Automated Verification for Real-Time Systems
via Implicit Clocks and an Extended Antimirov Algorithm

Yahui Song and Wei-Ngan Chin

School of Computing, National University of Singapore, Singapore
{yahuis,chinwn}@comp.nus.edu.sg

Abstract. The correctness of real-time systems depends both on the correct functionalities and the realtime constraints. To go beyond the existing Timed Automata based techniques, we propose a novel solution that integrates a modular Hoare-style forward verifier with a term rewriting system (TRS) on *Timed Effects* (*TimEffs*). The main purposes are to: increase the expressiveness, dynamically manipulate clocks, and efficiently solve clock constraints. We formally define a core language C^t, generalizing the real-time systems, modeled using mutable variables and timed behavioral patterns, such as *delay*, *timeout*, *interrupt*, *deadline*. Secondly, to capture real-time specifications, we introduce *TimEffs*, a new effects logic, that extends *regular expressions* with dependent values and arithmetic constraints. Thirdly, the forward verifier reasons temporal behaviors – expressed in *TimEffs* – of target C^t programs. Lastly, we present a purely algebraic TRS, i.e., an extended *Antimirov algorithm*, to efficiently check language inclusions between *TimEffs*. To demonstrate the feasibility of our proposal, we prototype the verification system; prove its soundness; report on case studies and experimental results.

1 Introduction

During the last three decades, a popular approach for specifying real-time systems has been based on Timed Automata (TAs) [1]. TAs are powerful in designing real-time models via explicit clocks, where real-time constraints are captured by explicitly setting/resetting clock variables. A number of automatic verification tools for TAs have proven to be successful [2–5]. Industrial case studies show that requirements for real-time systems are often structured into phases, which are then composed sequentially, in parallel, alternatively [6, 7]. TAs lack high-level compositional patterns for hierarchical design; moreover, users often need to manipulate clock variables with carefully calculated clock constraints manually. The process is tedious and error-prone.

There have been some translation-based approaches on building verification support for compositional timed-process representations. For example, Timed Communicating Sequential Process (TCSP), Timed Communicating Object-Z (TCOZ) and *Statechart* based hierarchical Timed Automata are well suited for presenting compositional models of complex real-time systems. Prior works [8, 9] systematically translate TCSP/TCOZ/Statechart models to flat TAs so that the

model checker Uppaal [3] can be applied. However, possible insufficiencies are: the expressiveness power is limited by the finite-state automata; and there is always a gap between the verified logic and the actual code implementation.

In this work, we investigate an alternative approach for verifying real-time systems. We propose a novel temporal specification language, Timed Effects (*TimEffs*), which enables a compositional verification via a Hoare-style forward verifier and a term rewriting system (TRS). More specifically, we specify system behaviors in the form of *TimEffs*, which integrates the Kleene Algebra with dependent values and arithmetic constraints, to provide real-time abstractions into traditional linear temporal logics. For example, one safety property, *"The event `Done` will be triggered no later than one time unit"*[1], is expressed in *TimEffs* as: $\Phi \triangleq$ `0≤t<1` $\wedge$ `(_*·Done)#t`. Here $\wedge$ connects the arithmetic formula and the timed trace; the operator `#` binds time variables to traces (here `t` is a time bound of `(_*·Done)`); `_` is a wildcard matching to any event; Kleene star $\star$ denotes a trace repetition. The above formula Φ corresponds to '$\Diamond_{[0,1)}$`Done`' in metric temporal logic (MTL), reads *"within one time unit, `Done` finally happens"*. Furthermore, the time bounds can be dependent on the program inputs, as shown in Fig. 1.

```
1  void addOneSugar()
2  /* req: true ∧ _*
3    ens: t>1 ∧ ϵ # t */
4  { timeout (() , 1); }
5
6  void addNSugar (int n)
7  /* req: true ∧ _*
8    ens: t≥n ∧ EndSugar # t */
9  { if (n == 0) {
10     event ["EndSugar"];}
11   else {
12     addOneSugar();
13     addNSugar (n-1);}}
```

Fig. 1. Value-dependent specification.

Function `addNSugar` takes a parameter `n`, representing the portion of the sugar to add. When `n=0`, it raises an event `EndSugar` to mark the end of the process. Otherwise, it adds one portion of the sugar by calling `addOneSugar()`, then recursively calls `addNSugar` with parameter `n-1`. The use of `timeout(e, d)` is standard [11], which executes a block of code `e` after the specified time `d`. Therefore, the time spent on adding one portion of the sugar is more than one time unit. Note that `ϵ#t` refers to an empty trace which takes time `t`. Both preconditions require no arithmetic constraints and no temporal constraints upon the history traces. The postcondition of `addNSugar(n)` indicates that the method generates a finite trace where `EndSugar` takes a no less than `n` time-units delay to finish.

Although these examples are simple, they show the benefits of deploying value-dependent time bounds, which is beyond the capability of TAs. Essentially, *TimEffs* define symbolic TAs, which stands for a set (possibly infinite) of concrete transition systems. Moreover, we deploy a Hoare-style forward verifier to soundly reason about the behaviors from the source level, with respect to the well-defined operational semantics. This approach provides a *direct* (opposite to the techniques which require manual and remote modeling processes), and modular verification – where modules can be replaced by their already verified properties – for real-time systems, which are not possible by any existing tech-

[1] In this paper, we pretend time is discrete and only integral values. However, it's just as easy to represent continuous time by letting time variables assume real values [10].

niques. Furthermore, we develop a novel TRS, which is inspired by Antimirov and Mosses' algorithm[2] [12] but solving the language inclusions between more expressive *TimEffs*. In short, the main contributions of this work are:

1. **Language Abstraction:** we formally define a core language C^t, by defining its syntax and operational semantics, generalizing the real-time systems with mutable variables and timed behavioral patterns, e.g., *delay*, *timeout*, *deadline*.
2. **Novel Specification:** we propose *TimEffs*, by defining its syntax and semantics, gaining the expressive power beyond traditional linear temporal logics.
3. **Forward Verifier:** we establish a sound effect system to reason about temporal behaviors of given programs. The verifier triggers the back-end solver TRS.
4. **Efficient TRS:** we present the rewriting rules to (dis)prove the inclusion relations between the actual behaviors and the given specifications, both in *TimEffs*.
5. **Implementation and Evaluation:** we prototype the automated verification system, prove its soundness, report on case studies and experimental results.

2 Overview

An overview of our automated verification system is given in Fig. 2. The system consists of a forward verifier and a TRS, i.e., the rounded boxes. The input of the forward verifier is a C^t program annotated with temporal specifications written in *TimEffs*. The input of the TRS is a pair of effects LHS and RHS, referring to the inclusion LHS $\sqsubseteq$ RHS[3] to be checked *(LHS and RHS refer to left/right-hand-side effects respectively)*. The forward verifier calls TRS to solve proof obligations. Next, we use Fig. 3 to highlight our main methodologies, which simulates a coffee machine, that dynamically adds sugar based on the user's input number.

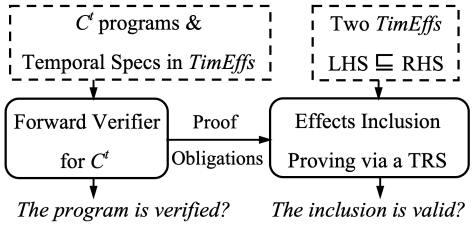

Fig. 2. System Overview.

2.1 *TimEffs*. We define Hoare-triple style specifications (enclosed in `/*...*/`) for each function, which leads to a compositional verification strategy, where static checking can be done locally. The precondition of `makeCoffee` specifies that the input value `n` is non-negative, and it *requires* that before entering into this function, this history trace must contain the event `CupReady` on the tail. The verification fails if the precondition is not satisfied at the caller sites. Line 17 sets a five time-units deadline (i.e., maximum 5 portion of sugar per coffee) while calling `addNSugar` (defined in Fig. 1); then emits event `Coffee` with a deadline, indicating the pouring coffer process takes no more than four time-units. The precondition of `main` requires no arithmetic constraints (expressed as `true`) and an empty history trace. The postcondition of `main` specifies that before the final

[2] Antimirov and Mosses' algorithm was designed for deciding the inequalities of regular expressions based on an axiomatic algorithm of the algebra of regular sets.

[3] The *TimEffs* inclusion relation $\sqsubseteq$ is formally defined in Definition 3.

Done happens, there is no occurrence of Done (! indicates the absence of events); and the whole process takes no more than nine time-units to hit the final event.

```
14 void makeCoffee (int n)
15 /* req: n≥0 ∧ _*· CupReady
16   ens: n≤t≤5 ∧ t'≤4 ∧
         (EndSugar # t) · (Coffee # t') */
17 { deadline (addNSugar(n), 5);
18   deadline (event["Coffee"],4);}
19
20 int main ()
21 /* req: true ∧ ϵ
22   ens: t≤9 ∧ ((!Done)* # t) · Done */
23 { event["CupReady"];
24   makeCoffee (3);
25   event["Done"];}
```

Fig. 3. To make coffee with three portions of sugar within nine time units.

TimEffs support more features such as *disjunctions*, *guards*, *parallelism* and *assertions*, etc (cf. Sec. 3.3), providing detailed information upon: branching properties: different arithmetic conditions on the inputs lead to different effects; and required history traces: by defining the prior effects in precondition. These capabilities are beyond traditional timed verification, and cannot be fully captured by any prior works [2–5, 8, 9]. Nevertheless, the increase in expressive power needs support from finer-grind reasoning and a more sophisticated back-end solver, discharged by our forward verifier and TRS.

1. `void addOneSugar(){` *// initialize the state using the function precondition.*
   $\Phi_C{=}\Phi_{pre}^{addOneSugar(n)} = \{\texttt{true} \wedge \_^* \}$ *[FV-Meth]*
2. `timeout (() , 1);}`
   $\Phi'_C{=}\{\texttt{t1>1} \wedge \_^* \cdot (\epsilon \texttt{ \# t1})\}$ *[FV-Timeout]*
3. $\Phi'_C \sqsubseteq \Phi_{pre}^{addOneSugar(n)} \cdot \Phi_{post}^{addOneSugar(n)} \Leftrightarrow \texttt{t1>1} \wedge \_^* \cdot (\epsilon\texttt{\#t1}) \sqsubseteq \texttt{t>1} \wedge \_^* \cdot (\epsilon\texttt{\#t})$

---

4. `void addNSugar (int n){` *// initialize the state using the function precondition.*
   $\Phi_C{=}\Phi_{pre}^{addNSugar(n)} = \{\texttt{true} \wedge \_^*\}$ *[FV-Meth]*
5. `if (n == 0){`
   $\{\texttt{n=0} \wedge \_^*\}$ *[FV-Cond]*
6. `event ["EndSugar"];}`
   $\{\texttt{n=0} \wedge \_^* \cdot \texttt{ EndSugar}\}$ *[FV-Event]*
7. `else {`
   $\{\texttt{n}\neq\texttt{0} \wedge \_^*\}$ *[FV-Cond]*
8. `addOneSugar();`
   $\{\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1} \wedge \_^* \cdot (\epsilon \texttt{ \# t2})\}$ *[FV-Call]*
9. `addNSugar (n-1);}}`
   $\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1} \wedge \_^* \cdot (\epsilon \texttt{ \# t2}) \sqsubseteq \Phi_{pre}^{addNSugar(n-1)}$ *// TRS: precondition checked.*
   $\{\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1} \wedge \_^* \cdot (\epsilon \texttt{ \# t2}) \cdot \Phi_{post}^{addNSugar(n-1)}\}$ *[FV-Call]*
10. $\Phi'_C{=}\ (\texttt{n=0} \wedge \_^*\cdot\texttt{EndSugar}) \vee (\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1} \wedge \_^*\cdot(\epsilon\texttt{\#t2})\cdot\Phi_{post}^{addNSugar(n-1)})$
11. $\Phi'_C \sqsubseteq \Phi_{pre}^{addNSugar(n)} \Leftrightarrow$ *//TRS: postcondition checked, cf. Table 1*
    $(\texttt{n=0} \wedge \texttt{EndSugar}) \vee (\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1} \wedge (\epsilon \texttt{ \# t2}) \cdot \Phi_{post}^{addNSugar(n-1)}) \sqsubseteq \Phi_{post}^{addNSugar(n)}$

Fig. 4. The forward verification examples (t1 and t2 are fresh time variables).

2.2 Forward Verification. Fig. 4 demonstrates the forward verification of functions addOneSugar and addNSugar, defined in Fig. 1. The effects states are captured in the form of $\{\Phi_C\}$. To facilitate the illustration, we label the steps

by (1) to (11), and mark the deployed forward rules (cf. Sec. 4.1) in [gray]. The initial states (1) and (4) are obtained from the preconditions, by the [*FV-Meth*] rule. States (5)(7)(10) are obtained by [*FV-Cond*], which enforces the conditional constraints into the effects states, and unions the effects accumulated from two branches. State (6) is obtained by [*FV-Event*], which concatenates an event to the current effects. The intermediate states (8) and (9) are obtained by [*FV-Call*]. Before each function call, [*FV-Call*] invokes the TRS to check whether the current effects states satisfy callees' preconditions. If it is not satisfied, the verification fails; otherwise, it concatenates the callee's postcondition to the current states (the precondition check for step (8) is omitted here).

State (2) is obtained by [*FV-Timeout*], which adds a lower time-bound to an empty trace. After these state transformations, steps (3) and (11) invoke the TRS to check the inclusions between the final effects and the declared postconditions.

2.3 The TRS. Having *TimEffs* to be the specification language, and the forward verifier to reason about the actual behaviors, we are interested in the following verification problem: Given a program $\mathcal{P}$, and a temporal specification Φ', does the inclusions $\Phi^{\mathcal{P}} \sqsubseteq \Phi'$ holds? Typically, checking the inclusion/entailment between the concrete program effects $\Phi^{\mathcal{P}}$ and the expected property Φ' proves that: the program $\mathcal{P}$ will never lead to unsafe traces which violate Φ'.

Our TRS is an extension of Antimirov and Mosses's algorithm [12], which can be deployed to decide inclusions of two regular expressions (REs) through an iterated process of checking inclusions of their *partial derivatives* [13]. There are two basic rules: [*Disprove*] infers false from trivially inconsistent inclusions; and [*Unfold*] applies Definition 2 to generate new inclusions.

Definition 1 (Derivative). *Given any formal language S over an alphabet Σ and any string $u \in \Sigma^*$, the derivative of S with respect to u is defined as:*

$$u^{-1}S = \{w \in \Sigma^* \mid uw \in S\}.$$

Definition 2 (REs Inclusion). *For REs r and s, $r \preceq s \Leftrightarrow \forall(\mathtt{A} \in \Sigma).\mathtt{A}^{-1}(r) \preceq \mathtt{A}^{-1}(s)$.*

Definition 3 (*TimEffs* Inclusion). *For* TimEffs Φ_1 *and* Φ_2,

$$\Phi_1 \sqsubseteq \Phi_2 \Leftrightarrow \forall \mathtt{A}.\forall \mathtt{t} \geq 0.\ (\mathtt{A\#t})^{-1}\Phi_1 \sqsubseteq (\mathtt{A\#t})^{-1}\Phi_2.$$

Similarly, we defined Definition 3 for unfolding the inclusions between *TimEffs*, where $(\mathtt{A\#t})^{-1}\Phi$ is the partial derivative of Φ w.r.t the event `A` with the time bound `t`. Termination of the rewriting is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization* (cf. Table 5) [14]. Next, we use Table 1 to demonstrate how the TRS automatically proves the final effects of `main` satisfying its postcondition (shown at step (11) in Fig. 4). We mark the rewriting rules (cf. Sec. 5) in [gray].

In Table 1, step ① renames the time variables to avoid the name clashes between the antecedent and the consequent. Step ② splits the proof tree into two branches, according to the different arithmetic constraints, by rule [`LHS-OR`]. In the first branch, step ③ eliminates the event `ES` from the head of both sides, by rule [`UNFOLD`]. Step ④ proves the inclusion, because evidently the consequent $\mathtt{tR} \geq 0 \ \wedge\ \epsilon\mathtt{\#tR}$ contains ϵ when `tR=0`. In the second branch, step ⑤ eliminates a

Table 1. An inclusion proving example. (I) is the right hand side sub-tree of the the main rewriting proof tree. (ES stands for the event EndSugar)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{n=0} \wedge \epsilon \sqsubseteq \texttt{tR}\geq\texttt{0} \wedge \epsilon \,\#\, \texttt{tR}}\,④\,[\text{PROVE}]}{\texttt{n=0} \wedge \cancel{\texttt{ES}} \sqsubseteq \texttt{tR}\geq\texttt{0} \wedge \texttt{ES\#tR}}\,③\,[\text{UNFOLD}] \quad (I)}{(\texttt{n=0}\wedge\texttt{ES}) \vee (\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1}\wedge\texttt{tL}\geq\texttt{(n-1)}\wedge \epsilon\texttt{\#t2} \cdot \texttt{ES\#tL}) \sqsubseteq \texttt{tR}\geq\texttt{n} \wedge \texttt{ES\#tR}}\,②\,[\text{LHS-OR}]}{(\texttt{n=0} \wedge \texttt{ES}) \vee (\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1} \wedge (\epsilon \,\#\, \texttt{t2}) \cdot \Phi^{addNSugar(n-1)}_{post}) \sqsubseteq \Phi^{addNSugar(n)}_{post}}\,①\,[\text{RENAME}]$$

(I)

$$\dfrac{\dfrac{\dfrac{}{\texttt{t2>1}\wedge\texttt{tL}\geq\texttt{(n-1)} \wedge \texttt{tL=(tR-t2)} \Rightarrow \texttt{tR}\geq\texttt{n}}\,⑦\,[\text{PROVE}]}{\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1}\wedge\texttt{tL}\geq\texttt{(n-1)} \wedge \epsilon \sqsubseteq \texttt{tR}\geq\texttt{n} \wedge \epsilon}\,⑥\,[\text{UNFOLD}]\ \pi_u\texttt{:tL=(tR-t2)}}{\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1}\wedge\texttt{tL}\geq\texttt{(n-1)} \wedge \underline{\texttt{ES\#tL}} \sqsubseteq \texttt{tR}\geq\texttt{n} \wedge \underline{\texttt{ES\#(tR-t2)}}}\,⑤\,[\text{UNFOLD}]$$
$$\texttt{n}\neq\texttt{0}\wedge\texttt{t2>1}\wedge\texttt{tL}\geq\texttt{(n-1)} \wedge \cancel{\epsilon\texttt{\#t2}}\cdot \texttt{ES\#tL} \sqsubseteq \texttt{tR}\geq\texttt{n} \wedge \texttt{ES\#}\cancel{\texttt{tR}}$$

time duration ϵ#t2 from both sides. Therefore the rule [UNFOLD] subtracts a time duration from the consequent, i.e., (tR-t2). Similarly, step ⑥ eliminates ES#tL from the both sides, adding tL=(tR-t2) to the unification constraints. Step ⑦ proves t2>1∧tL≥(n-1)∧tL=(tR-t2)⇒tR≥n [4]; therefore, the proof succeed.

2.4 Verifying the Fischer's Mutual Exclusion Protocol. Fig. 5 presents the classical Fischer's mutually exclusion protocol, in C^t. Global variables x and cs indicate *'which process attempted to access the critical section most recently'* and *'the number of processes accessing the critical section'* respectively. The main procedure is a parallel composition of three processes, where d and e are two constants. Each process attempts to enter the critical section when x is -1, i.e. no other process is currently attempting. Once the process is active (i.e., reaches line 6), it sets x to its identity number i within d time units, captured by deadline(...,d). Then it idles for e time units, captured by delay(e) and then checks whether x still equals to i. If so, it safely enters the critical section. Otherwise, it restarts from the beginning. Quantitative timing constraint d<e plays an important role in this algorithm to guarantee mutual exclusion. One way to prove mutual exclusion is to show that cs≤1 is always true. Or, using event temporal logic, we can show that the occurrence of Critical always indicates the next event is Exit. We show in Sec. 6 that our prototype system can verify such algorithms symbolically.

```
1  var x := -1;
2  var cs:= 0;
3
4  void proc (int i) {
5   [x=-1] //block waiting until true
6   deadline(event["Update"(i)]{x:=i},d);
7     delay (e);
8     if (x==i) {
9       event["Critical"(i)]{cs:=cs+1};
10      event["Exit"(i)]{cs:=cs-1;x:=-1};
11      proc (i);
12    } else {proc (i);}}
13
14 void main ()
15 /* req: d<e ∧ ϵ          ens_a: true∧(cs≤1)*
16   ens_b: true∧ ((_*).Critical.Exit.(_*))* */
17 { proc(0) || proc(1) || proc(2); }
```

Fig. 5. Fischer's mutually exclusion algorithm.

[4] The proof obligations for arithmetic constraints are discharged by the Z3 solver [15].

3 Language and Specifications

3.1 The Target Language

We define the core language C^t in Fig. 6, which is built based on C syntax and provides support for timed behavioral patterns.

$$
\begin{array}{lrl}
(\textit{Program}) & \mathcal{P} & ::= (\alpha^*, \textit{meth}^*) \\
(\textit{Types}) & \iota & ::= \textit{int} \mid \textit{bool} \mid \textit{void} \\
(\textit{Method}) & \textit{meth} & ::= \iota\ mn\ (\iota\ x)^*\ \{\textbf{req}\ \Phi_{pre}\ \textbf{ens}\ \Phi_{post}\}\ \{e\} \\
(\textit{Values}) & v & ::= () \mid c \mid b \mid x \\
(\textit{Assignment}) & \alpha & ::= x := v \\
(\textit{Expressions}) & e & ::= v \mid \alpha \mid [v]e \mid mn(v^*) \mid e_1; e_2 \mid e_1 || e_2 \mid \textit{if}\ v\ e_1\ e_2 \mid \texttt{event}[\texttt{A}(v, \alpha^*)] \\
 & & \quad \mid \texttt{delay}[v] \mid e_1\ \texttt{timeout}[v]\ e_2 \mid e\ \texttt{deadline}[v] \mid e_1\ \texttt{interrupt}[v]\ e_2 \\
(\textit{Terms}) & t & ::= c \mid x \mid t_1 \texttt{+} t_2 \mid t_1 \texttt{-} t_2
\end{array}
$$

$c \in \mathbb{Z}$ $\quad b \in \mathbb{B}$ $\quad mn, x \in \textbf{var}$ $\quad (\textit{Action labels})\ \texttt{A} \in \Sigma$

Fig. 6. A core first-order imperative language with timed constructs via implicit clocks.

Here, c and b stand for integer and Boolean constants, mn and x are meta-variables, drawn from **var** (the countably infinite set of arbitrary distinct identifiers). A program $\mathcal{P}$ comprises a list of global variable initializations α^* and a list of method declarations $\textit{meth}^*$. Here, we use the $*$ superscript to denote a finite list of items, for example, x^* refers to a list of variables, $x_1, ..., x_n$. Each method $\textit{meth}$ has a name mn, an expression-oriented body e, also is associated with a precondition Φ_{pre} and a postcondition Φ_{post} (specification syntax is given in Fig. 7). C^t allows each iterative loop to be optimized to an equivalent tail-recursive method, where mutation on parameters is made visible to the caller.

Expressions comprise: values v; guarded processes $[v]e$, where if v is true, it behaves as e, else it idles until v becomes true; method calls $mn(v^*)$; sequential composition $e_1; e_2$; parallel composition $e_1 || e_2$, where e_1 and e_2 may communicate via shared variables; conditionals $\textit{if}\ \ v\ \ e_1\ \ e_2$; and event raising expressions $\texttt{event}[\texttt{A}(v, \alpha^*)]$ where the event $\texttt{A}$ comes from the finite set of event labels Σ. Without loss of generality, events can be further parametrized with one value v and a set of assignments α^* to update the mutable variables. Moreover, a number of timed constructs can be used to capture common real-time system behaviors, which are explained via operational semantics rules in Sec. 3.2.

3.2 Operational Semantics of C^t

To build the semantics of the system model, we define the notion of a configuration in Definition 4, to capture the global system state during system execution.

Definition 4 (System configuration). *A system configuration ζ is a pair $(\mathcal{S}, e)$ where $\mathcal{S}$ is a variable valuation function (or a stack) and e is an expression.*

A transition of the system is of the form $\zeta \xrightarrow{l} \zeta'$ where ζ and ζ' are the system configurations before and after the transition respectively. Transition labels l include: d, denoting a non-negative integer; τ, denoting an invisible

event; A, denoting an observable event. For example, $\zeta \xrightarrow{\mathrm{d}} \zeta'$ denotes a d time-units elapse. Next, we present the firing rules, associated with timed constructs.

Process $\texttt{delay}[v]$ idles for exactly t time units. Rule $[delay_1]$ states that the process may idle for any amount of time given it is less than or equal to t; Rule $[delay_2]$ states that the process terminates immediately when t becomes 0.

$$\frac{\mathrm{d}{\leq}v}{(\mathcal{S}, \texttt{delay}[v]) \xrightarrow{\mathrm{d}} (\mathcal{S}, \texttt{delay}[v\text{-d}])}\,[delay_1] \qquad \frac{}{(\mathcal{S}, \texttt{delay}[0]) \xrightarrow{\tau} (\mathcal{S}, ())}\,[delay_2]$$

In e_1 $\texttt{timeout}[v]$ e_2, the first observable event of e_1 shall occur before t time units; otherwise, e_2 takes over the control after exactly t time units. Note that the usage of timeout in Fig. 1 is a special case where e_1 never starts by default.

$$\frac{(\mathcal{S}, e_1) \xrightarrow{\mathtt{A}} (\mathcal{S}', e_1')}{(\mathcal{S}, e_1\ \texttt{timeout}[v]\ e_2) \xrightarrow{\mathtt{A}} (\mathcal{S}', e_1')}\,[to_1] \qquad \frac{(\mathcal{S}, e_1) \xrightarrow{\tau} (\mathcal{S}', e_1')}{(\mathcal{S}, e_1\ \texttt{timeout}[v]\ e_2) \xrightarrow{\tau} (\mathcal{S}', e_1'\ \texttt{timeout}[v]e_2)}\,[to_2]$$

$$\frac{(\mathcal{S}, e_1) \xrightarrow{\mathrm{d}} (\mathcal{S}, e_1') \qquad (\mathrm{d}{\leq}v)}{(\mathcal{S}, e_1\ \texttt{timeout}[v]\ e_2) \xrightarrow{\mathrm{d}} (\mathcal{S}, e_1'\ \texttt{timeout}[v\text{-d}]e_2)}\,[to_3] \qquad \frac{}{(\mathcal{S}, e_1\ \texttt{timeout}[0]e_2) \xrightarrow{\tau} (\mathcal{S}, e_2)}\,[to_4]$$

Process $\texttt{deadline}$ $[v]$ e behaves exactly as e except that it must terminate before t time units. The guarded process $[v]e$ behaves as e when v is true, otherwise it idles until v becomes true. Process e_1 $\texttt{interrupt}[v]$ e_2 behaves as e_1 until t time units, and then e_2 takes over. We leave the rest rules in [16].

$$\frac{(\mathcal{S}, e) \xrightarrow{\mathtt{A}/\tau} (\mathcal{S}', e')}{(\mathcal{S}, \texttt{deadline}[v]\ e) \xrightarrow{\mathtt{A}/\tau} (\mathcal{S}', \texttt{deadline}[v]\ e')}\,[ddl_1] \qquad \frac{(\mathcal{S}, e) \xrightarrow{l} (\mathcal{S}', v)}{(\mathcal{S}, \texttt{deadline}[v]\ e) \xrightarrow{l} (\mathcal{S}', v)}\,[ddl_2]$$

$$\frac{\mathcal{S} \models (v{=}true)}{(\mathcal{S}, [v]e) \xrightarrow{\tau} (\mathcal{S}, e)}\,[gu_1] \qquad \frac{(\mathcal{S}, e) \xrightarrow{\mathrm{d}} (\mathcal{S}, e') \quad (\mathrm{d}{\leq}v)}{(\mathcal{S}, \texttt{deadline}[v]\ e) \xrightarrow{\mathrm{d}} (\mathcal{S}, \texttt{deadline}[v\text{-d}]\ e')}\,[ddl_3]$$

$$\frac{\mathcal{S} \not\models (v{=}true)}{(\mathcal{S}, [v]e) \xrightarrow{\tau} (\mathcal{S}, [v]e)}\,[gu_2] \qquad \frac{(\mathcal{S}, e_1) \xrightarrow{\mathtt{A}/\tau} (\mathcal{S}', e_1')}{(\mathcal{S}, e_1\ \texttt{interrupt}[v]\ e_2) \xrightarrow{\mathtt{A}/\tau} (\mathcal{S}', e_1'\ \texttt{interrupt}[v]\ e_2)}\,[int_1]$$

$$\frac{(\mathcal{S}, e_1) \xrightarrow{l} (\mathcal{S}', v)}{(\mathcal{S}, e_1\ \texttt{interrupt}[v]\ e_2) \xrightarrow{l} (\mathcal{S}', v)}\,[int_2] \qquad \frac{}{(\mathcal{S}, e_1\ \texttt{interrupt}[0]\ e_2) \xrightarrow{\tau} (\mathcal{S}, e_2)}\,[int_3]$$

$$\frac{(\mathcal{S}, e_1) \xrightarrow{\mathrm{d}} (\mathcal{S}, e_1') \quad (\mathrm{d}{\leq}v)}{(\mathcal{S}, e_1\ \texttt{interrupt}[v]\ e_2) \xrightarrow{\mathrm{d}} (\mathcal{S}, e_1'\ \texttt{interrupt}[v\text{-d}]\ e_2)}\,[int_4]$$

3.3 The Specification Language

We plant *TimEffs* specifications into the Hoare-style verification system, using Φ_{pre} and Φ_{post} to capture the temporal pre/post conditions. As shown in Fig. 7, *TimEffs* can be constructed by a conditioned event sequence $\pi \wedge \theta$; or an effects disjunction $\Phi_1 \vee \Phi_2$. Timed sequences comprise *nil* ($\perp$); empty trace ϵ; single event ev; concatenation $\theta_1 \cdot \theta_2$; disjunction $\theta_1 \vee \theta_2$; parallel composition $\theta_1 || \theta_2$; a block waiting for a certain constraint to be satisfied $\pi ? \theta$. We introduce a new operator #, and $\theta \# t$ represents the trace θ takes t time units to complete, where t

$$
\begin{array}{rl}
(\textit{Timed Effects}) & \Phi ::= \pi \wedge \theta \mid \Phi_1 \vee \Phi_2 \\
(\textit{Event Sequences}) & \theta ::= \bot \mid \epsilon \mid ev \mid \theta_1 \cdot \theta_2 \mid \theta_1 \vee \theta_2 \mid \theta_1 || \theta_2 \mid \pi ? \theta \mid \theta \# t \mid \theta^\star \\
(\textit{Events}) & ev ::= \mathtt{A}(v, \alpha^*) \mid \tau(\pi) \mid \overline{\mathtt{A}} \mid \_ \\
(\textit{Pure}) & \pi ::= True \mid False \mid bop(t_1, t_2) \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg \pi \mid \pi_1 \Rightarrow \pi_2 \\
(\textit{Real-Time Terms}) & t ::= c \mid x \mid t_1 + t_2 \mid t_1 - t_2
\end{array}
$$

$c \in \mathbb{Z}$ $\quad$ $x \in \mathbf{var}$ $\quad$ (*Real Time Bound*) $\#$ $\quad$ (*Kleene Star*) $\star$

Fig. 7. Syntax of *TimEffs*.

is a *real-time term*. A timed sequence also can be constructed by $\theta^\star$, representing zero or more times repetition of the trace θ. For single events, $\mathtt{A}(v, \alpha^*)$ stands for an observable event with label $\mathtt{A}$, parameterized by v, and the assignment operations α^*; $\tau(\pi)$ is an invisible event, parameterized with a pure formula π[5].

Events can also be $\overline{\mathtt{A}}$, referring to all events which are not labeled using $\mathtt{A}$; and a wildcard $\_$, which matches to all the events. We use π to denote a pure formula which captures the (Presburger) arithmetic conditions on terms or program parameters. We use $bop(t_1, t_2)$ to represent binary atomic formulas of terms (including $=$, $>$, $<$, $\geq$ and $\leq$). Terms consist of constant integer values c; integer variables x; simple computations of terms, $t_1 + t_2$ and $t_1 - t_2$.

3.4 Semantic Model of Timed Effects

Let $d, \mathcal{S}, \varphi \models \Phi$ denote the model relation, i.e., a stack $\mathcal{S}$, a concrete execution trace φ take d time units to complete, and they satisfy the specification Φ.

$$
\begin{array}{ll}
d, \mathcal{S}, \varphi \models \Phi_1 \vee \Phi_2 & \textit{iff } d, \mathcal{S}, \varphi \models \Phi_1 \textit{ or } d, \mathcal{S}, \varphi \models \Phi_2 \\
d, \mathcal{S}, \varphi \models \pi \wedge \epsilon & \textit{iff } d{=}0 \textit{ and } [\![\pi]\!]_s{=}\mathit{True} \textit{ and } \varphi{=}[] \\
d, \mathcal{S}, \varphi \models \pi \wedge ev & \textit{iff } d{=}0 \textit{ and } [\![\pi]\!]_s{=}\mathit{True} \textit{ and } \varphi{=}[ev] \\
d, \mathcal{S}, \varphi \models \pi \wedge (\theta_1 \cdot \theta_2) & \textit{iff } \exists \varphi_1, \varphi_2.\ \varphi_1 {+\!+} \varphi_2{=}\varphi \textit{ and } \exists d_1, d_2.\ d_1{+}d_2{=}d \\
 & \quad \textit{s.t. } d_1, \mathcal{S}, \varphi_1 \models \pi \wedge \theta_1 \textit{ and } d_2, \mathcal{S}, \varphi_2 \models \pi \wedge \theta_2 \\
d, \mathcal{S}, \varphi \models \pi \wedge (\theta_1 \vee \theta_2) & \textit{iff } d, \mathcal{S}, \varphi \models \pi \wedge \theta_1 \textit{ or } d, \mathcal{S}, \varphi \models \pi \wedge \theta_2 \\
d, \mathcal{S}, \varphi \models \pi \wedge (ev_1 \cdot \theta_1) || (ev_2 \cdot \theta_2) & \textit{iff } d, \mathcal{S}, \varphi \models \pi \wedge ev_1 \cdot (\theta_1 || (ev_2 \cdot \theta_2)) \textit{ or} \\
 & \quad d, \mathcal{S}, \varphi \models \pi \wedge ev_2 \cdot ((ev_1 \cdot \theta_1) || \theta_2) \\
d, \mathcal{S}, \varphi \models \pi \wedge (ev \cdot \theta_1) || (ev \cdot \theta_2) & \textit{iff } d, \mathcal{S}, \varphi \models \pi \wedge ev \cdot (\theta_1 || \theta_2) \\
d, \mathcal{S}, \varphi \models \pi \wedge (\epsilon \# t_1) || (\epsilon \# t_2) & \textit{iff } d, \mathcal{S}, \varphi \models (\pi \wedge t_1 \geq t_2) \wedge (\epsilon \# t_1) \cdot (\epsilon \#(t_1 - t_2)) \textit{ or} \\
 & \quad d, \mathcal{S}, \varphi \models (\pi \wedge t_1 < t_2) \wedge (\epsilon \# t_2) \cdot (\epsilon \#(t_2 - t_1)) \\
d, \mathcal{S}, \varphi \models \pi \wedge \pi_1 ? \theta & \textit{iff } [\![\pi_1]\!]_s{=}\mathit{True}, d, \mathcal{S}, \varphi \models \pi \wedge \theta \textit{ or} \\
 & \quad [\![\pi_1]\!]_s{=}\mathit{False}, d, \mathcal{S}, \varphi \models \pi \wedge \pi_1 ? \theta \\
d, \mathcal{S}, \varphi \models \pi \wedge \theta \# t & \textit{iff } [\![\pi \wedge t{\geq}0]\!]_s{=}\mathit{True},\ \exists \theta_1, \theta_2.\ \theta_1 \cdot \theta_2{=}\theta,\ \textit{fresh } t_1, t_2, \textit{s.t.} \\
 & \quad d, \mathcal{S}, \varphi \models (\pi \wedge t_1 {\geq} 0 \wedge t_2 {\geq} 0 \wedge t_1 {+} t_2 {=} t) \wedge (\theta_1 \# t_1) \cdot (\theta_2 \# t_2) \\
d, \mathcal{S}, \varphi \models \pi \wedge \theta^\star & \textit{iff } d, \mathcal{S}, \varphi \models \pi \wedge \epsilon \textit{ or } d, \mathcal{S}, \varphi \models \pi \wedge \theta \cdot \theta^\star \\
d, \mathcal{S}, \varphi \models \mathit{false} & \textit{iff } [\![\pi]\!]_s{=}\mathit{False} \textit{ or } \varphi{=}\bot
\end{array}
$$

Fig. 8. Semantics of *TimEffs*.

[5] The difference between $\tau(\pi)$ and $\pi?$ is: $\tau(\pi)$ marks an assertion which leads to false ($\bot$) if π is not satisfied, whereas $\pi?$ waits until π is satisfied.

To define the model, *var* is the set of program variables, *val* is the set of primitive values; and d, $\mathcal{S}$, φ are drawn from the following concrete domains: d: $\mathbb{N}$, $\mathcal{S}$: $var \rightarrow val$ and φ: *list of event*. As shown in Fig. 8, ++ appends event sequences; [] describes the empty sequences, $[ev]$ represents the singleton sequence contains event ev; $[\![\pi]\!]_{\mathcal{S}} = True$ represents π holds on the stack $\mathcal{S}$. Notice that, simple events, i.e., without #, are taken to be happening in instant time.

3.5 Expressiveness. *TimEffs* draw similarities to metric temporal logic (MTL), which is derived from LTL, where a set of non-negative real numbers is added to temporal modal operators. As shown in Table 2, we are able to encode MTL operators into *TimEffs*, making it more intuitive and readable. The basic modal operators are: $\Box$ for "globally"; $\Diamond$ for "finally"; $\bigcirc$ for "next"; $\mathcal{U}$ for "until", and their past time reversed versions: $\overleftarrow{\Box}$; $\overleftarrow{\Diamond}$; and $\ominus$ for "previous"; $\mathcal{S}$ for "since". I in MTL is the time interval with concrete upper/lower bounds; whereas in *TimEffs* they can be symbolic bounds which are dependent on program inputs.

Table 2. Examples for converting MTL formulae into *TimEffs* with $\mathtt{t} \in I$ applied.

Φ_{post}	$\Box_I \mathtt{A} \equiv (\mathtt{A}^\star)\#\mathtt{t}$	$\Diamond_I \mathtt{A} \equiv (\_^\star \cdot \mathtt{A})\#\mathtt{t}$	$\bigcirc_I \mathtt{A} \equiv (\_)\#\mathtt{t} \cdot \mathtt{A}$	$\mathtt{A}\mathcal{U}_I \mathtt{B} \equiv (\mathtt{A}^\star)\#\mathtt{t} \cdot \mathtt{B}$
Φ_{pre}	$\overleftarrow{\Box}_I \mathtt{A} \equiv (\mathtt{A}^\star)\#\mathtt{t}$	$\overleftarrow{\Diamond}_I \mathtt{A} \equiv (\mathtt{A} \cdot \_^\star)\#\mathtt{t}$	$\ominus_I \mathtt{A} \equiv \mathtt{A} \cdot ((\_)\#\mathtt{t})$	$\mathtt{A}\mathcal{S}_I \mathtt{B} \equiv \mathtt{B} \cdot ((\mathtt{A}^\star)\#\mathtt{t})$

4 Automated Forward Verification

4.1 Forward Rules

Forward rules are in the Hoare-style triples $\mathcal{S} \vdash \{\Pi, \Theta\}\ e\ \{\Pi', \Theta'\}$, where $\mathcal{S}$ is the stack environment; $\{\Pi, \Theta\}$ and $\{\Pi', \Theta'\}$ are program states, i.e., disjunctions of conditioned event sequence $\pi \wedge \theta$. The meaning of the transition is: $\{\Pi', \Theta'\} = \bigcup_{i=0}^{|\{\Pi,\Theta\}|-1} \{\Pi'_i, \Theta'_i\}$ *where* $(\pi_i \wedge \theta_i) \in \{\Pi, \Theta\}$ *and* $\vdash \{\pi_i, \theta_i\}\ e\ \{\Pi'_i, \Theta'_i\}$[6].

We here present the rules for time-related constructs and leave the rest rules in [16]. Rule [*FV-Delay*] creates a trace $\epsilon\#t$, where t is fresh, and concatenates it to the current program state, together with the additional constraint $t{=}v$. Rule [*FV-Deadline*] computes the effects from e and adds an upper time-bound to the results. Rule [*FV-Timeout*] computes the effects from e_1 and e_2 using the starting state $\{\pi, \epsilon\}$. The final state is an union of possible effects with corresponding time bounds and arithmetic constraints. Note that, $hd(\Theta_1)$ and $tl(\Theta_1)$ return the event *head* (cf. Definition 6), and the tail of Θ_1 respectively.

$$\frac{\theta' = \theta \cdot (\epsilon\#t) \quad (t\ \textit{is fresh})}{\mathcal{S} \vdash \{\pi, \theta\}\ \mathtt{delay}[v]\ \{\pi \wedge (t{=}v), \theta'\}}\ [\textit{FV-Delay}] \qquad \frac{\mathcal{S} \vdash \{\pi, \epsilon\}\ e\ \{\Pi_1, \Theta_1\} \quad (t\ \textit{is fresh})}{\mathcal{S} \vdash \{\pi, \theta\}\ \mathtt{deadline}[v]\ e\ \{\Pi_1 \wedge (t{\leq}v), \theta \cdot (\Theta_1\#t)\}}\ [\textit{FV-Deadline}]$$

$$\frac{\mathcal{S} \vdash \{\pi, \epsilon\}\ e_1\ \{\Pi_1, \Theta_1\} \qquad \mathcal{S} \vdash \{\pi, \epsilon\}\ e_2\ \{\Pi_2, \Theta_2\} \qquad (t_1, t_2\ \textit{are fresh}) \quad \{\Pi_f, \Theta_f\} = \{\Pi_1 \wedge t_1{<}v, (hd(\Theta_1)\#t_1) \cdot tl(\Theta_1)\} \cup \{\Pi_2 \wedge t_2{=}v, (\epsilon\#t_2) \cdot \Theta_2\}}{\mathcal{S} \vdash \{\pi, \theta\}\ e_1\ \mathtt{timeout}[v]\ e_2\ \{\Pi_f, \theta \cdot \Theta_f\}}\ [\textit{FV-Timeout}]$$

$$\frac{\mathcal{S} \vdash \{\pi, \epsilon\}\ e_1\ \{\Pi, \Theta\} \qquad \Delta = \bigcup_{i=0}^{|\{\Pi,\Theta\}|-1} \aleph_{Interleave}^{Interrupt(v,\pi_i)}(\theta_i, \epsilon) \qquad \mathcal{S} \vdash \{\Delta\}\ e_2\ \{\Pi', \Theta'\}}{\mathcal{S} \vdash \{\pi, \theta\}\ e_1\ \mathtt{interrupt}[v]\ e_2\ \{\Pi', \theta \cdot \Theta'\}}\ [\textit{FV-Interrupt}]$$

[6] $|\{\Pi, \Theta\}|$ is the size of $\{\Pi, \Theta\}$, i.e., the count of conditioned event sequence $\pi \wedge \theta$.

[*FV-Interrupt*] computes the interruption interleaves of e_1's effects, which come from the over-approximation of all the possibilities. For example, for trace $\mathtt{A} \cdot \mathtt{B}$, the interruption with time t creates three possibilities: $(\epsilon \# t) \vee (\mathtt{A} \# t) \vee ((\mathtt{A} \cdot \mathtt{B}) \# t)$. Then the rule continues to compute the effects of e_2; lastly, it prepends the original history θ to the final results. Algorithm 1 presents the interleaving algorithm for *interruptions*, where + unions program states (cf. Definition 7 and Definition 8 for *fst* and D functions).

Algorithm 1: Interruption Interleaving

Input: $v, \pi, \theta, \theta_{his}$
Output: Program States: Δ
function $\aleph^{Interrupt(v,\pi)}_{Interleave}(\theta, \theta_{his})$
  $\Delta \leftarrow []$
  foreach $f \in fst_\pi(\theta)$ **do**
    $\phi \leftarrow \pi \wedge (t < v) \wedge (\theta_{his} \# t)$
    $\theta' \leftarrow D^\pi_f(\theta)$
    $\theta'_{his} \leftarrow \theta_{his} \cdot f$
    $\Delta' \leftarrow \aleph^{Interrupt(v,\pi)}_{Interleave}(\theta', \theta'_{his})$
    $\Delta \leftarrow \Delta + \phi + \Delta'$
  return Δ

Theorem 1 (Soundness of Forward Rules). *Given any system configuration $\zeta = (\mathcal{S}, e)$, by applying the operational semantics rules, if $(\mathcal{S}, e) \rightarrow^* (\mathcal{S}', v)$ has execution time d and produces event sequence φ; and for any history effect $\pi \wedge \theta$, such that $d_1, \mathcal{S}, \varphi_1 \models (\pi \wedge \theta)$, and the forward verifier reasons $\mathcal{S} \vdash \{\pi, \theta\} e \{\Pi, \Theta\}$, then $\exists (\pi' \wedge \theta') \in \{\Pi, \Theta\}$ such that $(d_1 + d), \mathcal{S}', (\varphi_1 +\!+ \varphi) \models (\pi' \wedge \theta')$. ($\zeta \rightarrow^* \zeta'$ denotes the reflexive, transitive closure of $\zeta \rightarrow \zeta'$.)*

Proof. See the technical report [16].

5 Temporal Verification via a TRS

The TRS is an automated entailment checker to prove language inclusions between *TimEffs*. It is triggered prior to function calls for the precondition checking; and by the end of verifying a function, for the post condition checking.

Given two effects Φ_1 and Φ_2, the TRS decides if the inclusion $\Phi_1 \sqsubseteq \Phi_2$ is valid. During the effects rewriting process, the inclusions are in the form of $\Gamma \vdash \Phi_1 \sqsubseteq^\Phi \Phi_2$, a shorthand for: $\Gamma \vdash \Phi \cdot \Phi_1 \sqsubseteq \Phi \cdot \Phi_2$. To prove such inclusions is to check whether all the possible timed traces in the antecedent Φ_1 are legitimately allowed in the timed traces described by the consequent Φ_2. Here Γ is the proof context, i.e., a set of effects inclusion hypothesis; and Φ is the history effects from the antecedent that have been used to match the effects from the consequent. The checking is initially invoked with $\Gamma = \emptyset$ and $\Phi = True \wedge \epsilon$.

Effects Disjunctions. An inclusion with a disjunctive antecedent succeeds if both disjunctions entail the consequent. An inclusion with a disjunctive consequent succeeds if the antecedent entails either of the disjunctions.

$$\frac{\Gamma \vdash \Phi_1 \sqsubseteq \Phi \qquad \Gamma \vdash \Phi_2 \sqsubseteq \Phi}{\Gamma \vdash \Phi_1 \vee \Phi_2 \sqsubseteq \Phi}\ [LHS\text{-}OR] \qquad \frac{\Gamma \vdash \Phi \sqsubseteq \Phi_1 \quad or \quad \Gamma \vdash \Phi \sqsubseteq \Phi_2}{\Gamma \vdash \Phi \sqsubseteq \Phi_1 \vee \Phi_2}\ [RHS\text{-}OR]$$

Now, the inclusions are disjunction-free formulas. Next we provide the definitions and key implementations of auxiliary functions Nullable, First and Derivative. Intuitively, the Nullable function $\delta_\pi(\theta)$ returns a Boolean value indicating

whether $\pi \wedge \theta$ contains the empty trace; the First function $fst_\pi(\theta)$ computes a set of initial *heads*, denoted as h, of $\pi \wedge \theta$; the Derivative function $D_h^\pi(\theta)$ computes a next-state effects after eliminating the head h from the current effects $\pi \wedge \theta$.

Definition 5 (Nullable [7]). *Given any* $\Phi = \pi \wedge \theta$, $\delta_\pi(\theta) : bool = \begin{cases} true & if \ \epsilon \in [\![\pi \wedge \theta]\!] \\ false & if \ \epsilon \notin [\![\pi \wedge \theta]\!] \end{cases}$

$$\delta_\pi(\bot)=\delta_\pi(ev)=false \quad \delta_\pi(\epsilon)=\delta(\theta^\star)=true \quad \delta_\pi(\pi'?\theta)=\delta_\pi(\theta) \quad \delta_\pi(\theta_1 \vee \theta_2)=\delta(\theta_1)\vee\delta(\theta_2)$$
$$\delta_\pi(\theta \cdot \theta_2)=\delta(\theta_1)\wedge\delta(\theta_2) \quad \delta_\pi(\theta_1||\theta_2)=\delta(\theta_1)\wedge\delta(\theta_2) \quad \delta_\pi(\theta \# t)=SAT(\pi \wedge (t=0)) \wedge \delta_\pi(\theta)$$

Definition 6 (Heads). *If h is a head of $\pi \wedge \theta$, then there exist π' and θ', such that $\pi \wedge \theta = \pi' \wedge (h \cdot \theta')$. A head can be t, denoting a pure time passing; $A(v, \alpha^*)$, denoting an instant event passing; or $(A(v, \alpha^*), t)$, denoting an event passing which takes time t.*

Definition 7 (First). *Given any $\Phi = \pi \wedge \theta$, $fst_\pi(\theta)$ returns a set of heads, be the set of initial elements derivable from effects $\pi \wedge \theta$, where (t' is fresh):*

$$fst_\pi(\bot)=fst_\pi(\epsilon)=\{\} \quad fst_\pi(A(v,\alpha^*))=\{A(v,\alpha^*)\} \quad fst_\pi(\epsilon \# t)=\{t\} \quad fst_\pi(\theta^\star)=fst_\pi(\theta)$$
$$fst_\pi(\theta \# t)=\{(A(v,\alpha^*), t') \mid A(v,\alpha^*) \in fst_\pi(\theta)\} \quad fst_\pi(\theta_1 \vee \theta_2)=fst_\pi(\theta_1) \cup fst_\pi(\theta_2)$$
$$fst_\pi(\pi'?\theta)=fst_\pi(\theta) \quad fst_\pi(\theta_1||\theta_2)=fst_\pi(\theta_1) \cup fst_\pi(\theta_2)$$
$$fst_\pi(\theta_1 \cdot \theta_2)= \begin{cases} fst_\pi(\theta_1) \cup fst_\pi(\theta_2) & if \ \ \delta(\theta_1)=true \\ fst_\pi(\theta_1) & if \ \ \delta(\theta_1)=false \end{cases}$$

Definition 8 (*TimEffs* Partial Derivative). *Given any $\Phi = \pi \wedge \theta$, the partial derivative $D_h^\pi(\theta)$ computes the effects for the left quotient $h^{-1}(\pi \wedge \theta)$, cf. Definition 1.*

$$D_h^\pi(\bot)=D_h^\pi(\epsilon)=False \wedge \bot \quad D_h^\pi(A(v,\alpha^*))=(\pi \wedge (h=A(v,\alpha^*))) \wedge \epsilon \quad D_h^\pi(\theta^\star)=D_h^\pi(\theta)\cdot\theta^\star$$
$$D^\pi_{\tau(\pi_1)}(\pi'?\theta)= \begin{cases} \pi \wedge \pi'?\theta & if \ \ \pi_1 \not\Rightarrow \pi' \\ \pi \wedge \theta & if \ \ \pi_1 \Rightarrow \pi' \end{cases} \quad D_h^\pi(\theta_1 \cdot \theta_2)= \begin{cases} D_h^\pi(\theta_1)\cdot\theta_2 \vee D_h^\pi(\theta_2) & if \ \ \delta_\pi(\theta_1)=true \\ D_h^\pi(\theta_1)\cdot\theta_2 & if \ \ \delta_\pi(\theta_1)=false \end{cases}$$
$$D^\pi_{(A(v,\alpha^*),t)}(\theta) = \bigvee \{ D^{\pi'}_{A(v,\alpha^*)}(\theta') \mid (\pi' \wedge \theta') \in D_t^\pi(\theta) \}$$
$$D_t^\pi(\theta \# t')=(\pi \wedge t+t''=t') \wedge \theta \# t'' \quad (t'' \ is \ fresh) \quad D_h^\pi(\theta_1 \vee \theta_2)=D_h^\pi(\theta_1) \vee D_h^\pi(\theta_2)$$
$$D^\pi_{A(v,\alpha^*)}(\theta \# t)=\bigvee \{ (\pi' \wedge (\theta' \# t)) \mid (\pi' \wedge \theta') \in D^\pi_{A(v,\alpha^*)}(\theta) \} \quad D_h^\pi(\theta_1||\theta_2)=\bar{\bar{D}}_h^\pi(\theta_1)||\bar{\bar{D}}_h^\pi(\theta_2)$$

Notice that the derivatives of a parallel composition makes use of the *Parallel Derivative* $\bar{\bar{D}}_h^\pi(\theta)$, defined as follows: $\bar{\bar{D}}_h^\pi(\theta)= \begin{cases} \pi \wedge \theta & if \ \ D_h^\pi(\pi \wedge \theta) = (False \wedge \bot) \\ D_h^\pi(\theta) & otherwise \end{cases}$

5.1 Rewriting Rules. Given the well-defined auxiliary functions above, we now discuss the key rewriting rules that deployed in effects inclusion proofs.

$$\frac{}{\Gamma \vdash \pi \wedge \bot \sqsubseteq \Phi} \ [\texttt{Bot-LHS}] \qquad \frac{\Phi \neq \pi \wedge \bot}{\Gamma \vdash \Phi \not\sqsubseteq \pi \wedge \bot} \ [\texttt{Bot-RHS}]$$

$$\frac{\delta_{\pi_1}(\theta_1) \wedge \neg\delta_{\pi_2}(\theta_2)}{\Gamma \vdash \pi_1 \wedge \theta_1 \not\sqsubseteq \pi_2 \wedge \theta_2} \ [\texttt{DISPROVE}] \qquad \frac{\pi_1 \Rightarrow \pi_2 \qquad fst_{\pi_1}(\theta_1) = \{\}}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2} \ [\texttt{PROVE}]$$

[7] $SAT(\pi)$ stands for querying the Z3 theorem prover to check the satisfiability of π.

Axiom rules [Bot-LHS] and [Bot-RHS] are analogous to the standard propositional logic, $\bot$ (referring to *false*) entails any effects, while no *non-false* effects entails $\bot$. [DISPROVE] is used to disprove the inclusions when the antecedent is nullable, while the consequent is not nullable.

We use two rules to prove an inclusion: (i) [PROVE] is used when the antecedent has no head; and (ii) [REOCCUR] proves an inclusion when there exist inclusion hypotheses in the proof context Γ, which are able to soundly prove the current goal. [UNFOLD] is the inductive step of unfolding the inclusions. The proof of the original inclusion succeeds if all the derivative inclusions succeed.

$$\frac{(\pi_1 \wedge \theta_1 \sqsubseteq \pi_3 \wedge \theta_3) \in \Gamma \qquad (\pi_3 \wedge \theta_3 \sqsubseteq \pi_4 \wedge \theta_4) \in \Gamma \qquad (\pi_4 \wedge \theta_4 \sqsubseteq \pi_2 \wedge \theta_2) \in \Gamma}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2}\ [\texttt{REOCCUR}]$$

$$\frac{H{=}fst_{\pi_1}(\theta_1) \qquad \Gamma'{=}\Gamma, (\pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2) \qquad \forall h{\in}H.\ (\Gamma' \vdash D_h^{\pi_1}(\theta_1) \sqsubseteq D_h^{\pi_2}(\theta_2))}{\Gamma \vdash \pi_1 \wedge \theta_1 \sqsubseteq \pi_2 \wedge \theta_2}\ [\texttt{UNFOLD}]$$

Theorem 2 (Termination of the TRS). *The TRS is terminating.*

Proof. See the technical report [16].

Theorem 3 (Soundness of the TRS). *Given an inclusion $\Phi_1 \sqsubseteq \Phi_2$, if the TRS returns TRUE with a proof, then $\Phi_1 \sqsubseteq \Phi_2$ is valid.*

Proof. See the technical report [16].

6 Implementation and Evaluation

To show the feasibility, we prototype our automated verification system using OCaml (∼5k LOC); and prove soundness for both the forward verifier and the TRS. We set up two experiments to evaluate our implementation: i) functionality validation via verifying symbolic timed programs; and ii) comparison with PAT [17] and Uppaal [3] using real-life Fischer's mutual exclusion algorithm. Experiments are done on a MacBook with a 2.6 GHz 6-Core Intel i7 processor. The source code and the evaluation benchmark are openly accessible from [18].

6.1 Experimental Results for Symbolic Timed Models. We manually annotate *TimEffs* specifications for a set of synthetic examples (for about 54 programs), to test the main contributions, including: computing effects from symbolic timed programs written in C^t; and the inclusion checking for *TimEffs* with the parallel composition, block waiting operator and shared global variables.

Table 3 presents the evaluation results for another 16 C^t programs[8], and the annotated temporal specifications are in a 1:1 ratio for succeeded/failed cases. The table records: **No.**, index of the program; **LOC**, lines of code; **Forward(ms)**, effects computation time; **#Prop(✓)**, number of valid properties; **Avg-Prove(ms)**, average proving time for the valid properties; **#Prop(✗)**, number of invalid properties; **Avg-Dis(ms)**, average disproving time for the invalid properties; **#AskZ3**, number of querying Z3 through out the experiments.

[8] All programs contain timed constructs, conditionals, and parallel compositions.

Table 3. Experimental Results for Manually Constructed Synthetic Examples.

No.	LOC	Forward(ms)	#Prop(✓)	Avg-Prove(ms)	#Prop(✗)	Avg-Dis(ms)	#AskZ3
1	26	0.006	5	52.379	5	21.31	77
2	37	43.955	5	83.374	5	52.165	188
3	44	32.654	5	52.524	5	33.444	104
4	72	202.181	5	82.922	5	55.971	229
5	98	42.706	7	149.345	7	60.325	396
6	134	403.617	7	160.932	7	292.304	940
7	133	51.492	7	17.901	7	47.643	118
8	173	57.114	7	40.772	7	30.977	128
9	182	872.995	9	252.123	9	113.838	1142
10	210	546.222	9	146.341	9	57.832	570
11	240	643.133	9	146.268	9	69.245	608
12	260	1032.31	9	242.699	9	123.054	928
13	265	12558.05	11	150.999	11	117.288	2465
14	286	12257.834	11	501.994	11	257.800	3090
15	287	1383.034	11	546.064	11	407.952	1489
16	337	49873.835	11	1863.901	11	954.996	15505

Observations: i) the proving/disproving time increases when the effect computation time increases because larger **Forward(ms)** indicates the higher complexity w.r.t the timed constructs, which complicates the inclusion checking; ii) while *the number of querying Z3 per property* (**#AskZ3**/(**#Prop(✓)**+**#Prop(✗)**)) goes up, the proving/disproving time goes up. Besides, we notice that iii) the disproving times for invalid properties are constantly lower than the proving process, regardless of the program's complexity, which is as expected in a TRS.

6.2 Verifying Fischer's mutual exclusion algorithm. As shown in Fig. 4, the data in columns **PAT(s)** and **Uppaal(s)** are drawn from prior work [19], which indicate the time to prove Fischer's mutual exclusion w.r.t the number of processes (**#Proc**) in PAT and Uppaal respectively. For our system, based on the implementation presented in Fig. 5, we are able to prove the mutual exclusion properties, given the arithmetic constraint `d<e`. Besides, the system disproves mutual exclusion when `d`$\leq$`e`. We record the proving (**Prove(s)**) and disproving (**Disprove(s)**) time and their number of uniquely querying Z3 (**#AskZ3-u**).

Table 4. Comparison with PAT via verifying Fischer's mutual exclusion algorithm

#Proc	Prove(s)	#AskZ3-u	Disprove(s)	#AskZ3-u	PAT(s)	Uppaal(s)
2	0.09	31	0.110	37	$\leq$0.05	$\leq$0.09
3	0.21	35	0.093	42	$\leq$0.05	$\leq$0.09
4	0.46	63	0.120	47	0.05	0.09
5	25.0	84	0.128	52	0.15	0.19

Observations: i) automata-based model checkers (both PAT and Uppaal) are vastly efficient when given concrete values for constants `d` and `e`; however ii) our proposal is able to symbolically prove the algorithm by only providing the constraints of `d` and `e`, which cannot be achieved by existing model checkers; ii) our verification time largely depends on the number of querying Z3, which is optimized in our implementation by keeping a table for already queried constraints.

6.3 Case Study: Prove it when Reoccur. Termination of TRS is guaranteed because the set of derivatives to be considered is finite, and possible cycles are detected using *memorization* [14], demonstrated in Table 5. In step ②, in order to eliminate the first event B, $\texttt{A}^\star\texttt{\#tR}$ has to be reduced to ϵ, therefore the RHS time constraint has been strengthened to tR=0. Looking at the sub-tree (I), in step ⑤, tL and tR are split into $\texttt{tL}^1+\texttt{tL}^2$ and $\texttt{tR}^1+\texttt{tR}^2$. Then in step ⑥, $\texttt{A\#tL}^1$ together with $\texttt{A\#tR}^1$ are eliminated, unifying $\texttt{tL}^1$ and $\texttt{tR}^1$ by adding the side constraint $\texttt{tL}^1=\texttt{tR}^1$. In step ⑧, we observe the proposition is isomorphic with one of the the previous step, marked using (‡). Hence we apply the rule [REOCCUR] to prove it with a succeed side constraints entailment.

Table 5. The reoccurrence proving example. (I) is the left hand side sub-tree of the main rewriting proof tree.

$$\dfrac{\dfrac{(I) \qquad \dfrac{\dfrac{}{\texttt{True} \wedge \epsilon \sqsubseteq \texttt{tR=0} \wedge \epsilon}\,④\,[\texttt{PROVE}]}{\texttt{True} \wedge \cancel{\texttt{B}} \sqsubseteq \texttt{tR=0} \wedge \cancel{\epsilon}\cancel{\texttt{B}}}\,③\,[\texttt{Normal}]}{\texttt{True} \wedge \texttt{B} \sqsubseteq \texttt{tR<4} \wedge (\texttt{A}^\star\texttt{\#tR}) \cdot \texttt{B}}\,②\,[\texttt{UNFOLD}]}{(\texttt{tL<3} \wedge (\texttt{A}^\star\texttt{\#tL}) \cdot \texttt{B}) \vee (\texttt{True} \wedge \texttt{B}) \sqsubseteq \texttt{tR<4} \wedge (\texttt{A}^\star\texttt{\#tR}) \cdot \texttt{B}}\,①\,[\texttt{OR-LHS}]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(I):\ \texttt{tL<3} \wedge \texttt{tL}^1+\texttt{tL}^2=\texttt{tL} \wedge \texttt{tR}=\texttt{tR}^1+\texttt{tR}^2 \wedge \texttt{tL}^1=\texttt{tR}^1 \wedge \texttt{tL}^2=\texttt{tR}^2 \Rightarrow \texttt{tR<4}}\,⑧\,[\texttt{REOCCUR}]}{\texttt{tL<3} \wedge (\texttt{A}^\star\texttt{\#tL}^2) \cdot \texttt{B} \sqsubseteq \texttt{tR<4} \wedge (\texttt{A}^\star\texttt{\#tR}^2) \cdot \texttt{B} \quad (‡)}\,⑦\,[\texttt{UNFOLD}]}{\texttt{tL<3} \wedge \cancel{\texttt{A\#tL}^1} \cdot \texttt{A}^\star\texttt{\#tL}^2 \cdot \texttt{B} \sqsubseteq \texttt{tR<4} \wedge \cancel{\texttt{A\#tR}^1} \cdot \texttt{A}^\star\texttt{\#tR}^2 \cdot \texttt{B}}\,⑥\,[\texttt{UNFOLD}]\ \pi_u{:}\texttt{tL}^1=\texttt{tR}^1}{\texttt{tL<3} \wedge (\texttt{A\#tL}^1 \cdot \texttt{A}^\star\texttt{\#tL}^2) \cdot \texttt{B} \sqsubseteq \texttt{tR<4} \wedge (\texttt{A\#tR}^1 \cdot \texttt{A}^\star\texttt{\#tR}^2) \cdot \texttt{B}}\,⑤\,[\texttt{SPLIT}]\texttt{tL}^1+\texttt{tL}^2=\texttt{tL} \wedge \texttt{tR}^1+\texttt{tR}^2=\texttt{tR}$$
$$\texttt{tL<3} \wedge (\texttt{A}^\star\texttt{\#tL}) \cdot \texttt{B} \sqsubseteq \texttt{tR<4} \wedge (\texttt{A}^\star\texttt{\#tR}) \cdot \texttt{B} \quad (‡)$$

6.4 Discussion. Our implementation is the first that proves the inclusion of symbolic TAs, which is considered significant because it overcomes the following main limitations of traditional timed model checking: i) TAs cannot be used to specify/verify incompletely specified systems (i.e., whose timing constants have yet to be known) and hence cannot be used in early design phases; ii) verifying a system with a set of timing constants usually requires enumerating all of them if they are supposed to be integer-valued; iii) TAs cannot be used to verify systems with timing constants to be taken in a real-valued dense interval.

7 Related Work

7.1 Verification Framework. This work draws the most similarities to [20], which also deploys a forward verifier and a TRS for extended regular expressions. The differences are: i) [20] targets general-purpose sequential programs without shared variables, whereas this work targets time-critical programs with the presence of concurrency and global shared states; ii) the dependent values in [20] denote the number of repetitions of a trace, whereas in this work, they abstract the real-time bounds; iii) in this work, the TRS supports inclusion checking for the block waiting operator π? and the concurrent composition ||. These are essential in timed verification (or, more generally, for distributed systems), which are not supported in [20] or any other TRS-related works.

7.2 Specifications and Real-Time Verification. Apart from compositional modelling for real-time systems based on timed-process algebras, such as Timed CSP [8] and CCS+Time [21], there have been a number of translation-based approaches on building verification support for timed-process algebras. For example, in [8], Timed CSP is translated to TAs (TAs) so that the model checker Uppaal [3] can be applied. On the other hand, all the translation-based approaches share the common problem: the overhead introduced by the complex translation makes it particularly inefficient when *disproving* properties. We are of the opinion that in that the goal of verifying real-time systems, in particular safety-critical systems is to check logical temporal properties, which can be done without constructing the whole reachability graph or the full power of model-checking. We consider our approach is simpler as it is based directly on constraint-solving techniques and can be fairly efficient in verifying systems consisting of many components as it avoids to explore the whole state-space [20,22].

This work draws similarities to Real-Time Maude [23], which complements timed automata with more expressive object-oriented specifications.

7.3 Clock Manipulation and Zone-based Bisimulation. The concept of implicit clocks has also been used in time Petri nets, and implemented in a several model checking engines, e.g., [24]. On the other hand, to make model checking more efficient with *explicit* clocks, [25–28] work on dynamically deleting or merging clocks. Our work also draw connections with region/zone-based bisimulations [29], which is broadly used in reasoning timed automata.

8 Conclusion

This work provides an alternative approach for verifying real-time systems, where temporal behaviors are reasoned at the source level, and the specification expressiveness goes beyond traditional Timed Automata. We define the novel effects logic *TimEffs*, to capture real-time behavioral patterns and temporal properties. We demonstrate how to build axiomatic semantics (or rather an effects system) for C^t via timed-trace processing functions. We use this semantic model to enable a Hoare-style forward verifier, which computes the program effects constructively. We present an effects inclusion checker – the TRS – to efficiently prove the annotated temporal properties. We prototype the verification system and show its feasibility. To the best of our knowledge, our work proposes the first algebraic TRS for solving inclusion relations between timed specifications.

Limitations And Future Work. Our TRS is incomplete, meaning there exist valid inclusions which will be disproved in our system. That is mainly because of insufficient unification in favour of achieving automation. We also foresee the possibilities of adding other logics into our existing trace-based temporal logic, such as separation logic for verifying heap-manipulating distributed programs.

9 Acknowledgements

The authors would like to thank anonymous reviewers for their comments. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOET32021-0001.

References

1. R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994. [Online]. Available: https://doi.org/10.1016/0304-3975(94)90010-8
2. X. Wang, J. Sun, T. Wang, and S. Qin, "Language inclusion checking of timed automata with non-zenoness," *IEEE Trans. Software Eng.*, vol. 43, no. 11, pp. 995–1008, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2017.2653778
3. K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *Int. J. Softw. Tools Technol. Transf.*, vol. 1, no. 1-2, pp. 134–152, 1997. [Online]. Available: https://doi.org/10.1007/s100090050010
4. S. Yovine, "KRONOS: A verification tool for real-time systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 1, no. 1-2, pp. 123–133, 1997. [Online]. Available: https://doi.org/10.1007/s100090050009
5. F. Wang, R. Wu, and G. Huang, "Verifying timed and linear hybrid rule-systems with RED," in *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'2005), Taipei, Taiwan, Republic of China, July 14-16, 2005*, W. C. Chu, N. J. Juzgado, and W. E. Wong, Eds., 2005, pp. 448–454.
6. K. Havelund, A. Skou, K. G. Larsen, and K. Lund, "Formal modeling and analysis of an audio/video protocol: an industrial case study using UPPAAL," in *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA*. IEEE Computer Society, 1997, pp. 2–13. [Online]. Available: https://doi.org/10.1109/REAL.1997.641264
7. K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using UPPAAL-TRON: an industrial case study," in *EMSOFT 2005, September 18-22, 2005, Jersey City, NJ, USA, 5th ACM International Conference On Embedded Software, Proceedings*, W. H. Wolf, Ed. ACM, 2005, pp. 299–306. [Online]. Available: https://doi.org/10.1145/1086228.1086283
8. J. S. Dong, P. Hao, S. Qin, J. Sun, and W. Yi, "Timed automata patterns," *IEEE Trans. Software Eng.*, vol. 34, no. 6, pp. 844–859, 2008. [Online]. Available: https://doi.org/10.1109/TSE.2008.52
9. A. David and M. D. Möller, "From huppaal to uppaal–a translation from hierarchical timed automata to flat timed automata," 2001.
10. L. Lamport, "Real-time model checking is really simple," in *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, ser. Lecture Notes in Computer Science, D. Borrione and W. J. Paul, Eds., vol. 3725. Springer, 2005, pp. 162–175. [Online]. Available: https://doi.org/10.1007/11560548_14
11. P. L. P. Ltd., https://www.programiz.com/javascript/setTimeout, 2022.
12. V. M. Antimirov and P. D. Mosses, "Rewriting extended regular expressions," *Theor. Comput. Sci.*, vol. 143, no. 1, pp. 51–72, 1995. [Online]. Available: https://doi.org/10.1016/0304-3975(95)80024-4
13. V. Antimirov, "Partial derivatives of regular expressions and finite automata constructions," in *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 1995, pp. 455–466.
14. J. Brotherston, "Cyclic proofs for first-order logic with inductive definitions," in *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2005, Koblenz, Germany, September 14-17, 2005,*

Proceedings, ser. Lecture Notes in Computer Science, B. Beckert, Ed., vol. 3702. Springer, 2005, pp. 78–92. [Online]. Available: https://doi.org/10.1007/11554554_8
15. L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24
16. Anonymous, https://www.comp.nus.edu.sg/ yahuis/TACAS2023.pdf, 2023.
17. J. Sun, Y. Liu, J. S. Dong, and J. Pang, "PAT: towards flexible verification under fairness," in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 709–714. [Online]. Available: https://doi.org/10.1007/978-3-642-02658-4_59
18. Y. Song, https://zenodo.org/record/7192718#.Y7rTmi8RpOQ, 2022.
19. Y. Liu, J. Sun, and J. S. Dong, "PAT 3: An extensible architecture for building multi-domain model checkers," in *IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011*, T. Dohi and B. Cukic, Eds. IEEE Computer Society, 2011, pp. 190–199. [Online]. Available: https://doi.org/10.1109/ISSRE.2011.19
20. Y. Song and W. Chin, "Automated temporal verification of integrated dependent effects," in *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, ser. Lecture Notes in Computer Science, S. Lin, Z. Hou, and B. P. Mahony, Eds., vol. 12531. Springer, 2020, pp. 73–90. [Online]. Available: https://doi.org/10.1007/978-3-030-63406-3_5
21. W. Yi, "CCS + time = an interleaving model for real time systems," in *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings*, ser. Lecture Notes in Computer Science, J. L. Albert, B. Monien, and M. Rodríguez-Artalejo, Eds., vol. 510. Springer, 1991, pp. 217–228. [Online]. Available: https://doi.org/10.1007/3-540-54233-7_136
22. W. Yi, P. Pettersson, and M. Daniels, "Automatic verification of real-time communicating systems by constraint-solving," in *Formal Description Techniques VII, Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques, Berne, Switzerland, 1994*, ser. IFIP Conference Proceedings, D. Hogrefe and S. Leue, Eds., vol. 6. Chapman & Hall, 1994, pp. 243–258.
23. P. C. Ölveczky and J. Meseguer, "Semantics and pragmatics of Real-Time Maude," *Higher-Order and Symbolic Computation*, vol. 20, no. 1-2, pp. 161–196, 2007.
24. B. Berthomieu and F. Vernadat, "Time petri nets analysis with TINA," in *Third International Conference on the Quantitative Evaluation of Systems (QEST 2006), 11-14 September 2006, Riverside, California, USA*. IEEE Computer Society, 2006, pp. 123–124. [Online]. Available: https://doi.org/10.1109/QEST.2006.56
25. C. Daws and S. Yovine, "Reducing the number of clock variables of timed automata," in *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), December 4-6, 1996, Washington, DC, USA*. IEEE Computer Society, 1996, pp. 73–81. [Online]. Available: https://doi.org/10.1109/REAL.1996.563702
26. S. Balaguer and T. Chatain, "Avoiding shared clocks in networks of timed automata," *Log. Methods Comput. Sci.*, vol. 9, no. 4, 2013. [Online]. Available: https://doi.org/10.2168/LMCS-9(4:13)2013

27. M. Muñiz, B. Westphal, and A. Podelski, "Detecting quasi-equal clocks in timed automata," in *Formal Modeling and Analysis of Timed Systems - 11th International Conference, FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proceedings*, ser. Lecture Notes in Computer Science, V. A. Braberman and L. Fribourg, Eds., vol. 8053. Springer, 2013, pp. 198–212. [Online]. Available: https://doi.org/10.1007/978-3-642-40229-6_14
28. S. Guha, C. Narayan, and S. Arun-Kumar, "Reducing clocks in timed automata while preserving bisimulation," in *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, ser. Lecture Notes in Computer Science, P. Baldan and D. Gorla, Eds., vol. 8704. Springer, 2014, pp. 527–543. [Online]. Available: https://doi.org/10.1007/978-3-662-44584-6_36
29. L. Luthmann, H. Göttmann, and M. Lochau, "Checking timed bisimulation with bounded zone-history graphs - technical report," *CoRR*, vol. abs/1910.08992, 2019. [Online]. Available: http://arxiv.org/abs/1910.08992