x} = \overline{x}' + (\rho, \delta) \wedge \delta \geq 0 \tag{9}$$

Let $K(\overline{x})$ be a set of valuations that can be defined as constraint in $\beta(X)$. It follows that $[\![\alpha]\!](K(\overline{x}))$ is defined by:

$$\exists \delta, \overline{x}'', \overline{x}' \text{ such that } K(\overline{x}'') \wedge \mathit{Enc}(\alpha, \overline{x}'', \overline{x}', \overline{x}, \delta) \tag{10}$$

In other terms, $[\![\alpha]\!](K(\overline{x}))$ is not empty iff $K(\overline{x}'') \wedge \mathit{Enc}(\alpha, \overline{x}'', \overline{x}', \overline{x}, \delta)$ is *satisfiable*.

We can now define the encoding of a sequence of instructions $w = \alpha_0.\alpha_1.\cdots.\alpha_n \in \Sigma^*$. Given a set of variables $W$, we define the corresponding set of super-scripted variables $W^k = \{w^j, w \in W, 0 \leq$

[9] The union of the variables in $\gamma, \mu, \rho$.

[10] $\overline{x}$ denotes the vector of variables $\{x_1, x_2, \cdots, x_k\}$.

$j \leq k\}$. Instead of using $x, x', x''$ we use super-scripted variables $\overline{x}^k$ (and $\overline{y}^k$ for the intermediate variables $x'$) to encode the side-effect of each instruction in the trace:

$$\mathit{Enc}(w) = \bigwedge_{i=0}^{n} \mathit{Enc}(\alpha_i, \overline{x}^i, \overline{y}^i, \overline{x}^{i+1}, \delta^i)$$

It is straighgforward to prove that the function $\mathit{Enc} : \Sigma^* \to \beta(X^{n+1} \cup Y^n \cup \{\delta\}^n)$ constructs a constraint-system characterizing exactly the feasibility of a word $w$:

**Fact 4.1.** For each $w \in \Sigma^*$, $\mathit{Post}(\mathit{True}, w) \not\subseteq \mathit{False}$ iff $\mathit{Enc}(w)$ is satisfiable.

If the terms we build are in a logic supported by SMT-solvers (e.g., Linear Real Arithmetic) we can automatically check satisfiability. If $\mathit{Enc}(w)$ is satisfiable we can even collect some *model* which provides witness values for the $\delta_k$. Otherwise, if $\mathit{Enc}(w)$ is unsatisfiable, there are some options to collect *some reasons for unsatisfiability*: unsat cores or interpolants. The latter is discussed in the next section.

An example of an encoding for the real-time program $P_1$ (Figure 1) and the sequence $w_1 = i.t_0.t_2$ is given by the predicates in Equation $(C_0)$–$(C_2)$. Hence the sequence $w_1 = i.t_0.t_2$ is feasible iff $\mathit{Enc}(w_1) = C_0 \wedge C_1 \wedge C_2$ is satisfiable. Using a SMT-solver, e.g., with Z3, we can confirm that $\mathit{Enc}(w_1)$ is unsatisfiable. The interpolating[11] solver Z3 can also generate a sequence of interpolants, $I_0 = x \leq y$ and $I_1 = x - y \leq z$, that provide a general reason for unsatisfiability and satisfy:

$$\{\mathit{True}\} \quad i \quad \{I_0\} \quad t_0 \quad \{I_1\} \quad t_2 \quad \{\mathit{False}\}.$$

We can use the interpolants to build interpolant automata as described in the next section.

## 4.3. Construction of interpolant automata

### 4.3.1. Inductive interpolant

When it is determined that a trace $w$ is infeasible, we can easily discard such a single trace and continue searching for a different one. However, the power of the TAR method is to generalize the infeasibility of a single trace $w$ into a family (regular set) of traces. This regular set of infeasible traces is computed from *a reason of infeasibility* of $w$ and is formally specified by an *interpolant automaton*, $\mathit{ITA}(w)$. The reason for infeasibility itself can be the predicates obtained by computing strongest post-conditions or weakest-preconditions or anything in between but it must be an *inductive interpolant*[12].

Given a conjunctive formula $f = C_0 \wedge \cdots \wedge C_m$, if $f$ is unsatisfiable, an *inductive interpolant* is a sequence of predicates $I_0, \ldots, I_{m-1}$ s.t:

- $\mathit{True} \wedge C_0 \implies I_0$,

[11] The interpolating feature of Z3 has been phased out from version 4.6.x. However, there are alternative techniques to obtain inductive interpolants e.g., using unsat cores [30].

[12] Strongest post-conditions and weakest pre-conditions can provide inductive interpolants

- $I_{m-1} \wedge C_m \implies \mathit{False}$,
- For each $0 \leq n < m-1$, $I_n \wedge C_{n+1} \implies I_{n+1}$, and the variables in $I_n$ appear in both $C_n$ and $C_{n+1}$ i.e., $\mathit{Vars}(I_n) \subseteq \mathit{Vars}(C_n) \cap \mathit{Vars}(C_{n+1})$.

If the predicates $C_0, C_1, \cdots, C_m$ encode the side effects of a sequence of instructions $\alpha_0.\alpha_1.\cdots,\alpha_m$, then one can intuitively think of each interpolant as a *sufficient* condition for infeasibility of the post-fix of the trace and this can be represented by a sequence of valid Hoare triples of the form $\{C\}\ a\ \{D\}$:

$$\{\mathit{True}\} \quad \alpha_0 \quad \{I_0\} \quad \alpha_1 \quad \{I_1\} \quad \cdots \quad \{I_{m-1}\} \quad \alpha_m \quad \{\mathit{False}\}$$

Consider the real-time program $P_3$ of Figure 3 and the two infeasible untimed words $w_1 = i.t_0.t_2$ and $w_2 = i.t_0.t_1.t_0.t_2$. Some inductive interpolants for $w_1$ and $w_2$ can be given by: $I_0 = y_0 \geq x_0 \wedge (k_0 = 0)$, $I_1 = y_1 \geq k_1$ for $w_1$ and $I'_0 = y_0 \geq x_0 \wedge k_0 \leq 0$, $I'_1 = y_1 \geq 1 \wedge k_1 \leq 0$, $I'_2 = y_2 \geq k_2 + x_2$, $I'_3 = y_3 \geq k_3 + 1$ for $w_2$. From the inductive interpolants one can obtain valid Hoare triples by de-indexing the predicates in the inductive interpolants[13] as shown in Equations 11-12:

$$\{\mathit{True}\} \quad i \quad \{\pi(I_0)\} \quad t_0 \quad \{\pi(I_1)\} \quad t_2 \quad \{\mathit{False}\} \tag{11}$$

$$\{\mathit{True}\} \quad i \quad \{\pi(I'_0)\} \quad t_0 \quad \{\pi(I'_1)\} \quad t_1 \quad \{\pi(I'_2)\} \quad t_0 \quad \{\pi(I'_3)\} \quad t_2 \quad \{\mathit{False}\} \tag{12}$$

where $\pi(I_k)$ is the same as $I_k$ where each indexed variable $x_j$ replaced by $x$. As can be seen in Equation 12, the sequence contains two occurrences of $t_0$: this suggests that a loop occurs in the program, and this loop may be infeasible as well. Formally, because $\mathit{Post}(\pi(I'_2), t_0) \subseteq I'_1$, any trace of the form $i.t_0.t_1.(t_0.t_1)^*.t_0.t_2$ is infeasible. This enables us to construct an interpolant automaton $\mathit{ITA}(w_2)$ accepting the regular set of infeasible traces $i.t_0.t_1.(t_0.t_1)^*.t_0.t_2$. Overall, because $w_1$ is also infeasible, the *union* of the languages accepted by $\mathit{ITA}(w_2)$ and $\mathit{ITA}(w_1)$ is a set of infeasible traces as defined by the finite automaton in Figure 4.

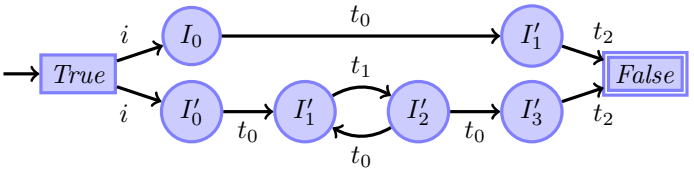



Figure 4: Interpolant automaton for $\mathcal{L}(\mathit{ITA}(w_1)) \cup \mathcal{L}(\mathit{ITA}(w2))$.

Given $w$ such that $\mathit{Enc}(w)$ is unsatisfiable we can always find an inductive interpolant: the strongest post-conditions $\mathit{Post}(\mathit{True}, w[i])$ or (the weakest pre-conditions from $\mathit{False}$) defines an inductive interpolant. More generally, we have:

**Lemma 4.2.** Let $w = \alpha_0.\alpha_1.\cdots.\alpha_m \in \Sigma^*$. If $\mathit{Enc}(w) = C_0 \wedge C_1 \wedge \cdots \wedge C_m$ is unsatisfiable and $I_0, \cdots, I_{m-1}$ is an inductive interpolant for $\mathit{Enc}(w)$, the following sequence of Hoare triples

$$\{\mathit{True}\} \quad \alpha_0 \quad \{\pi(I_0)\} \quad \alpha_1 \quad \{\pi(I_1)\} \quad \cdots \quad \alpha_{m-1} \quad \{\pi(I_{m-1})\} \quad \alpha_m \quad \{\mathit{False}\}$$

is valid.

[13] This is a direct result of the encoding function *Enc*. The interpolants can only contain at most one version of each indexed variables.

**Proof:**
The proof follows from the encoding $\mathit{Enc}(w)$ and the fact that each $I_k$ is included in the weakest pre-condition $wp(\mathit{False}, \alpha_{k+1}.\alpha_m)$ which can be proved by induction using the property of inductive interpolants. □

#### 4.3.2. Interpolant automata

Let us formalize the interpolant-automata construction. Let $w = \alpha_0.\alpha_1.\cdots.\alpha_m \in \Sigma^*$, $\mathit{Enc}(w) = C_0 \wedge \cdots \wedge C_m$ and assume $\mathit{Post}(\mathit{True}, w) \subseteq \mathit{False}$ i.e., $\mathit{Enc}(w)$ is unsatisfiable (Fact 4.1).

Let $I_0, \ldots I_{m-1}$ be an inductive interpolant for $C_0 \wedge \cdots \wedge C_m$. We can construct an interpolant automaton for $w$, $\mathit{ITA}(w) = (Q^w, q_0^w, \Sigma^w, \Delta^w, F^w)$ as follows:

- $Q^w = \{\mathit{True}, \mathit{False}, \pi(I_0), \cdots, \pi(I_{m-1})\}$, (note that if two de-indexed interpolants are the same they account for one state only),
- $\Sigma^w = \{\alpha_0, \alpha_1, \cdots, \alpha_m\}$,
- $F^w = \{\mathit{False}\}$,
- $\Delta^w$ satisfies following conditions:
  1. $(\mathit{True}, \alpha_0, \pi(I_0)) \in \Delta^w$,
  2. $(\pi(I_{m-1}), \alpha_m, \mathit{False}) \in \Delta^w$,
  3. $\forall a \in \Sigma^w, \forall 0 \leq k, j \leq m-1$, if $\mathit{Post}(\pi(I_k), a) \subseteq \pi(I_j)$ then $(\pi(I_k), a, \pi(I_j)) \in \Delta^w$.

Notice that as $\mathit{Post}(\pi(I_k), \alpha_{k+1}) \subseteq \pi(I_{k+1})$ the word $w$ itself is accepted by $\mathit{ITA}(w)$ and $\mathit{ITA}(w)$ is never empty.

**Theorem 4.3. (Interpolant Automata)**
Let $w$ be an infeasible word over $P$, then for all $w' \in \mathcal{L}(\mathit{ITA}(w))$, $w'$ is infeasible.

**Proof:**
This proof is essentially the same as the original one in [3]. The proof uses rule 3 in the construction of $\mathit{ITA}(w)$: every word accepted by $\mathit{ITA}(w)$ goes through a sequence of states that form a sequence of valid Hoare triples and end up in $\mathit{False}$. It follows that if $w' \in \mathit{ITA}(w)$, $\mathit{Post}(\mathit{True}, w') \subseteq \mathit{False}$. □

### 4.4. Union of interpolant automata

In the TAR algorithm we construct interpolant automata at each iteration and the current refinement $R$ is the *union* of the regular languages $\mathcal{L}(\mathit{ITA}(w_k))$ for each infeasible $w_k$. The union can be computed using standard automata-theoretic operations. This assumes that we somehow *forget* the predicates associated with each state of an interpolant automaton.

In this section we introduce a new technique to re-use the information computed in each $\mathit{ITA}(w_k)$ and obtain larger refinements.

Let $A = (Q, q_0, \Sigma, \Delta, F)$ be a finite automaton such that each $q \in Q$ is a predicate in $\varphi(X)$. We say that $A$ is *sound* if the transition relation $\Delta$ satisfies: $(I, \alpha, J) \in \Delta$ implies that $[\![\alpha]\!](I) \subseteq J$ (or $\mathit{Post}(I, \alpha) \subseteq J$).

Let $R = (Q^R, \{True\}, \Sigma^R, \Delta^R, \{False\})$ be a sound finite automaton that accepts only infeasible traces. Let $w \in \Sigma^*$ with $w$ infeasible. The automaton $\mathit{ITA}(w) = (Q^w, \{True\}, \Sigma^w, \Delta^w, \{False\})$ built as described in section 4.3 is sound. We can define an *extended union*, $R \uplus \mathit{ITA}(w) = (Q^R \cup Q^w, \{True\}, \Sigma^R \cup \Sigma^w, \Delta^{R \uplus \mathit{ITA}(w)}, \{False\})$ of $R$ and $\mathit{ITA}(w)$ with:

$$\Delta^{R \uplus \mathit{ITA}(w)} = \{(p, \alpha, p')\} \mid \exists (q, \alpha, q') \in \Delta^R \cup \Delta^w \text{ s.t.} p \subseteq q \text{ and } p' \supseteq q'\}.$$

It is easy to see that $\mathcal{L}(R \uplus \mathit{ITA}(w)) \supseteq \mathcal{L}(R) \cup \mathcal{L}(\mathit{ITA}(w))$ but also:

**Theorem 4.4.** Let $w' \in \mathcal{L}(R \uplus \mathit{ITA}(w))$. Then $\mathit{Post}(True, w') \subseteq False$.

**Proof:**
Each transition $(p, \alpha, p')$ in $R \uplus \mathit{ITA}(w)$ corresponds to a valid Hoare triple. It is either in $\Delta^R$ or $\Delta^w$ and then is valid by construction or it is weaker than an established Hoare triple in $\Delta^R$ or $\Delta^w$. □

This theorem allows us to use the $\uplus$ operator in Algorithm 1 instead of the standard union of regular languages. The advantage is that we re-use already established Hoare triples to build a larger refinement at each iteration.

## 4.5. Feasibility beyond timed automata

Satisfiability can be checked with an SMT-solver (and decision procedures exist for useful theories). In the case of timed automata and stopwatch automata, the feasibility of a trace can be encoded in linear arithmetic. The corresponding theory, Linear Real Arithmetic (LRA) is decidable and supported by most SMT-solvers. It is also possible to encode non-linear constraints (non-linear guards and assignments). In the latter cases, the SMT-solver may not be able to provide an answer to the SAT problem as non-linear theories are undecidable. However, we can still build on a semi-decision procedure of the SMT-solver, and if it provides an answer, get the status of a trace (feasible or not).

## 4.6. Sufficient conditions for termination

Let us now construct a set of criteria on a real-time program $P = ((Q, q_0, \Sigma, \Delta, F), [\![\cdot]\!])$ s.t. our proposed method is guaranteed to terminate.

**Lemma 4.5.** Termination The algorithm presented in Figure 2 terminates if the following three conditions hold.

1. For any word $\sigma \in \Sigma^*$, then $[\![\sigma]\!]$ is expressible within a decidable theory (supported by the solver), and
2. the statespace of $P$ has a finite representation, and
3. the solver used returns interpolants within the finite statespace representation.

**Proof:**
First consider the algorithm presented in Figure 2, then we can initially state that for each iteration of the loop $R$ grows and thus the NFA representing $R$ ($\mathcal{A}^R$) must also. As per the construction presented in Section 4.4 we can observe that the transition-function of $\mathcal{A}^R$ will increase by at least one in each iteration in Step 3. If not, the selection of $\sigma$ between step 1 and step 2 is surely violated or the construction of *ITA* in step 3 is.

From Conditions 2 and 3 we have that the statespace is finitely representable and that these representatives are used by the solver. Thus we know that the interpolant automata also has a finite set of states as per the construction of Section 4.4. Together with the finiteness of the set of instructions, this implies that the transition-function of the interpolant automata must also be finite. Hence, the algorithm can (at most) introduce a transition between each pair of states with each instruction, but must at least introduce a new one in every iteration. □

As this termination condition relies on the solver, it is heavily dependent on the construction of the solver. However, if we consider the class of real-time programs captured by Timed Automata, we know that condition 1 is satisfied (in fact it is Linear Real Arithmetic), condition 2 is satisfied via the region-graph construction. This leaves the construction of a solver satisfying condition 3, which in turn should be feasible already from condition 2, but is practically achievable for TA via extrapolation-techniques and difference bound matrices (or for systems with only non-strict guards; timed-darts or integer representatives).

# 5. Parameter synthesis for real-time programs

In this section we show how to use the trace abstraction refinement semi-algorithm presented in Section 4 to synthesize *good initial values* for some of the program variables, and to check *robustness* of timed automata. We first define the *Maximal Safe Initial State* problem and then show how to reduce parameter synthesis and robustness to special cases of this problem.

## 5.1. Maximal safe initial set problem

Given a real-time program $P$, the objective is to determine a set of *initial valuations* $I \subseteq [V \to \mathbb{R}]$ such that, when we start the program in $I$, $\mathcal{TL}(P)$ is empty.

Given a constraint $I \in \beta(V)$, we define the corresponding *assume* instruction by: $\mathit{Assume}(I) = (I, \mathit{Id}, \overline{0})$. This instruction leaves all the variables unchanged (discrete update is the identity function and the rate vector is $\overline{0}$) and this acts as a guard only.

Let $P = (Q, q_0, \Sigma, \Delta, F)$ be a real-time program and $I \in \beta(V)$. We define the real-time program $\mathit{Assume}(I).P = (Q, \{\iota\}, \Sigma \cup \{\mathit{Assume}(I)\}, \Delta \cup \{(\iota, \mathit{Assume}(I), q_0)\}, F)$.

The ***maximal safe initial state problem*** asks the following:

**Given a real-time program $P$, find a maximal $I \in \beta(V)$ s.t. $\mathcal{TL}(\mathit{Assume}(I).P) = \varnothing$.**

## 5.2. Semi-algorithm for the maximal safe initial state problem

Let $w \in \mathcal{L}(\mathit{Assume}(I).P)$ be a feasible word. It follows that $\mathit{Enc}(w)$ must be satisfiable. We can define the set of initial values for which $\mathit{Enc}(w)$ is satisfiable by projecting away all the variables in the encoding $\mathit{Enc}(w)$ except the ones indexed by 0. Let $I_0 = \exists(\mathit{Vars}(\mathit{Enc}(w)) \setminus X^0).\mathit{Enc}(w)$ be the resulting (existentially quantified) predicate and $\pi(I_0)$ be the corresponding constraint on the program variables without indices. We let $\exists_i(w) = \pi(I_0)$. It follows that $\exists_i(w)$ is the maximal set of valuations for which $w$ is feasible. Note that existential quantification for the theory of Linear Real Arithmetic is within the theory via Fourier–Motzkin-elimination – hence the computation of $\exists_i(w)$ by an SMT-solver only needs support for Linear Real Arithmetic when $P$ encodes a linear hybrid, stopwatch or timed automaton.[14]

The TAR-based semi-algorithm for the maximal safe initial state problem is presented in Figure 5. The semi-algorithm in Figure 5 works as follows:

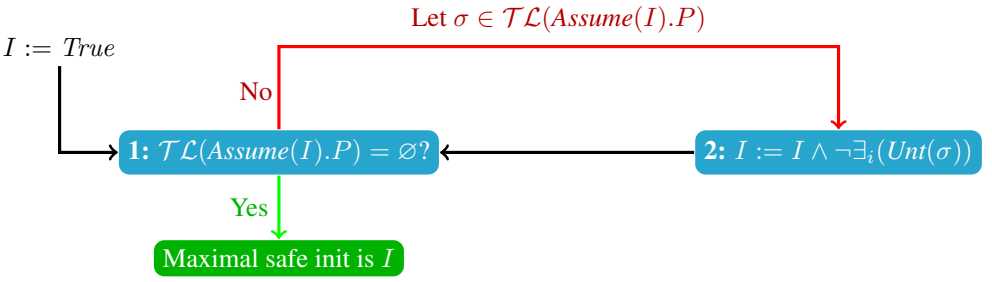



Figure 5: Semi-algorithm $\mathit{SafeInit}$.

1. initially $I = \mathit{True}$
2. using the semi-algorithm 1, check whether $\mathcal{TL}(\mathit{Assume}(I).P)$ is empty
3. if so $P$ does not accept any timed word when we start from $[\![I]\!]$;
4. Otherwise, there is a witness word $\sigma \in \mathcal{TL}(\mathit{Assume}(I).P)$, implying that $I \wedge \mathit{Enc}(\mathit{Unt}(\sigma))$ is satisfiable. It follows that $\exists_i.\mathit{Enc}(\mathit{Unt}(\sigma))$ cannot be part of the maximal set. It is used to strengthen $I$ and repeating from step 2.

If the semi-algorithm terminates, it computes exactly **the** maximal set of values for which the system is safe ($I$), captured formally by Theorem 5.1.

**Theorem 5.1.** If the semi-algorithm $\mathit{SafeInit}$ terminates and outputs $I$, then:

1. $\mathcal{TL}(\mathit{Assume}(I).P) = \varnothing$ and
2. for any $I' \in \beta(V)$, $\mathcal{TL}(\mathit{Assume}(I').P) = \varnothing$ implies $I' \subseteq I$.

[14] This idea of using Fourier-Motzkin elimination has already been proposed [20] in the context of timed Petri nets.

**Proof:**
The fact that $\mathcal{TL}(\mathit{Assume}(I).P) = \varnothing$ follows from termination.

The fact that $I$ is maximal is an invariant of the semi-algorithm: at the beginning, $I = \mathit{True}$ and is clearly maximal. At each iteration, we may subtract a set of valuations $K$ from the previously computed $I$, but these valuations are all such that $\mathcal{TL}(\mathit{Assume}(\nu).P) \neq \varnothing$ for any $\nu \in K$ by definition of existential quantification.

Hence every time a set of valuations is removed by strengthening $I$ only unsafe initial valuations are removed. It follows that if *safeInit* terminates, $I$ is maximal. □

## 5.3. Parameter synthesis

Let $P = (Q, q_0, \Sigma, \Delta, F)$ be a real-time program over a set of variables $X \cup U$ such that: $\forall u \in U, \forall (g, \mu, \rho) \in \Delta, (\nu, \nu') \in \mu \implies \nu(u) = \nu'(u)$ and $\rho(u) = 0$. In words, variables in $U$ are constant variables. Note that they can appear in the guard $g$.

The ***parameter synthesis problem*** asks the following:

**Given a real-time program $P$, find a maximal set $I \in \beta(U)$ s.t. $\mathcal{TL}(\mathit{Assume}(I).P) = \varnothing$.**

The *parameter synthesis problem* is a special case of the maximal safe initial state problem. Indeed, solving the maximal safe initial state problem allows us to find the maximal set of parameters such that $\mathcal{TL}(P) = \varnothing$. Let $I$ be a solution[15] to the maximal safe initial state problem. Then $\exists(\mathit{Vars}(P) \setminus U).I$ is a maximal set of parameter values such that $\mathcal{TL}(P) = \varnothing$.

## 5.4. Robustness checking

Another remarkable feature of our technique is that it can readily be used to check *robustness* of real-time programs and hence timed automata. In essence, checking robustness amounts to enlarging the guards of a real-time program $P$ by an $\varepsilon > 0$. The resulting program is $P_\varepsilon$.

The ***robustness problem*** asks the following:

Given a real-time program $P$, is there some $\epsilon > 0$, s.t. $\mathcal{TL}(P_\epsilon) = \varnothing$.

Using our method we can solve the ***robustness synthesis problem*** which asks the following:

Given a real-time program $P$, find a maximal $\epsilon > 0$, s.t. $\mathcal{TL}(P_\epsilon) = \varnothing$.

This problem asks for a witness (maximal) value for $\epsilon$.

The robustness synthesis is a special case of the parameter synthesis problem where $\epsilon$ is a parameter of the program $P$.

Note that in our experiments (next section), we assume that $P$ is robust and in this case we can compute a maximal value for $\epsilon$. Proving that a program is non-robust requires proving *feasibility* of infinite traces for ever decreasing $\epsilon$. We have developed some techniques (similar to proving termination for standard programs) to do so but this is still under development.

[15] For now assume there is a unique maximal solution.

# 6. Experiments

We have conducted three sets of experiments, each testing the applicability of our proposed method (denoted by RTTAR) compared to state-of-the-art tools with specialized data-structures and algorithms for the given setting. All experiments were conducted on AMD EPYC 7551 Processors and limited to 1 hour of computation. The RTTAR tool uses the UPPAAL parsing-library, but relies on Z3 [31] for the interpolant computation. Our experimental setup is available online [32].

## 6.1. Verification of timed and stopwatch automata

The real-time programs, $P_1$ of Figure 1 and $P_2$ of Figure 3 can be analyzed with our technique. The analysis (RTTAR algorithm 1) terminates in two iterations for the program $P_1$, a stopwatch automaton. As emphasized in the introduction, neither UPPAAL (over-approximation with DBMs) nor PHAVER can provide the correct answer to the reachability problem for $P_1$.

To prove that location 2 is unreachable in program $P_2$ requires to discover an invariant that mixes integers (discrete part of the state) and clocks (continuous part). Our technique successfully discovers the program invariants. As a result the refinement depicted in Figure 4 is constructed and as it contains $\mathcal{L}(A_{P_2})$ the refinement algorithm RTTAR terminates and proves that 2 is not reachable. $A_{P_2}$ can only be analyzed in UPPAAL with significant computational effort and bounded integers.

## 6.2. Parametric stopwatch automata

We compare the RTTAR tool to IMITATOR [25] – the state-of-the-art parameter synthesis tool for reachability[16]. We shall here use the semi-algorithm presented in Section 5 For the test-cases we use the gadget presented initially in Figure 1, a few of the test-cases used in [33], as well as two modified versions of Fischers Protocol, shown in Figure 6. In the first version we replace the constants in the model with parameters. In the second version (marked by robust), we wish to compute an expression, that given an arbitrary upper and lower bound yields the robustness of the system – in the same style as the experiments presented in Section 6.3, but here for arbitrary guard-values.

As illustrated by Table 2 the performance of RTTAR is slower than IMITATOR when IMITATOR is able to compute the results. On the other hand, when using IMITATOR to verify our motivating example from Figure 1, we observe that IMITATOR never terminates, due to the divergence of the polyhedra-computation. This is the effect illustrated in Table 1.

When trying to synthesize the parameters for Fischers algorithm, in all cases, IMITATOR times out and never computes a result. For both two and four processes in Fischers algorithm, our tool detects that the system is safe if and only if $a < 0 \vee b < 0 \vee b - a > 0$. Notice that $a < 0 \vee b < 0$ is a trivial constraint preventing the system from doing anything. The constraint $b - a > 0$ is the only useful one. Our technique provides a formal proof that the algorithm is correct for $b - a > 0$.

In the same manner, our technique can compute the most general constraint ensuring that Fischers algorithm is robust. The result of RTTAR algorithm is that the system is robust iff

[16]We compare with the `EFSynth`-algorithm in the IMITATOR tool as this yielded the lowest computation time in the two terminating instances.

$\epsilon \leq 0 \vee a < 0 \vee b < 0 \vee b - a - 2\epsilon > 0$ – which for $\epsilon = 0$ (modulo the initial non-zero constraint on $\epsilon$) reduces to the constraint-system obtained in the non-robust case.

Table 2: Results for parameter-synthesis comparing RTTAR with IMITATOR. Time is given in seconds. DNF marks that the tool did not complete the computation within an hour.

| | IMITATOR-2.12 | RTTAR |
|---|---|---|
| A1 | DNF | 0.08 |
| Sched2.100.0 | 7.16 | 492.73 |
| Sched2.50.0 | 4.95 | 273.36 |
| fischer_2 | DNF | 0.26 |
| fischer_2_robust | DNF | 0.25 |
| fischer_4 | DNF | 47.96 |
| fischer_4_robust | DNF | 50.26 |

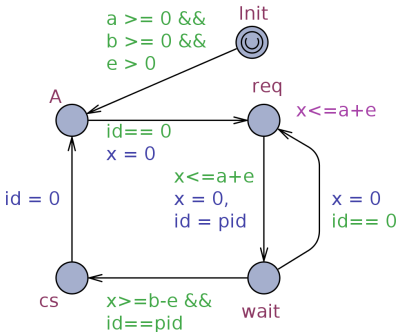



Figure 6: A UPPAAL template for a single process in Fischers Algorithm. The variables `e`, `a` and `b` are parameters for $\epsilon$, lower and upper bounds for clock-values respectively.

## 6.3. Robustness of timed automata

To address the robustness problem for a real-time program $P$, we use the semi-algorithm presented in Section 5 and reduce the robustness-checking problem to that of parameter-synthesis. Notice the delimitation of the input-problems to robust-only instances from Section 5.4.

As Table 3 demonstrates, SYMROB [24] and RTTAR do not always agree on the results. Notably, since the TA `M3` contains strict guards, SYMROB is unable to compute the robustness of it. Furthermore, SYMROB under-approximates $\epsilon$, an artifact of the so-called "loop-acceleration"-technique and the polyhedra-based algorithm. This can be observed in the modified model `M3c`, which is now analyzable by SYMROB, but differs in results compared to RTTAR. This is the same case with the model denoted `a`. We experimented with $\epsilon$-values to confirm that `M3` is safe for all the values tested – while

Table 3: Results for robustness analysis comparing RTTAR with SYMROB. Time is given in seconds. N/A indicates that SYMROB was unable to compute the robustness for the given model.

| | RTTAR_ROBUST | | SYMROB | |
|---|---|---|---|---|
| csma_05 | 32.38 | 1/3 | 0.51 | 1/3 |
| csma_06 | 87.55 | 1/3 | 1.91 | 1/3 |
| csma_07 | 294.30 | 1/3 | 7.37 | 1/3 |
| fischer_04 | 17.64 | 1/2 | 0.19 | 1/2 |
| fischer_05 | 102.50 | 1/2 | 0.77 | 1/2 |
| fischer_06 | 519.41 | 1/2 | 2.83 | 1/2 |
| M3 | 17.14 | $\infty$ | **N/A** | **N/A** |
| M3c | 17.72 | $\infty$ | 3.91 | 250/3 |
| a | 3470.95 | 1/2 | 19.66 | 1/4 |

a is safe only for values tested respecting $\epsilon < \frac{1}{2}$. We can also see that our proposed method is significantly slower than the special-purpose algorithms deployed by SYMROB, but in contrast to SYMROB, it computes the maximal set of good paramaters.

# 7. Conclusion

We have proposed a version of the trace abstraction refinement approach to real-time programs. We have demonstrated that our semi-algorithm can be used to solve the reachability problem for instances which are not solvable by state-of-the-art analysis tools.

Our algorithms can handle the general class of real-time programs that comprises of classical models for real-time systems including timed automata, stopwatch automata, hybrid automata and time(d) Petri nets.

As demonstrated in Section 6, our tool is capable of solving instances of reachability problems problems, robustness, parameter synthesis, that current tools are incapable of handling.

For future work we would like to improve the scalability of the proposed method, utilizing well known techniques such as extrapolations, partial order reduction [34] and compositional verification [35]. Another short-term improvement is to use unsat cores to compute interpolant automata as proposed in [30]. Furthermore, we would like to extend our approach from reachability to more expressive temporal logics.

**Acknowledgments.** The research was partially funded by Innovation Fund Denmark center DiCyPS and ERC Advanced Grant LASSO. Furthermore, these results was made possible by an external stay partially funded by Otto Mønsted Fonden.

# References


[1] Cassez F, Jensen PG, Guldstrand Larsen K. Refinement of Trace Abstraction for Real-Time Programs. In: Hague M, Potapov I (eds.), Reachability Problems. Springer International Publishing, Cham, 2017 pp. 42–58. ISBN: 978-3-319-67089-8.

[2] Clarke EM, Grumberg O, Jha S, Lu Y, Veith H. Counterexample-Guided Abstraction Refinement. In: Emerson EA, Sistla AP (eds.), Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings, volume 1855 of *Lecture Notes in Computer Science*. Springer. ISBN: 3-540-67770-4, 2000 pp. 154–169. doi:10.1007/10722167_15.

[3] Heizmann M, Hoenicke J, Podelski A. Refinement of Trace Abstraction. In: Palsberg J, Su Z (eds.), Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings, volume 5673 of *Lecture Notes in Computer Science*. Springer. ISBN: 978-3-642-03236-3, 2009 pp. 69–85. doi:10.1007/978-3-642-03237-0_7.

[4] Heizmann M, Hoenicke J, Podelski A. Software Model Checking for People Who Love Automata. In: Sharygina N, Veith H (eds.), CAV, volume 8044 of *LNCS*. Springer, 2013 pp. 36–52. ISBN: 978-3-642-39798-1.

[5] Beyer D, Huisman M, Kordon F, Steffen B (eds.). Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III, volume 11429 of *Lecture Notes in Computer Science*. Springer, 2019. ISBN: 978-3-030-17501-6. doi:10.1007/978-3-030-17502-3.

[6] Alur R, Dill DL. A Theory of Timed Automata. *Theor. Comput. Sci.*, 1994. **126**(2):183–235. doi:10.1016/0304-3975(94)90010-8.

[7] Behrmann G, David A, Larsen K, Hakansson J, Petterson P, Yi W, Hendriks M. UPPAAL 4.0. In: QEST'06. 2006 pp. 125–126. doi:10.1109/QEST.2006.59.

[8] Cassez F, Larsen KG. The Impressive Power of Stopwatches. In: Palamidessi C (ed.), CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings, volume 1877 of *Lecture Notes in Computer Science*. Springer, 2000 pp. 138–152. doi:10.1007/3-540-44618-4_12.

[9] Henzinger TA, Kopke PW, Puri A, Varaiya P. What's Decidable about Hybrid Automata? *Journal of Computer and System Sciences*, 1998. **57**(1):94 – 124. doi:http://dx.doi.org/10.1006/jcss.1998.1581.

[10] Frehse G. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In: Morari M, Thiele L (eds.), Hybrid Systems: Computation and Control, volume 3414 of *Lecture Notes in Computer Science*, pp. 258–273. Springer Berlin Heidelberg. ISBN: 978-3-540-25108-8, 2005. doi:10.1007/978-3-540-31954-2_17.

[11] Frehse G, Le Guernic C, Donzé A, Cotton S, Ray R, Lebeltel O, Ripado R, Girard A, Dang T, Maler O. SpaceEx: Scalable Verification of Hybrid Systems. In: Gopalakrishnan G, Qadeer S (eds.), Computer Aided Verification, volume 6806 of *Lecture Notes in Computer Science*, pp. 379–395. Springer Berlin Heidelberg. ISBN: 978-3-642-22109-5, 2011. doi:10.1007/978-3-642-22110-1_30.

[12] Henzinger TA, Ho PH, Wong-toi H. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer*, 1997. **1**:460–463.

[13] Wang W, Jiao L. Trace Abstraction Refinement for Timed Automata. In: Cassez F, Raskin J (eds.), Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings, volume 8837 of *Lecture Notes in Computer Science*. Springer. ISBN: 978-3-319-11935-9, 2014 pp. 396–410. doi:10.1007/978-3-319-11936-6_28.

[14] Alur R, Dang T, Ivančić F. Reachability Analysis of Hybrid Systems via Predicate Abstraction. In: Tomlin CJ, Greenstreet MR (eds.), Hybrid Systems: Computation and Control. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002 pp. 35–48.

[15] Dierks H, Kupferschmid S, Larsen KG. Automatic abstraction refinement for timed automata. In: International Conference on Formal Modeling and Analysis of Timed Systems. Springer, 2007 pp. 114–129.

[16] Frehse G, Jha SK, Krogh BH. A Counterexample-Guided Approach to Parameter Synthesis for Linear Hybrid Automata. In: Egerstedt M, Mishra B (eds.), Hybrid Systems: Computation and Control. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008 pp. 187–200.

[17] Tiwari A. Abstractions for hybrid systems. *Formal Methods in System Design*, 2008. **32**(1):57–83. doi:10.1007/s10703-007-0044-3.

[18] Bradley AR. SAT-Based Model Checking without Unrolling. In: Jhala R, Schmidt D (eds.), Verification, Model Checking, and Abstract Interpretation. Springer Berlin Heidelberg, Berlin, Heidelberg. 2011 pp. 70–87. ISBN: 978-3-642-18275-4.

[19] Bozzano M, Cimatti A, Griggio A, Mattarei C. Efficient Anytime Techniques for Model-Based Safety Analysis. In: Kroening D, Păsăreanu CS (eds.), Computer Aided Verification. Springer International Publishing, Cham. 2015 pp. 603–621. ISBN: 978-3-319-21690-4.

[20] Bérard B, Fribourg L. Reachability Analysis of (Timed) Petri Nets Using Real Arithmetic. In: Baeten JCM, Mauw S (eds.), CONCUR'99 Concurrency Theory. Springer Berlin Heidelberg, Berlin, Heidelberg. 1999 pp. 178–193. ISBN: 978-3-540-48320-5.

[21] Cimatti A, Griggio A, Mover S, Tonetta S. Parameter synthesis with IC3. In: 2013 Formal Methods in Computer-Aided Design. 2013 pp. 165–168. doi:10.1109/FMCAD.2013.6679406.

[22] Kafle B, Gallagher JP, Gange G, Schachte P, Søndergaard H, Stuckey PJ. An iterative approach to precondition inference using constrained Horn clauses. *CoRR*, 2018. **abs/1804.05989**. 1804.05989, URL http://arxiv.org/abs/1804.05989.

[23] Kordy P, Langerak R, Mauw S, Polderman JW. A Symbolic Algorithm for the Analysis of Robust Timed Automata. In: Jones CB, Pihlajasaari P, Sun J (eds.), FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings, volume 8442 of *Lecture Notes in Computer Science*. Springer. ISBN: 978-3-319-06409-3, 2014 pp. 351–366. doi:10.1007/978-3-319-06410-9_25.

[24] Sankur O. Symbolic Quantitative Robustness Analysis of Timed Automata. In: Baier C, Tinelli C (eds.), Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, volume 9035 of *Lecture Notes in Computer Science*. Springer. ISBN: 978-3-662-46680-3, 2015 pp. 484–498. doi:10.1007/978-3-662-46681-0_48.

[25] André É, Fribourg L, Kühne U, Soulat R. IMITATOR 2.5: A Tool for Analyzing Robustness in Scheduling Problems. In: Giannakopoulou D, Méry D (eds.), FM 2012: Formal Methods: 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN: 978-3-642-32759-9, 2012 pp. 33–36. doi:10.1007/978-3-642-32759-9_6.

[26] Bérard B, Cassez F, Haddad S, Lime D, Roux OH. Comparison of the Expressiveness of Timed Automata and Time Petri Nets. In: Pettersson P, Yi W (eds.), Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Uppsala, Sweden, September 26-28, 2005, Proceedings, volume 3829 of *Lecture Notes in Computer Science*. Springer, 2005 pp. 211–225. doi:10.1007/11603009_17.

[27] Cassez F, Roux OH. Structural Translation from Time Petri Nets to Timed Automata. *Journal of Software and Systems*, 2006. **79**(10):1456–1468.

[28] Bérard B, Cassez F, Haddad S, Lime D, Roux OH. The expressive power of time Petri nets. *Theoretical Computer Science*, 2013. **474**:1–20. doi:http://dx.doi.org/10.1016/j.tcs.2012.12.005.

[29] Byg J, Jacobsen M, Jacobsen L, Jørgensen K, Møller M, Srba J. TCTL-Preserving Translations from Timed-Arc Petri Nets to Networks of Timed Automata. *TCS*, 2013. doi:10.1016/j.tcs.2013.07.011.

[30] Dietsch D, Heizmann M, Musa B, Nutz A, Podelski A. Craig vs. Newton in software model checking. In: Bodden E, Schäfer W, van Deursen A, Zisman A (eds.), Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017. ACM. ISBN: 978-1-4503-5105-8, 2017 pp. 487–497. doi:10.1145/3106237.3106307.

[31] De Moura L, Bjørner N. Z3: An Efficient SMT Solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'08/ETAPS'08. Springer-Verlag, Berlin, Heidelberg. ISBN: 3-540-78799-2, 978-3-540-78799-0, 2008 pp. 337–340. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

[32] Jensen PG, Cassez F, Larsen KG. Repeatability for "Verification and Parameter Synthesis for Real-Time Programs using Refinement of Trace Abstraction", 2020. doi:10.5281/zenodo.3952856.

[33] André É, Lipari G, Nguyen HG, Sun Y. Reachability Preservation Based Parameter Synthesis for Timed Automata. In: Havelund K, Holzmann G, Joshi R (eds.), NASA Formal Methods: 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings. Springer International Publishing, Cham. ISBN: 978-3-319-17524-9, 2015 pp. 50–65. doi:10.1007/978-3-319-17524-9_5.

[34] Cassez F, Ziegler F. Verification of Concurrent Programs Using Trace Abstraction Refinement. In: Davis M, Fehnker A, McIver A, Voronkov A (eds.), Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings, volume 9450 of *Lecture Notes in Computer Science*. Springer. ISBN: 978-3-662-48898-0, 2015 pp. 233–248. doi:10.1007/978-3-662-48899-7_17.

[35] Cassez F, Müller C, Burnett K. Summary-Based Inter-Procedural Analysis via Modular Trace Refinement. In: Raman V, Suresh SP (eds.), 34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India, volume 29 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. ISBN: 978-3-939897-77-4, 2014 pp. 545–556. doi:10.4230/LIPIcs.FSTTCS.2014.545.