Refinement of Trace Abstraction for Real-Time Programs

Franck Cassez[1], Peter Gjøl Jensen[1,2] and Kim Guldstrand Larsen[2]

[1] Department of Computing, Macquarie University, Sydney, Australia
[2] Department of Computer Science, Aalborg University, Denmark

Abstract. Real-time programs are made of instructions that can perform assignements 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 the reachability verification problem for real-time programs. We report on the implementation of our algorithm and we show that our new method provides solutions to problems which are unsolvable by the current state-of-the-art tools.

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 is 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.* [12]. In this method the state space is abstracted with predicates on the concrete values of the program variables. The (counter-example guided) *refinement of trace abstraction* (TAR) method was proposed recently by Heizmann *et al.* [17, 18] 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 [11, 6].

Analysis of timed systems. Reasoning about *quantitative* properties of programs requires extended modeling features like real-time clocks. *Timed Automata* [1] (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 [4]. Extending TA, e.g., with the ability to stop and resume

clocks (stopwatches), leads to undecidability of the reachability problem [20, 9]. Semi-algorithms have been designed to verify *hybrid systems* (extended classes of TA) and are implemented in a number of dedicated tools [16, 19, 15]. However, a common difficulty with the analysis of quantitative properties of timed automata and extensions thereof is that ad-hoc 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 analyse 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 *refinement of trace abstractions* (TAR) was proposed by Heizmann *et al.* [17, 18]. It has not been extended to the verification of real-time systems. Wang *et al.* [23] 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. 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. Dierks *et al.* [14] proposed a CEGAR based method for Timed Systems. To the best of our knowledge, this method got limited attention in the community.

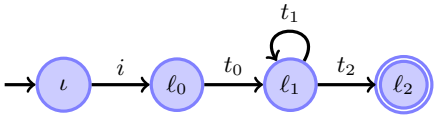
Tools such as UPPAAL [4], SPACEEX [16], HYTECH [19], PHAVER [15], VERIFIX [21], SYMROB [22] and IMITATOR [2] all rely on special-purpose polyhedra libraries to realize their computation.

Our technique is radicaly different to previous approaches and leverages the power of SMT-solvers to discover non-trivial invariants for 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 statespace. 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. This enables us to successfully analyse (terminate) instances of non-decidable classes like stopwatch automata. A simple example is discussed in Section 2.

Our contribution. In this paper, we propose a refinement of trace abstractions (TAR) technique to solev the reachability 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 outperform special-purpose tools and algorithms in their respective domain.

2 Motivating Example

The finite automaton A_1 (Fig. 1), accepts the regular language $\mathcal{L}(A_1) = i.t_0.t_1^*.t_2$. By interpreting the labels of A_1 according to the Table in Fig. 1, we can view it as a stopwatch automaton with 2 clocks, x and z, and one stopwatch y (the variables). Each label defines a *guard* g (a Boolean constraint on the variables), an *update* u which is an (discrete) assignment to the variables, and a *rate* (vector) r that defines the derivatives of the variables.[3] We associate with a sequence $w = a_0.a_1.\cdots.a_n \in \mathcal{L}(A_1)$, a (possibly empty) set of *timed words*, $\tau(w)$, of the form $(a_0, \delta_0).\cdots.(a_n, \delta_n)$ where $\delta_i \geq 0, i \in [0..n]$. 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 $\delta_i \in \mathbb{R}_{\geq 0}, i = 0, 1, 2$ such that following constraints can be satisfied:



Edge	Guard	Update	Rate
i	*True*	`x:=y:=z:=0`	`dy/dt=1`
t_0	*True*	`z:=0`	`dy/dt=0`
t_1	`x==1`	`x:=0`	`dy/dt=0`
t_2	`x-y>=1 and z<1`	-	`dy/dt=0`

Fig. 1: Finite Automaton A_1

$$x_0 = y_0 = z_0 = \delta_0 \wedge \delta_0 \geq 0 \qquad (P_0)$$

$$x_1 = x_0 + \delta_1 \wedge y_1 = y_0 \wedge z_1 = \delta_1 \wedge \delta_1 \geq 0 \qquad (P_1)$$

$$x_1 - y_1 \geq 1 \wedge z_1 < 1 \wedge x_2 = x_1 + \delta_2 \wedge y_2 = y_1 \wedge z_2 = z_1 + \delta_2 \wedge \delta_2 \geq 0 \qquad (P_2)$$

Initially (in location ι, source of edge i) there are no constraints on the variables and the initial values of the variables x, y, z are denoted x_{-1}, y_{-1}, z_{-1} and are unconstrained. Hence we assume that the initial predicate on the variables x_{-1}, y_{-1}, z_{-1} is $P_{-1} = \mathit{True}$. P_0 must be satisfied after taking i and letting time progress for $\delta_0 \geq 0$ time units, which is enforced by a constraint on the variables[4] x_0, y_0, z_0 that stand for the values of x, y, z after taking i; similarly $P_0 \wedge P_1$ must hold after $i.t_0$ and $P_0 \wedge P_1 \wedge P_2$ after $i.t_0.t_2$. Hence the set of timed words associated with $i.t_0.t_2$ is not empty iff $P_0 \wedge P_1 \wedge P_2$ is satisfiable. The *timed language*, $\mathcal{TL}(A_1)$, accepted by A_1 is the set of timed words associated with all the words w accepted by A_1 i.e., $\mathcal{TL}(A_1) = \cup_{w \in \mathcal{L}(A_1)} \tau(w)$.

The *language emptiness problem* is a standard problem in Timed Automata theory and is stated as follows [1]: "given a (Timed) Automaton A, is $\mathcal{TL}(A)$ empty?". It is known that the emptiness problem is decidable for some classes of real-time programs (e.g., Timed Automata [1]), but undecidable for slightly more expressive classes (e.g., Stopwatch Automata [20]). It is ususally possible to compute symbolic representations of sets of *reachable* valuations after a sequence of labels. 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 A_1 does not terminate (Table 1). To force termination, we can compute an over-approximation of the set

[3] As x and z are clocks their rate is always 1 and omitted in the Table.

[4] If x was not reset by i, we would have a constraint $x_0 = x_{-1}$, with x_{-1} unconstrained.

Sequence	PHAver	Uppaal
$i.t_0$	$z = x - y \wedge 0 \leq z \leq x$	$0 \leq y \leq x \wedge 0 \leq z \leq x$
$i.t_0.t_1$	$z = x - y + 1 \wedge 0 \leq x \leq z \leq x + 1$	$0 \leq z - x \leq 1 \wedge 0 \leq y$
$i.t_0.(t_1)^2$	$z = x - y + 2 \wedge 0 \leq x \leq z - 1 \leq x + 1$	$1 \leq z - x \leq 2 \wedge 0 \leq y$
$i.t_0.(t_1)^3$	$z = x - y + 3 \wedge 0 \leq x \leq z - 2 \leq x + 1$	$2 \leq z - x \leq 3 \wedge 0 \leq y$
...	...	...
$i.t_0.(t_1)^k$	$z = x - y + k \wedge 0 \leq x \leq z - k + 1 \leq x + 1$	$k - 1 \leq z - x \leq k \wedge 0 \leq y$
...	...	...

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.

of reachable valuations. Computing an over-approximation is sound (if we declare a timed language to be empty it 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 stopwatch automaton with DBMs. After $i.t_0$, the over-approximation is $0 \leq y \leq x \wedge 0 \leq z \leq x$. This over-approximation intersects the guard $x - y \geq 1 \wedge z - y < 1$ of t_2 and ℓ_2 is reachable but this is an artifact of the over-approximation.[5]

Neither Uppaal nor PHAver can prove that $\mathcal{TL}(A_1) \neq \varnothing$. The technique we introduce in this paper enables us to discover arbitrary abstractions and invariants that enable us to prove $\mathcal{TL}(A_1) \neq \varnothing$. Our method is a version of the *Trace Abstraction Refinement* (TAR) technique introduced in [17]. We illustrate how the method works on the stopwatch automaton A_1:

- find a (untimed) word accepted by A_1. Let $\sigma_1 = i.t_0.t_2$ be such a word. We check whether $\tau(\sigma_1) = \varnothing$ by encoding the corresponding associated timed traces as described by Equations (P_0)–(P_2) and then check whether $P_0 \wedge P_1 \wedge P_2$ is satisfiable[6]. As $P_0 \wedge P_1 \wedge P_2$ is not satisfiable we have $\tau(\sigma_1) = \varnothing$.
- from the proof that $P_0 \wedge P_1 \wedge P_2$ is not satisfiable, we can obtain an *inductive interpolant* that comprises of two predicates I_0, I_1 – one for each conjunction – over the clocks x, y, z. An example of inductive interpolant[7] is $I_0 = x \leq y$ and $I_1 = x - y \leq z$. These predicates are *invariants* of the trace σ_1, and can be used to annotate σ_1 with pre and post-conditions (Equation 1), which are Hoare triples of the form $\{P\}\ a\ \{Q\}$:

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

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

  We can also prove that $\{I_1\}\ (t_1)^*\ \{I_1\}$ is a valid Hoare triple and combined with Equation 1 this gives Equation 2. For each word $w \in i.t_0.(t_1)^*.t_2$, $\tau(w) = \varnothing$ and as $\mathcal{L}(A_1) \subseteq i.t_0.(t_1)^*.t_2$ we can conclude that $\mathcal{TL}(A_1) = \varnothing$.

[5] Uppaal terminates with the result "the language may not be empty".
[6] This can be done using an SMT-solver e.g., Z3.
[7] This is the pair returned by Z3 for $P_0 \wedge P_1 \wedge P_2$.

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 define *real-time programs*. Real-time programs define 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.

Notations. A finite automaton over an alphabet Σ 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, Σ 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 $\sigma = \alpha_0.\alpha_1.\cdots.\alpha_n$ is a finite sequence of letters from Σ; we let $\sigma[i] = \alpha_i$ the i-th letter, $|\sigma|$ be the length of σ which is $n+1$. ϵ is the empty word and $|\epsilon| = 0$, Σ^* is the set of finite words over Σ. 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 ι.

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 $\beta(V)$ a set of *constraints* on the variables in V. Given $\varphi \in \beta(V)$, we let $\mathit{Vars}(\varphi)$ be the set of free variables in φ. The truth value of a constraint φ given a valuation ν is denoted by $\varphi(\nu)$ and we write $\nu \models \varphi$ when $\varphi(\nu) = \mathit{True}$. We let $[\![\varphi]\!] = \{\nu \mid \nu \models \varphi\}$. An *update* of the variables in V is a binary relation $\mu \subseteq [V \to \mathbb{R}] \times [V \to \mathbb{R}]$. Given an update μ 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. A *rate* ρ is a function from V to $\mathbb{Q}$ (rates can be negative), i.e., an element of $\mathbb{Q}^V$. We let $\mathcal{R}(V) \subseteq \mathbb{Q}^V$ be a set of *valid* rates – that is, rates that can be written (syntactically) as a predicate on an edge. Given a valuation ν, a valid rate $\rho \in \mathbb{Q}(V)$ and a timestep $\delta \in \mathbb{R}_{\geq 0}$ the valuation $\nu + \rho \times \delta$ is defined by: $(\nu + \rho \times \delta)(v) = \nu(v) + \rho(v) \times \delta$ for $v \in V$.

Real-Time Instructions. Let $\mathcal{I} = \beta(V) \times \mathcal{U}(V) \times \mathcal{R}(V)$ be a countable set of instructions. Each $\alpha \in \mathcal{I}$ 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 $(\alpha, \delta) \in \mathcal{I} \times \mathbb{R}_{\geq 0}$ we define the following transition relation:

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

The semantics of $\alpha \in \mathcal{I}$ is a mapping $[\![\alpha]\!] : [V \to \mathbb{R}] \to [V \to \mathbb{R}]$ that can be extended to sets of valuations as follows:

$$\begin{aligned} \nu \in [V \to \mathbb{R}],\ [\![\alpha]\!](\nu) &= \{\nu' \mid \exists \delta \geq 0, \nu \xrightarrow{\alpha,\delta} \nu'\} \\ K \subseteq [V \to \mathbb{R}],\ [\![\alpha]\!](K) &= \bigcup_{\nu \in K} [\![\alpha]\!](\nu). \end{aligned}$$

Let K be a set of valuations, $\alpha \in \mathcal{I}$ and $\sigma \in \mathcal{I}^*$. We inductively define the *post operator* $Post$ as follows:

$$Post(K, \epsilon) = K$$
$$Post(K, \alpha.\sigma) = Post([\![\alpha]\!](K), \sigma)$$

The post operator extends to logical constraints $\varphi \in \beta(V)$ by defining $Post(\varphi, \sigma) = Post([\![\varphi]\!], \sigma)$. 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 $Post(\varphi, \sigma)$ can also be expressed as a constraint in $\beta(V)$ for sequences of instructions $\sigma \in \mathcal{I}^*$.

Timed Words and Feasible Words. A *timed word* (over alphabet $\mathcal{I}$) 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 $\alpha_i \in \mathcal{I}$. The timed word σ 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:

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

We let $Unt(\sigma) = \alpha_0.\alpha_1.\cdots.\alpha_n$ be the *untimed* version of σ. We overload the term *feasible* as follows: an untimed word $w \in \mathcal{I}^*$ is feasible iff $w = Unt(\sigma)$ for some feasible timed word σ.

Lemma 1. *An untimed word $w \in \mathcal{I}^*$ is* feasible *iff $Post(True, w) \neq False$.*

Proof. The lemma follows trivially from the inductive definition of *Post*. □

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). A *real-time program* is a pair $P = (A_P, [\![\cdot]\!])$ where A_P is a finite automaton $A_P = (Q, \iota, I, \Delta, F)$ over the finite alphabet[8] $I \subseteq \mathcal{I}$, Δ defines the control-flow graph of the program and $[\![\cdot]\!]$ (as defined previously for $\mathcal{I}$) provides the semantics of each instruction. A timed word σ is *accepted* by P if and only if:

1. $Unt(\sigma)$ is accepted by A_P ($Unt(\sigma) \in \mathcal{L}(A_P)$) and
2. σ is feasible.

Notice that the definition of feasibility of a timed word σ is independent from the acceptance of $Unt(\sigma)$ by A_P. 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 $Unt(\sigma) \in \mathcal{L}(A_P)$ and σ is feasible.

Remark 1. 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 contraining the initial one ν_0. When specifying a real-time program, initial values can be set by regular instructions. This is similar to standard programs where the first instructions can set the values of some variables.

[8] $\mathcal{I}$ can be infinite but we require the transition relation of A_P to be finite.

Timed Language Emptiness Problem. The *(timed) language emptiness problem* asks the following:

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

Theorem 1. *$\mathcal{TL}(P) \neq \varnothing$ 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 σ 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 1, feasibility of w is equivalent to $Post(True, w) \not\subseteq False$. □

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:

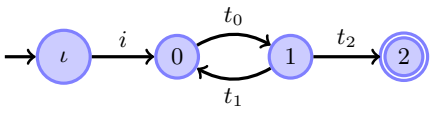
$$b_1, b_2 ::= True \mid False \mid x - y \bowtie k \mid x \bowtie k \mid b_1 \wedge b_2 \tag{3}$$

where $x, y \in V$, $k \in \mathbb{Q}_{\geq 0}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$[9]. 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 [9], 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 Fig. 1.

As there exists syntactic translations (preserving reachability) that maps hybrid automata to stopwatch automata [9], and translations that map time Petri nets [5, 10] and extensions [8, 7] 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 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 automaton in Fig. 2 has two clocks x and y and an unbounded integer variable k. Even though k is unbounded, our technique discovers the invariant $y \geq k$ at location 1 which is over a real-time clock y and the integer variable k. It allows us to prove that $\mathcal{TL}(P_2) = \varnothing$.



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

Fig. 2: Real-time program P_2

[9] 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 propose a semi-algorithm to solve the language emptiness problem for real-time programs. The algorithm is a version of the *refinement of trace abstractions* (TAR) approach [17] for timed systems.

Refinement of Trace Abstration for Real-Time Programs. Fig. 3 gives a precise description of the TAR algorithm for real-time programs. This is the standard trace abstraction refinement algorithm as introduced in [17] – we therefore omit theorems of completeness and soundness as these will be equivalent to the theorems in [17] and are proved in the exact same manner. The input to the algorithm is a real-time program $P = (A_P, [\![\cdot]\!])$. An invariant of the algorithm is that R is empty or contains only infeasible traces.

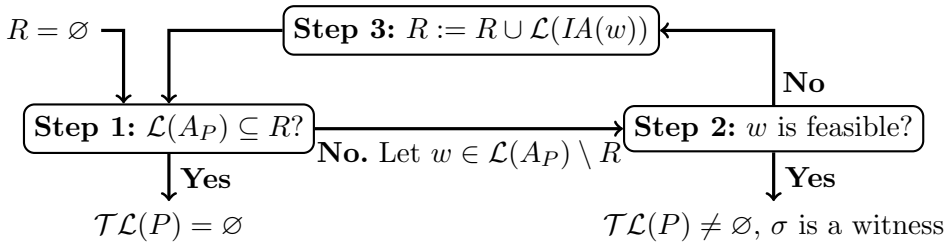


Fig. 3: Trace Abstraction Refinement Algorithm for Real-Time Programs

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

Step 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 algorithm terminates. Otherwise, there is a sequence $w \in \mathcal{L}(A_P) \setminus R$, goto Step 2;

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

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

Contruction of Interpolant Automata. When it is determined that a trace w is infeasible, we can easily discard such a single trace and continue searching. 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 the reason of infeasiblity of w and is formally specified by an *interpolant automaton*, $IA(w)$. The reason for infeasibility itself has the form of an *inductive interpolant*.

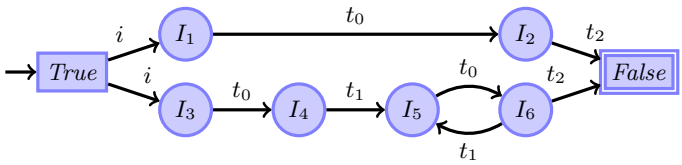


Fig. 4: Interpolant automaton for $\mathcal{L}(IA(w_1)) \cup \mathcal{L}(IA(w2))$.

Given a word $w = a_0.a_1.\cdots.a_m \in \mathcal{I}^*$, we can check whether w is feasible by encoding the side-effects of each instruction in w, similar to a Static Single Assignment (SSA) form in programming languages. We denote $Enc(w) = P_0 \wedge P_1 \wedge \cdots \wedge P_m$ the result of the encoding of the side effects of w.

An example of an encoding for the real-time program A_1 (Fig. 1) is given by the predicates in Equation (P_0)–(P_2). The variables x_k, y_k, z_k denote the values of x, y, z after k steps (initially the variables can have arbitrary values). The sequence $w_1 = i.t_0.t_2$ is feasible iff $Enc(w_1) = P_0 \wedge P_1 \wedge P_2$ is satisfiable. Given a conjunctive formula $f = P_0 \wedge \cdots \wedge P_m$, if f is unsatisfiable, an *interpolating* SMT-solver is capable of producing inductive arguments for the unsatisfiability reason. This argument is an *inductive interpolant* $I_0, \ldots, I_{m-1}$ s.t for any postfix-sequence $0 \leq n \leq m$ it holds that $I_n \wedge P_{n+1} \wedge \ldots P_m$ is unsatisfiable (and the variables appearing in I_n must be variables used in $P_0 \cdots P_n$ and $P_{n+1}, \cdots, P_m$).

One can intuitively think of each interpolant as a *sufficient* conditions for infeasibility of the post-fix of the trace and this can be represented by a sequence of Hoare triples of the form $\{P\}\ a\ \{Q\}$:

$$\{True\} \quad a_0 \quad \{I_0\} \quad a_1 \quad \{I_1\} \quad \cdots \quad \{I_{m-1}\} \quad a_m \quad \{False\}$$

Consider the real-time program P_2 of Fig. 2 and the two infeasible untimed words $w_1 = i.t_0.t_2$ and $w_2 = i.t_0.t_1.t_0.t_2$. The Hoare triples for w_1 and w_2 are given by Equation 4-5 where the predicates are: $I_1 = y \geq x \wedge (k = 0)$, $I_2 = y \geq k$, $I_3 = y \geq x \wedge k \leq 0$, $I_4 = y \geq 1 \wedge k \leq 0$, $I_5 = y \geq k + x$, $I_6 = y \geq k + 1$.

$$\{True\} \quad i \quad \{I_1\} \quad t_0 \quad \{I_2\} \quad t_2 \quad \{False\} \tag{4}$$

$$\{True\} \quad i \quad \{I_3\} \quad t_0 \quad \{I_4\} \quad t_1 \quad \{I_5\} \quad t_0 \quad \{I_6\} \quad t_2 \quad \{False\} \tag{5}$$

As can be seen in Equation 5, 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 $Post(I_6, t_1) \subseteq I_5$, 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 $IA(w_2)$ as 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, we obtain a refinement which is $\mathcal{L}(IA(w_1)) \cup \mathcal{L}(IA(w_2))$, Fig. 4. The detailed construction of $IA(w)$ for an infeasible word w is given in Appendix A.

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 as a linear program. 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).

In the sequel, we assume that our encoding satisfies Lemma 2, Appendix A.

5 Experiments

We have conducted two 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 Opteron 6376 Processors and limited to 1 hour of computation. The RTTAR tool uses the UPPAAL parsing-library, but relies on Z3 [13] for the interpolant computation.

Verification of Timed and Stopwatch Automata. The real-time programs, P_1 of Fig. 1 and P_2 of Fig. 2 can be analysed with our technique. The analysis (RTTAR algorithm, 3) terminates in two iterations for the program P_1, a stopwatch automaton. As emphasised in the introduction, neither UPPAAL (over-approximation with DBMs) nor PHAVER can provide the correct answer to 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 I_5 and I_6 (thanks to the interpolating SMT-solver). As a result the refinement depicted in Fig. 2 is constructed and as it contains $\mathcal{L}(A_{P_2})$ the refinement algorithm terminates and proves that 2 is not reachable. A_{P_2} can only be analysed in UPPAAL with significant computational effort and bounded integers.

Robustness of Timed Automata. Another remarkable feature of our technique is that it can readily be used to check *robustness* of timed automata. In essence, checking robustness amounts to enlarging the guards of an TA A by an $\varepsilon > 0$. The resulting TA is A_ε. The automaton A is (safety) robust iff there is some $\varepsilon > 0$ such $\mathcal{TL}(A_\epsilon) = \varnothing$.

To address the robustness problem for a real-time program P, we use our RTTAR algorithm as follows:

1. build P_ϵ with guards enlarged by ϵ; ϵ is a parameter (a stopwatch with rate 0) that is unconstrained at the beginning of the program. The first instruction in the program P_ϵ is a constraint enforcing $\epsilon > 0$ and then P_ϵ is similar to P (with guards enlarged);
2. use RTTAR to check whether $\mathcal{TL}(P_\epsilon) = \varnothing$;

3. if $\mathcal{TL}(A_\epsilon) = \varnothing$ then P is robust for any $\epsilon > 0$. Otherwise, we can synthesise a constraint[10] of the form $\epsilon \bowtie k$, $\bowtie \in \{>, \geq\}$, such $\mathcal{TL}(A_\epsilon) \neq \varnothing$ for $\epsilon \bowtie k$. Hence we know that $\mathcal{TL}(A_\epsilon) = \varnothing$ can only be achieved if $\epsilon \leq k$ if the constraint is $\epsilon > k$ (or $\epsilon < k$ if $\epsilon \geq k$).
4. We now build $P_{\epsilon<k}$ similar to P_ϵ but the first instruction enforces $0 < \epsilon < k$. We iterate and use RTTAR to check whether $\mathcal{TL}(P_{\epsilon<k}) = \varnothing$.

The details and proofs of this algorithm are special cases of the algorithm provided in Appendix B. Assuming P is robust[11] i.e., there exists some $\epsilon > 0$ such that $\mathcal{TL}(A_\epsilon) = \varnothing$ and the previous process terminates we can compute the largest set of parameters for which P is robust.

As Table 2 demonstrates, SYMROB [22] 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 over-approximates ϵ, 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 differ in results compared to RTTAR. This is the same case with the model denoted `a`. We experimented with ϵ-values to confirm that `M3` is safe for all the values tested – while `a` is safe only for values tested respecting $\epsilon < \frac{1}{2}$. We can also see that our proposed method is significantly slower than SYMROB. As our tool is currently only a prototype with rudimentary state-space-reduction-techniques, this is to be expected.

Test	Time	$\epsilon <$	Time	$\epsilon <$
	SYMROB		RTTAR	
`csma_05`	0.43	1/3	68.23	1/3
`csma_06`	2.44	1/3	227.15	1/3
`csma_07`	8.15	1/3	1031.72	1/3
`fischer_04`	0.16	1/2	45.24	1/2
`fischer_05`	0.65	1/2	249.45	1/2
`fischer_06`	3.71	1/2	1550.89	1/2
`M3c`	4.34	250/3	43.10	∞
`M3`	**N/A**	**N/A**	43.07	∞
`a`	27.90	1/4	15661.14	1/2

Table 2: 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.

Parametric Stopwatch Automata. In our last series of tests, we compare the RTTAR tool to IMITATOR [2] – the state-of-the-art parameter synthesis tool for reachability [12]. We here extend our algorithm in a similar manner as to what was proposed in Section 5, but for arbitrary parameters – the exact algorithm is available in Appendix B. For the test-cases we use the gadget presented initially in Fig. 1, a few of the test-cases used in [3], as well as two modified version of

[10] The form of the constraint is due upward closure properties of the robustness problem.

[11] Proving that a system is non-robust requires proving *feasibility* of infinite traces for ever decreasing ϵ. We have developed some techniques to do so but this is outside of the scope of this paper.

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

Test	IMITATOR	RTTAR
`Sched2.50.0`	201.95	1656.00
`Sched2.100.0`	225.07	656.26
A_1	DNF	0.1
`fischer_2`	DNF	0.23
`fischer_4`	DNF	40.13
`fischer_2_robust`	DNF	0.38
`fischer_4_robust`	DNF	118.11

Table 3: 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.

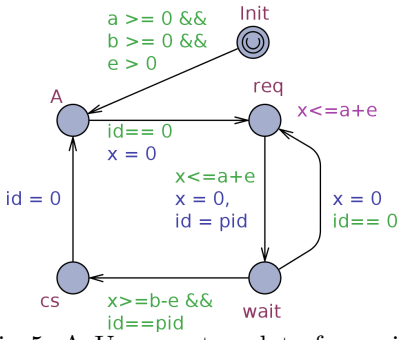


Fig. 5: A UPPAAL template for a single process in Fischers Algorithm. The variables `e`, `a` and `b` are parameters for ϵ, lower and upper bounds for clock-values respectively.

Fischers Protocol, shown in Fig. 5. 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 5, but here for arbitrary guard-values.

As illustrated by Table 3 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 Fig. 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 Fishers algorithm, in all cases, IMITATOR times out and never computes a result. For both two and four processes in Fishers 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 trival 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 Fishers algorithm is robust. The result of RTTAR algorithm is that the system is robust iff $\epsilon \leq 0 \vee a < 0 \vee b < 0 \vee b - a - 2\epsilon > 0$ – which for $\epsilon = 0$ (modulo the initial non-zero constraint on ϵ) reduces to the constraint-system obtained in the non-robust case.

6 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 5, 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 and compositional verification. Furthermore, we would like to extend our approach from reachability to more expressive temporal logics.

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.
2. Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. *IMITATOR 2.5: A Tool for Analyzing Robustness in Scheduling Problems*, pages 33–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
3. Étienne André, Giuseppe Lipari, Hoang Gia Nguyen, and Youcheng Sun. *Reachability Preservation Based Parameter Synthesis for Timed Automata*, pages 50–65. Springer International Publishing, Cham, 2015.
4. G. Behrmann, A. David, K.G. Larsen, J. Hakansson, P. Petterson, Wang Yi, and M. Hendriks. Uppaal 4.0. In *QEST'06*, pages 125–126, 2006.
5. Béatrice Bérard, Franck Cassez, Serge Haddad, Didier Lime, and Olivier H. Roux. Comparison of the expressiveness of timed automata and time petri nets. In Paul Pettersson and Wang Yi, editors, *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*, pages 211–225. Springer, 2005.
6. Dirk Beyer. Competition on software verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 504–524. Springer, 2012.
7. J. Byg, M. Jacobsen, L. Jacobsen, K.Y. Jørgensen, M.H. Møller, and J. Srba. TCTL-preserving translations from timed-arc Petri nets to networks of timed automata. *TCS*, 2013. Available at http://dx.doi.org/10.1016/j.tcs.2013.07.011.
8. Béatrice Bérard, Franck Cassez, Serge Haddad, Didier Lime, and Olivier H. Roux. The expressive power of time Petri nets. *Theoretical Computer Science*, 474:1–20, 2013.
9. Franck Cassez and Kim Guldstrand Larsen. The impressive power of stopwatches. In Catuscia Palamidessi, editor, *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, volume 1877 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2000.
10. Franck Cassez and Olivier Henri Roux. Structural translation from time petri nets to timed automata. *Journal of Software and Systems*, 79(10):1456–1468, October 2006.
11. Franck Cassez, Anthony M. Sloane, Matthew Roberts, Matthew Pigram, Pongsak Suvanpong, and Pablo González de Aledo Marugán. Skink: Static analysis of programs in LLVM intermediate representation (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017. Proceedings*, 2017. forthcoming.

12. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. *Counterexample-Guided Abstraction Refinement*, pages 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
13. Leonardo De Moura and Nikolaj Bjørner. 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, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
14. Henning Dierks, Sebastian Kupferschmid, and Kim G Larsen. Automatic abstraction refinement for timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 114–129. Springer, 2007.
15. Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer Berlin Heidelberg, 2005.
16. Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer Berlin Heidelberg, 2011.
17. Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. *Refinement of Trace Abstraction*, pages 69–85. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
18. Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 36–52. Springer, 2013.
19. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.
20. Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94 – 124, 1998.
21. Piotr Kordy, Rom Langerak, Sjouke Mauw, and Jan Willem Polderman. *A Symbolic Algorithm for the Analysis of Robust Timed Automata*, pages 351–366. Springer International Publishing, Cham, 2014.
22. Ocan Sankur. *Symbolic Quantitative Robustness Analysis of Timed Automata*, pages 484–498. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.
23. Weifeng Wang and Li Jiao. *Trace Abstraction Refinement for Timed Automata*, pages 396–410. Springer International Publishing, Cham, 2014.

A Construction of Interpolant Automata

In this section we detail how to construct interpolant automata given an infeasible trace. Let us initially detail the encoding of a single untimed word as a constraint-system.

Checking Feasibility. Given a word $\sigma \in \mathcal{I}^*$, we can check whether σ is feasible by encoding the side-effects of each instruction in σ, similar to a Static Single Assignment (SSA) form in programming languages.

Let us define a function for constructing such a constraint-system characterizing the feasibility of a given trace. We shall assume that constraints in $\beta(V)$ and updates in $\mathcal{U}(V)$ are syntactically defined. Let $P = (Q, q_0, \mathcal{I}, \Delta, F)$ be a real-time program and $\sigma \in \mathcal{I}^*$ be a word over $\mathcal{I}$. Let $V^n = \{x^n, x^n_\mu \mid x \in V\} \cup \{\delta^n\}$ be a set of variables extended with an index $n \in \mathbb{N}_{\geq 0}$. For a given constraint-system $\varphi \in \beta(V)$ write $\varphi_{[V/V^n]}$ for replacing all occurrences of V with their indexed occurrence in V^n (implying that $\varphi_{[V/V^n]} \in \beta(V^n)$). We assume that the relation $\mu \in \mathcal{U}(V)$ is of SSA form, and let $\mu_{[V/(V^n,V^m)]}$ be the replacement of all occurrences of variables $x \in V$ with their indexes and sub-scripted occurrence in V^n if x is assigned to and from V^m if x is read from, such that $(v \leftarrow v + w)_{[V/(V^n,V^m)]} = v^n_\mu \leftarrow v^m + w^m$. Given this we can now recursively define the function $Enc : \mathcal{I}^* \to \beta(\{V^n \mid 0 \leq n \leq |\sigma|\})$

$$
\begin{aligned}
Enc(\epsilon) =& \mathit{True} \\
Enc(\sigma.\alpha) =& Enc(\sigma) \wedge \delta^n \geq 0 \wedge \varphi_{[V/V^{n-1}]} \wedge \delta^n \geq 0 \wedge \mu_{[V/(V^n_\mu, V^{n-1})]} \\
& \wedge \bigwedge_{v \in V} v^n = v^n_\mu + \rho(v) \times \delta^n \\
& \text{where } n = |\sigma| - 1 \text{ and } (\varphi, \mu, \rho) = \alpha
\end{aligned}
$$

The function $Enc : \mathcal{I}^* \to \beta(V^{\mathbb{N}_{\geq 0}})$ constructs a constraint-system characterizing exactly the feasibility of a word σ:

Lemma 2. *A word σ is feasible i.e., $Post(True, \sigma) \not\subseteq False$ iff $Enc(\sigma)$ is satisfiable.*

We shall frequently refer to such a constraints system $C = Enc(\sigma)$ for some word σ where $|\sigma| = n$ as a sequence of conjunctions $P_0 \wedge \cdots \wedge P_m \wedge \cdots \wedge P_n = C$ where $P_m \in \beta(V^{m-1} \cup V^m)$ refers to the encoding of the m'th instruction, and we shall call such an element P_m a predicate.

An example of an encoding for the real-time program A_1 (Fig. 1) is given by the predicates P_0, P_1, P_2 (Equation (P_0)–(P_2)). The variables x_k, y_k, z_k denote the values of x, y, z after k steps (initially the variables can have arbitrary values). The sequence $i.t_0.t_2$ is feasible iff $P_0 \wedge P_1 \wedge P_2$ is satisfiable.

From such a sequence we can use interpolating SMT-solvers to construct a sequence of *craig-interpolants* – this procedure is discussed in more detail in Section 4.

The IA Algorithm. From a sequence of interpolants, we can generalize the trace in the following manner. Let $I_0, \dots I_k$ be interpolants for the constraint-system $P_0 \wedge \cdots \wedge P_{k+1} = \mathit{Enc}(\sigma)$ for some word σ where $k = |\sigma| - 1$ and let $\mathcal{A} = (Q, q_0, \Sigma, \Delta, F)$ be the automata description of our Real Time Program. Then we can construct an interpolant automaton $\mathcal{A}^I = (Q^I, q_0^I, \Sigma^I, \Delta^I, F^I)$ s.t. $\sigma \in \mathcal{L}(\mathcal{A}^I)$ and for all $\sigma' \in \mathcal{L}(\mathcal{A}^I)$ we have that σ' is infeasible.

Let $Q = \{\mathit{True}, \mathit{False}, I_0, \dots, I_k\}$, $q_0 = \mathit{True}$, $\Sigma^I = \Sigma$, $F = \{\mathit{False}\}$, then we let the transition-function be defined as follows.

1. $(\mathit{True}, \sigma[0], I_0) \in \Delta^I$,
2. $(I_k, \sigma[k], \mathit{False}) \in \Delta^I$,
3. $(I_{n-1}, \sigma[n-1], l_n) \in \Delta^I$ for $1 < n \leq k$, and
4. for each $1 \leq n, m \leq k$, if $I_m \subset_V I_n$ then $(I_{n-1}, \sigma[n-1], I_m) \in \Delta^I$ where $\subset_V$ is subset-checking, modulo variable indexing.

The above construction gives us the algorithm IA for constructing interpolant automata from a trace σ.

Lemma 3 (Interpolants). *Let σ be an infeasible word over P, then for all $\sigma' \in \mathcal{L}(IA(\sigma))$, σ' is infeasible.*

We can verify that the construction using rules 1-3 is correct as these come directly from the feasibility-check of the trace and the definition of interpolants.

The *pumping*-rule (rule 4) utilizes that if by firing some transition labeled α from some interpolant I_{n-1} gives us a "stronger" argument for infeasibility than in I_m, then surely every sequence which is infeasible from I_m is also infeasible from I_{n-1} after firing α.

B Parameter Synthesis

In this section we show how to use the trace abstraction refinement algorithm presented in Section 4 to synthesise *good initial values* for some of the program variables. 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, P does not accept any timed word.

Given a constraint $I \in \beta(V)$, we define the associated *assume* guard-transformer for instructions that for a letter $\alpha = (\gamma, \rho, \mu)$ defines $\mathit{Assume}(\alpha, I) = (\gamma', \rho, \mu)$ s.t. $\gamma = \gamma \wedge I$. Let $P = (Q, \iota, \mathcal{I}, \Delta, F)$ be a real-time program. Then we can define the real-time program $\mathit{Assume}(I).P = (Q, \iota, \mathcal{I}, (\Delta \backslash \{(\iota, i, q_0)\}) \cup \{(\iota, \mathit{Assume}(i, I), q_0)\}, F)$.

Safe Initial Set Problem. The *safe initial state problem* asks the following:

> Given a real-time program P, is there $I \in \beta(V)$ s.t. $\mathcal{TL}(\mathit{Assume}(I).P) = \varnothing$?

Semi-Algorithm for the Safe Initial State Problem. Let $w \in \mathcal{L}(P)$. When $Enc(w)$ is satisfiable, we define the (existentially quantified) constraint $\exists Vars(Enc(w)) \setminus V_{-1}.Enc(w)$ i.e., the projection of the set of solutions on the initial values of the variables. We let $\exists_i(w)$ be $\exists Vars(Enc(w)) \setminus V_{-1}.Enc(w)$ with all the free occurrences of x_{-1} replaced by x (remove index for each var). $\exists_i(w)$ is a contraint over the set of variables V (and existential quantifiers)[13].

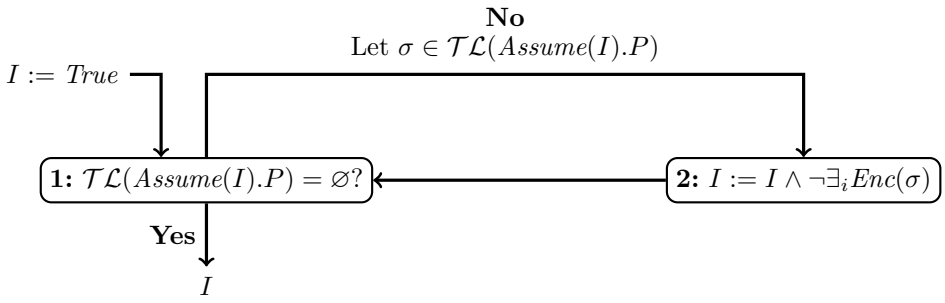


Fig. 6: Semi-algorithm *SafeInit*.

The Algorithm in Fig. 6 works as follows:

1. initially $I = True$;
2. using the algorithm from Figure 3, test if $\mathcal{TL}(Assume(I).P)$ is empty – if so P does not accept any timed word when we start from $[\![I]\!]$.
3. Otherwise, there is a witness word $\sigma \in Unt(\mathcal{TL}(Assume(I).P))$, implying that $I \wedge Enc(\sigma)$ is satisfiable. We can then determine a sufficient condition $\exists_i(\sigma)$ to be feasible when starting the program from $\exists_i(Enc(\sigma))$. We therefore strengthen the constraint I (step 2).

Algorithm *SafeInit* when it terminates, computes the maximal constraint I for which $\mathcal{TL}(Assume(I).P) = \varnothing$. This is a consequence of Theorem 2:

Theorem 2. *If algorithm SafeInit terminates and outputs I, then for any $I' \in \beta(V)$, $\mathcal{TL}(Assume(I').P) = \varnothing$ if and only if $I' \subseteq I$.*

Proof ($\Longrightarrow$). Let us assume by contraposition that upon termination we have $\mathcal{TL}(Assume(I).P) \neq \varnothing$. This contradicts either the termination criterion of the algorithm from Figure 3 or the termination criterion of the algorithm presented in Figure 6. □

Proof ($\Longleftarrow$). Let us assume by contraposition that upon termination there exists some $I' \neq \varnothing$ for which $I' \cap I = \varnothing$ and $\mathcal{TL}(Assume(I').P) = \varnothing$. Then let us prove inductively that no such I' can ever exist.

[13] Existential quantification for the theory of Liniear Real Arithmetic is within the theory via Fourier–Motzkin-elimination – hence the solver only needs support for Linear Real Arithmetic for Parameter Synthesis for Stopwatch and Timed Automata.

In the base-case in step 1, if the algorithm terminates, clearly $I' = \varnothing$ contradicting our requirements for the contraposition.

For our contraposition to be valid, we must instead look at how we modify I in step 2. For I' to exist, the quantification over parameter-values for σ must construct a larger-than-needed set of parameter value, i.e., that $I' \in \neg\exists_i Enc(\sigma)$. This contradicts the definition of existential quantification. As we never over-approximate the parameter-set needed for the valuation in step 2, we can conclude that I' cannot exist. □