Periodica Polytechnica
Electrical Engineering
and Computer Science

*61(2), pp. 166-174, 2017*
*https://doi.org/10.3311/PPee.9766*
*Creative Commons Attribution*

RESEARCH ARTICLE

# Formal Verification of Real-Time Systems with Data Processing

Tamás Tóth[1*], István Majzik[1]

Received 19 July 2016; accepted after revision 16 October 2016

***Abstract***
*The behavior of practical safety critical systems often combines real-time behavior with structured data flow. To ensure correctness of such systems, both aspects have to be modeled and formally verified. Time related behavior can be efficiently modeled and analyzed in terms of timed automata. At the same time, program verification techniques like abstract interpretation and software model checking can efficiently handle data flow. In this paper, we describe a simple formalism that represents both aspects of such systems in a uniform and explicit way, thus enables the combination of formal analysis methods for real-time systems and software using standard techniques.*

***Keywords***
*model checking, abstract interpretation, abstraction refinement, timed automata, control flow automata*

## 1 Introduction

Ensuring the correctness of safety critical systems using formal verification is a challenging task as it requires formal modeling of the system in question, as well as the application of formal analysis techniques. Usually, the behavior of practical safety critical embedded systems exhibits both real-time aspects (e.g. switching to an error state after a certain amount of time has passed since the last event occurred) and data flow (e.g. branching on the value of a program variable or initializing a loop counter).

Time-related behavior can be conveniently modeled in terms of timed automata [1]. Model checkers for timed automata like Uppaal [2] and Kronos [3] can efficiently verify models using dedicated data structures that represent abstractions over real-valued clock variables.

On the other hand, state-of-the-art program verifiers [4] are designed to handle complex data flow, described in terms of a control flow automaton, and often use abstraction-refinement techniques [5] to handle variables of possibly infinite domains.

In this paper, to enable integration of verification techniques used in real-time verification and program verification, we define a formalism, Timed Control Flow Automata (TCFA), that is an extension of Control Flow Automata (CFA) used in program verification, with notions of Timed Automata (TA), the prominent formalism of real-time verification. Its main advantage is that it represents both data flow and timing uniformly, explicitly and in a way that is similar to the original formalisms, thus enables the application and combination of analyses that fit to the respective aspects. We define the syntax and semantics of the formalism, and describe how it relates to CFAs and TAs. Furthermore, we outline how analyses for the two formalisms can be combined using standard techniques to obtain efficient modular verifiers for TCFAs.

This paper is an extended version of [6]. Among a more detailed description of the concepts outlined there, it contains the description of an implementation of a verifier based on those concepts. Moreover, the paper contains the evaluation of said verifier on examples, including the verification of an industrial SCADA system.

[1] Budapest University of Technology and Economics,
Department of Measurement and Information Systems
Fault Tolerant Systems Research Group

[*] Corresponding author, e-mail: totht@mit.bme.hu

### 1.1 Related Work

Timed automata [1] is a widely used formalism for the modeling and verification of real-time systems. Dedicated model checkers for timed automata like Uppaal [2] and Kronos [3] usually use zone based abstractions [7-9] and efficient data structures like difference bound matrices [10] as their implementation to represent the infinite state space induced by real-valued clock variables. Abstraction-refinement techniques have also been developed for timed automata [11-13] to reduce size of the search space or the number of clocks variables considered.

A different line of work addresses the problem of real-time verification by using general infinite state model checking techniques based on satisfiability modulo theories (SMT) [14] solving, either directly or by tailoring them to timed automata. Proposed approaches include predicate abstraction with counterexample guided abstraction refinement [15, 16], symbolic backward search [17, 18], bounded model checking [19], k-induction [20], abstraction refinement using Craig interpolation [21], property directed reachability [20, 22-24] and Horn clause solving [22, 25].

While the formalization we propose allows for the utilization of both approaches for the formal verification of real-time systems, it naturally enables the application of a combined approach, where timed aspects are handled by dedicated algorithms for real-time, and data manipulation is addressed by the use of SMT-based infinite state model checking techniques.

### 1.2 Organization of the Paper

In Section 2, we define the notations used throughout the paper, and present the theoretical background of our work. In Section 3, we introduce the formalism of Timed Control Flow Automata, an extension of Control Flow Automata with notions of Timed Automata, and describe how verifiers for data flow and timing can be combined using standard combination methods to obtain efficient modular verifiers for the formalism. Section 4 describes our implementation of the formalism and corresponding analyses, and the evaluation of our implementation on examples. Finally, conclusions are given in Section 5.

## 2 Background and Notations

In this section, we define the notations used throughout the paper, and outline the theoretical background of our work.

### 2.1 General notions

**Types.** Let $Type$ denote a set of types and $Dom$ a mapping from types to their semantic domains. We assume $\{\textbf{bool}, \textbf{int}, \textbf{real}\} \subseteq Type$ such that $Dom(\textbf{bool}) = \mathbb{B}$, $Dom(\textbf{int}) = \mathbb{Z}$ and $Dom(\textbf{real}) = \mathbb{R}$.

**Variables.** Let $Var$ be a set of program variables. Variables have types, expressed as function $type : Var \to Type$. We abbreviate $Dom(type(v))$ by $Dom(v)$. The set of variables of type $\tau \in Type$ is denoted by $Var(\tau) = \{v \in Var \mid type(v) = \tau\}$.

**Expressions.** Let $Expr$ be a set of well-typed expressions over $Var$. An expressions can contain program variables $v \in Var$, logical connectives (**true**, **false**, $\neg$, $\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$), quantifiers ($\forall x : \tau . \varphi$, $\exists x : \tau . \varphi$) and logical variables, interpreted function symbols (e.g. 0. +, $\cdot$), interpreted predicate symbols (e.g. $\doteq$ , $<$, $\leq$), uninterpreted function and predicate symbols, and type constructors and accessors in case the type system supports complex data types. Given an expression $e \in Expr$ and a type $\tau \in Type$, we denote by $e : \tau$ iff $e$ has type $\tau$. Naturally, $v : \tau$ iff $type(v) = \tau$ for all variables $v \in Var$ and types $\tau \in Type$. The set of formulas is denoted by $Form = \{\varphi \in Expr \mid \varphi : \textbf{bool}\}$.

**States.** A concrete data state $\mathcal{S} \in State$ is a mapping from variables to values such that $\mathcal{S}(x) \in Dom(x)$ for all $x \in Var$. We also extend this notion to arbitrary expressions. For a state $\mathcal{S} \in State$ and formula $\varphi \in Expr$, we denote by $\mathcal{S} \vDash \varphi$ iff $\mathcal{S}(\varphi) = 1$.

### 2.2 Transition Systems

Transition systems are widely used for the formal modeling of the behavior of reactive systems.

**Syntax.** Formally, a labeled transition system is a tuple $TS = (S, Act, \to, I)$ where

- $S$ is a set of states,
- $Act$ is a set of actions,
- $\to \subseteq S \times Act \times S$ is a transition relation and
- $I \subseteq S$ is the set of initial states.

For convenience, we denote by $s \xrightarrow{\alpha} s'$ iff $(s, \alpha, s') \in \to$.

**Semantics.** A finite execution fragment of a transition system is an alternating sequence of states $s_i \in S$ and actions $\alpha_i \in Act$ of the form $s_0\alpha_1 s_1\alpha_2 \ldots \alpha_n s_n$ such that $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i < n$. An execution fragment is initial iff $s_0 \in I$. We say that a state $s \in S$ is reachable in a transition system iff there exists a finite initial execution fragment for the system such that $s = s_n$.

### 2.3 Timed Automata

Timed automata [1] is the most prevalent formalism for modeling real-time systems.

**Syntax.** A timed automaton is a tuple

$$TA = (Loc, Clock, \hookrightarrow, Inv, \ell_0)$$

where

- $Loc$ is a finite set of locations,

- $Clock$ is a finite set of clock variables such that $x : \mathbf{real}_{\geq 0}$ for all $x \in Clock$,
- $\hookrightarrow \subseteq Loc \times ClockConstr \times \mathcal{P}(Clock) \times Loc$ is a set of transitions where given a transition $(\ell, g, R, \ell') \in \hookrightarrow$, $g \in ClockConstr$ is a guard and $R \in Clock$ is a set containing clocks to be reset,
- $Inv : Loc \to ClockConstr$ is a function that maps to each location an invariant condition over clocks, and
- $\ell_0 \in Loc$ is the initial location.

Here, $ClockConstr \subseteq Form$ denotes the set of clock constraints, that is, formulas of the restricted form

$$\varphi ::= \mathbf{true} \mid x_i \sim c \mid x_i - x_j \sim c \mid \varphi_1 \wedge \varphi_2$$

where $x_i, x_j \in Clock$, $\sim \in \{<, \leq, >, \geq, =\}$ and $c$ is an integer literal.

**Semantics.** The operational semantics of a timed automaton can be defined as a labeled transition system $(S, Act, \to, I)$ where

- $S = Loc \times State$,
- $I = \{\ell_0\} \times \{\mathcal{S} \in State \mid \mathcal{S}(x) = 0 \text{ for all } x \in Clock \text{ and } \mathcal{S} \models Inv(\ell_0)\}$,
- $Act = \mathbb{R}_{\geq 0} \cup \{\tau\}$,
- and a transition $t \in \to$ of the transition relation $\to \subseteq S \times Act \times S$ is either a delay transition that increases all clocks with a value $\delta \geq 0$:

$$\frac{\ell \in Loc \quad \delta \geq 0 \quad \mathcal{S}' = Delay(\mathcal{S}, \delta) \quad \mathcal{S}' \models Inv(\ell)}{(\ell, \mathcal{S}) \xrightarrow{\delta} (\ell, \mathcal{S}')}$$

or a discrete transition:

$$\frac{\ell \xhookrightarrow{g,R} \ell' \qquad \mathcal{S} \models g \qquad \mathcal{S}' = Reset(\mathcal{S}, R) \qquad \mathcal{S}' \models Inv(\ell')}{(\ell, \mathcal{S}) \xrightarrow{\tau} (\ell', \mathcal{S}')}$$

Here, $Delay : State \times \mathbb{R}_{\geq 0} \to State$ assigns to a state $\mathcal{S} \in State$ and a real number $\delta \geq 0$ a state $Delay(\mathcal{S}, \delta)$ such that

$$Delay(\mathcal{S}, \delta)(v) = \begin{cases} \mathcal{S}(v) + \delta & \text{if } v \in Clock \\ \mathcal{S}(v) & \text{otherwise} \end{cases}$$

Moreover, function $Reset : State \times \mathcal{P}(Clock) \to State$ models the effect of resetting clocks in $R$ to 0 in state $\mathcal{S} \in State$:

$$Reset(\mathcal{S}, R)(v) = \begin{cases} 0 & \text{if } v \in R \\ \mathcal{S}(v) & \text{otherwise} \end{cases}$$

**Example.** Fig. 1 shows the timed automaton of a simple timed switch, that after switched on, remains in that state for at least one and at most two time units. On the figure, edges are labeled by clock resets and guards, and locations are labeled by invariants, in accordance with the syntax.

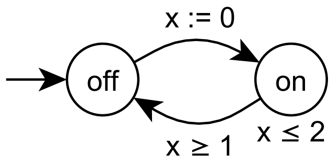

**Fig. 1** Timed switch as a TA

## 2.4 Control Flow Automata

In software model checking, programs are often modeled in terms of control flow automata.

**Syntax.** A control flow automaton is a tuple

$$CFA = (Loc, Var, \hookrightarrow, \ell_0)$$

where

- $Loc$ is a finite set of program locations,
- $Var$ is a set of program variables,
- $\hookrightarrow \subseteq Loc \times Stmt \times Loc$ is a set of control flow edges, and
- $\ell_0 \in Loc$ is the initial location.

Here, $Stmt$ denotes the set of statements. Although our formalization admits arbitrary structured statements, for the sake of simplicity we assume that statements are of the form

$$s ::= [\varphi] \mid v := e \mid \mathbf{havoc}\, v \mid s \,;\, s$$

where $v \in Var$, $e \in Expr$ and $\varphi \in Form$. Statement $[\varphi]$ is an **assume** statement, $v := e$ is an assignment of $e$ to $v$, $\mathbf{havoc}\, v$ is an assignment of an arbitrary value of a suitable type to $v$, and $s \,;\, s$ is a sequential statement.

**Semantics.** The operational semantics of a control flow automaton can be conveniently expressed in terms of a labeled transition system $(S, Act, \to, I)$ where

- $S = Loc \times State$,
- $I = \{\ell_0\} \times State$,
- $Act = Stmt$,
- and the transition relation $\to \subseteq S \times Act \times S$ is defined by the rule

$$\frac{\ell \overset{s}{\hookrightarrow} \ell' \qquad \mathcal{S}' \in Succ(\mathcal{S}, s)}{(\ell, \mathcal{S}) \xrightarrow{s} (\ell', \mathcal{S}')}$$

Here, $Succ : State \times Stmt \to \mathcal{P}(State)$ is the (not necessarily total) semantic function that assigns to a state $\mathcal{S} \in State$ and a statement $s \in Stmt$ a set of successor states $Succ(\mathcal{S}, s)$. It can be defined as the smallest relation satisfying the following rules:

$$\frac{s = [\varphi] \quad \mathcal{S} \models \varphi}{\mathcal{S} \in Succ(\mathcal{S}, s)}$$

$$\frac{s = (v := e) \quad \mathcal{S}' = \mathcal{S}[v \hookleftarrow \mathcal{S}(e)]}{\mathcal{S}' \in Succ(\mathcal{S}, s)}$$

$$\frac{s = \textbf{havoc}\,v \quad x \in Dom(v) \quad \mathcal{S}' = \mathcal{S}[v \hookleftarrow x]}{\mathcal{S}' \in Succ(\mathcal{S}, s)}$$

$$\frac{s = (s_1 ; s_2) \quad \mathcal{S}' \in Succ(\mathcal{S}, s_1) \quad \mathcal{S}'' \in Succ(\mathcal{S}', s_2)}{\mathcal{S}'' \in Succ(\mathcal{S}, s)}$$

**Example.** As a simple example, Fig. 2 shows the CFA model for the Euclidean algorithm for computing the greatest common divisor for two integers *a* and *b*.

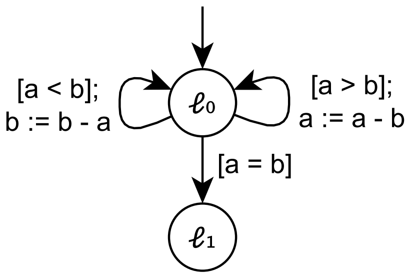

**Fig. 2** Euclidean algorithm as a CFA

## 2.5 Abstraction

To ensure termination or efficiency, program analyzers and modern model checkers check abstractions of systems [26], expressed in terms of abstract domains.

**Abstract domain.** An abstract domain is a triple $\mathbb{D} = (S, \mathcal{E}, \gamma)$ where

- $S$ is the set of concrete states, also called the concrete domain,
- $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup)$ is a semi-lattice over the set of abstract states $E$ with a top element $\top \in E$, a bottom element $\bot \in E$, a preorder $\sqsubseteq \subseteq E \times E$ and a join operator $\sqcup : E \times E \to E$, and
- $\gamma : E \to \mathcal{P}(S)$ is the concretization function that assigns to each abstract state the set of concrete states it represents. It satisfies the following properties:
  - $\gamma(\bot) = \varnothing$,
  - $\gamma(\top) = S$ and
  - $\gamma(e_1) \cup \gamma(e_2) \subseteq \gamma(e_1 \sqcup e_2)$ for all $e_1, e_2 \in E$

**Abstract semantics.** Given a transition system $(S, Act, \to, I)$ for the concrete semantics, the abstract semantics of the system w. r. t. abstract domain $(S, \mathcal{E}, \gamma)$ can be expressed as a transition system $(E, Act, \rightsquigarrow, E_0)$. Here, $\rightsquigarrow$ is the abstract transition relation (also called a transfer relation) and $E_0$ is the set of initial abstract states. For soundness of the abstraction, the following properties must hold:

- $I \subseteq \bigcup_{e_0 \in E_0} \gamma(e_0)$ and
- $\bigcup_{s \in \gamma(e)} \left(s' \in S \mid s \xrightarrow{\alpha} s'\right) \subseteq \bigcup_{e \overset{\alpha}{\rightsquigarrow} e'} \gamma(e')$ for all $e \in E$ and $\alpha \in Act$.

A verifier for the reachability of error states can then analyze the system by exploring the abstract state space and applying abstraction refinement [5] in case of a spurious counterexample that cannot be simulated according to the concrete semantics.

**Combining abstractions.** A modular way for constructing complex abstractions is by combining simpler abstract domains [27]. Known methods for combination of abstract domains include direct and reduced product [27], and logical product [28] constructions.

Although in general weaker than the other two methods, for simple cases where the component analyses refer to completely independent aspects of the system, even the direct product construction provides the strongest combination. Due to this fact and for ease of exposition, we restrict presentation to direct products, indicating that methods outlined later can be generalized directly to more involved combinators as well.

Given two abstract domains $\mathbb{D}_1 = (S, \mathcal{E}_1, \gamma_1)$ and $\mathbb{D}_2 = (S, \mathcal{E}_2, \gamma_2)$, their direct product is $\mathbb{D}_1 \times \mathbb{D}_2 = (S, \mathcal{E}, \gamma)$ where $\mathcal{E} = \mathcal{E}_1 \times \mathcal{E}_2$ and $\gamma((e_1, e_2)) = \gamma_1(e_1) \cap \gamma_2(e_2)$ for all $e_1 \in E_1$ and $e_2 \in E_2$. Moreover, given two transfer relations $\rightsquigarrow_1$ and $\rightsquigarrow_2$ for the respective domains over a common set of actions $Act$, their direct product is defined as the strongest relation $\rightsquigarrow = \rightsquigarrow_1 \times \rightsquigarrow_2$ such that the following property holds:

$$\frac{e_1 \overset{\alpha}{\rightsquigarrow}_1 e_1' \quad e_2 \overset{\alpha}{\rightsquigarrow}_2 e_2'}{(e_1, e_2) \overset{\alpha}{\rightsquigarrow} (e_1', e_2')}$$

Hence, the direct product abstraction tracks the „conjunction" of the information that can be obtained by running the component analyses independently.

**Examples.** Predicate abstraction [29] is a well known technique for obtaining finite abstractions of system with a potentially infinite state space (e.g. sequential programs). Here, given a set of predicates $\{p_1, p_2, \dots p_n\}$ (the precision), each one of the at most $2^n$ possible consistent valuations of the predicates corresponds to an abstract state. Predicate abstraction is well suited to counterexample-guided abstraction refinement [5], where the precision is augmented by additional predicates in case the current abstraction is not strong enough to exclude a spurious counterexample.

Zone abstraction (see e.g. [7]) is widely used for the verification of timed systems. In zone abstraction, abstract states are zones, that is, sets of interpretations of clock constraints (or in general, conjunctions of difference constraints). Over timed automata, zone abstraction is exact for both forward and backward search. To ensure termination however, extrapolation is used as the number of reachable zones of an automaton is not necessarily finite.

## 3 Timed Control Flow Automata

In this section, we introduce timed control flow automata, an extension of control flow automata with notions of timed automata. The formalism models both data flow and timing uniformly, explicitly and in a way that is similar to the original formalisms. As a consequence, the formalism enables the creation of efficient, modular verifiers by the combination of abstractions for the respective aspects using standard combination techniques.

### 3.1 Modeling formalism

**Syntax.** A timed control flow automaton is a tuple

$$TCFA = \left(Loc, Urg, Var, Clock, \hookrightarrow, Inv, \ell_0\right)$$

where

- $(Loc, Var, \hookrightarrow, \ell_0)$ is a CFA,
- $Urg \subseteq Loc$ is a set of urgent locations [2] that model locations where time shouldn't pass,
- $Clock \subseteq Var$ is the set of clock variables, and
- $Inv : Loc \to Form$ is a function that maps invariants to locations.

As can be seen from the definition, a TCFA can either be considered a CFA extended with clock variables, urgent locations and location invariants, or alternatively, as a generalized TA with urgent locations where guards and clock resets are represented as statements.

**Semantics.** The semantics of a timed control flow automaton is a transition system $(S, Act, \to, I)$ where

- $S = Loc \times State$,
- $I = \{\ell_0\} \times \{\mathcal{S}_0 \in State \mid \mathcal{S}_0 \models Inv(\ell_0)\}$, that is, all variables (including clocks) have an arbitrary initial value of the corresponding type that satisfies the invariant of the initial location,
- $Act = Stmt \cup \{\textbf{delay}\}$,
- and a transition $t \in \to$ of the transition relation $\to \subseteq S \times Act \times S$ is either a delay transition that increases all clocks with a value $\delta \geq 0$:

$$\frac{\ell \notin Urg \quad \delta \geq 0 \quad \mathcal{S}' = Delay(\mathcal{S}, \delta) \quad \mathcal{S}' \models Inv(\ell)}{(\ell, \mathcal{S}) \xrightarrow{\textbf{delay}} (\ell, \mathcal{S}')}$$

or a discrete transition that models the execution of a statement $s \in Stmt$:

$$\frac{\ell \overset{s}{\hookrightarrow} \ell' \qquad \mathcal{S}' \in Succ(\mathcal{S}, s) \qquad \mathcal{S}' \models Inv(\ell')}{(\ell, \mathcal{S}) \xrightarrow{s} (\ell', \mathcal{S}')}$$

As it can be seen, the two types of transitions are analogous to the cases described for timed automata. We note however that for the analysis of reachability properties, a combined step semantics [20] can be defined as an alternative where a transition is a combination of a single delay and a discrete transition. In this case, finite initial execution fragments that witness the reachability of a given state are potentially half as long. This enhances performance of reachability checking compared to the original formulation, especially for verifiers based on unrolling of the transition relation or counterexample guided abstraction refinement.

**Example.** As an example, Fig. 3 depicts Fischer's protocol [30] as a TCFA. Here, $a$, $b$ and $i$ are constant values of type **int**.

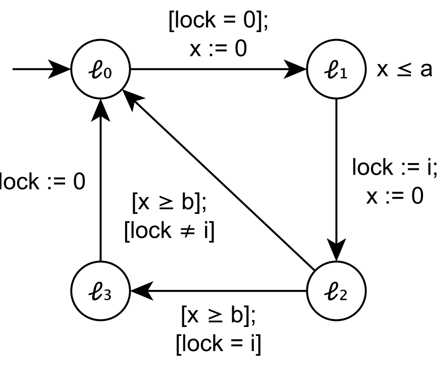

**Fig. 3** Fischer's protocol as a TCFA

**Interleaving of TCFAs.** To enable modeling of concurrent systems, we define the interleaving of two TCFAs. Given

$$TCFA_i = (Loc_i, Urg_i, Var_i, Clock_i, \hookrightarrow_i, Inv_i, \ell_{0,i})$$

for $i \in \{1, 2\}$, their interleaving is defined as

$$TCFA_1 \mid\mid\mid TCFA_2 = (Loc, Urg, Var, Clock, \hookrightarrow, Inv, \ell_0)$$

where

- $Loc = Loc_1 \times Loc_2$,
- $Var = Var_1 \cup Var_2$,
- $\ell_0 = (\ell_{0,1}, \ell_{0,2})$
- $Urg = Urg_1 \times Loc_2 \cup Loc_1 \times Urg_2$
- $Clock = Clock_1 \cup Clock_2$,
- $Inv((\ell_1, \ell_2)) = Inv_1(\ell_1) \wedge Inv_2(\ell_2)$, and
- $\hookrightarrow$ is defined by the following rules:

$$\frac{\ell_1 \overset{s}{\hookrightarrow}_1 \ell_1'}{(\ell_1, \ell_2) \overset{s}{\hookrightarrow} (\ell_1', \ell_2)} \quad \text{and} \quad \frac{\ell_2 \overset{s}{\hookrightarrow}_2 \ell_2'}{(\ell_1, \ell_2) \overset{s}{\hookrightarrow} (\ell_1, \ell_2')}$$

Here, $Var_1 \cap Var_2$ are shared variables, and $Clock_1 \cap Clock_2$ are shared clocks.

### 3.2 Connection to TAs and CFAs

The TCFA formulation admits a simple and uniform description of both CFAs and TAs.

**TA as TCFA.** For any $TA = (Loc, Clock, \hookrightarrow_0, Inv, \ell_0)$ there exists a $TCFA = (Loc, \emptyset, Clock, Clock, \hookrightarrow, Inv, \ell_0)$ that - aside from the inital values of clocks and the action labels on transitions - is semantically equivalent to the original TA. Here, $\hookrightarrow$ is defined by the following rule:

$$\frac{\ell \overset{g,R}{\hookrightarrow}_0 \ell' \qquad R = \{x_1, \ldots, x_n\}}{\ell \xhookrightarrow{[g]\ ;\ x_1:=0\ ;\ \ldots\ ;\ x_n:=0} \ell'}$$

**CFA as TCFA.** For any $CFA = (Loc, Var, \hookrightarrow, \ell_0)$, there exists a semantically equivalent $TCFA = (Loc, Loc, Var, \emptyset, \hookrightarrow, Inv, \ell_0)$ where $Inv(\ell) = \mathbf{true}$ for all $\ell \in Loc$.

**TCFA as CFA.** For any $TCFA = (Loc, Urg, Var, Clock, \hookrightarrow_0, Inv, \ell_0)$ such that $Inv(\ell_0) = \mathbf{true}$, there exists a semantically equivalent $CFA = (Loc, Var, \hookrightarrow, \ell_0)$ (aside from action labels on transitions), where $\hookrightarrow$ is defined by the following rules:

- Simulating discrete transitions:

$$\frac{\ell \overset{s}{\hookrightarrow}_0 \ell' \qquad \varphi = Inv(\ell')}{\ell \xhookrightarrow{s\ ;\ [\varphi]} \ell'}$$

- Simulating delay transitions:

$$\frac{\ell \notin Urg \qquad \varphi = Inv(\ell)}{\ell \xhookrightarrow{\mathbf{delay}\ ;\ [\varphi]} \ell}$$

Here, **delay** stands for the statement

$$\mathbf{havoc}\,\delta\,; x_1 := x_1 + \delta; \ldots; x_n := x_n + \delta$$

where $\delta : \mathbf{real}_{\geq 0}$ is a distinguished delay variable and $Clock = \{x_1, \ldots, x_n\}$.

By applying this simple translation, the application of program verifiers become directly available for the analysis of timed systems. However, as in this case time related behavior becomes implicitly encoded during the translation, such an approach is expected to be not necessarily complete and in general less efficient than the use of a dedicated algorithm (see related work in Section 1) that addresses real-time aspects directly, especially for models with many clock variables.

## 3.3 Abstraction for TCFAs

The main advantage of the above formulation is that it admits verifiers to be built in a modular way. More precisely, given abstractions $\mathbb{D}_{data}$ for data variables and $\mathbb{D}_{time}$ for clock variables with respective transfer relations $\rightsquigarrow_{data}$ and $\rightsquigarrow_{time}$, by combining the two abstractions (e.g. by taking their direct product in the simplest case) an analysis can be built that is a full-fledged verifier for the complete system (we assume that the control location is explicitly tracked). Here, $\mathbb{D}_{data}$ essentially realizes an analysis for software that operates on CFAs, and $\mathbb{D}_{time}$ realizes an analysis for TAs, expressed over the uniform representation of TCFAs.

**Example.** As a simple example, Fig. 4 illustrates the abstract state space of Fischer's protocol where $\mathbb{D}_{data}$ is predicate abstraction over a singleton set of predicates $\{lock = i\}$ for some $i \neq 0$, $\mathbb{D}_{time}$ is zone abstraction over a singleton set of clocks $\{x\}$, and the combined abstraction is $\mathbb{D}_{data} \times \mathbb{D}_{time}$. On the figure, abstract states are depicted as rounded rectangles, where the current location is shown in the left compartment, and the information tracked by the zone and predicate analyses are depicted in the middle and right compartments, respectively. With both timing and data handled with an appropriate abstraction, a compact over-approximation of the concrete state space is obtained that enables sound and efficient reachability analysis of the system.

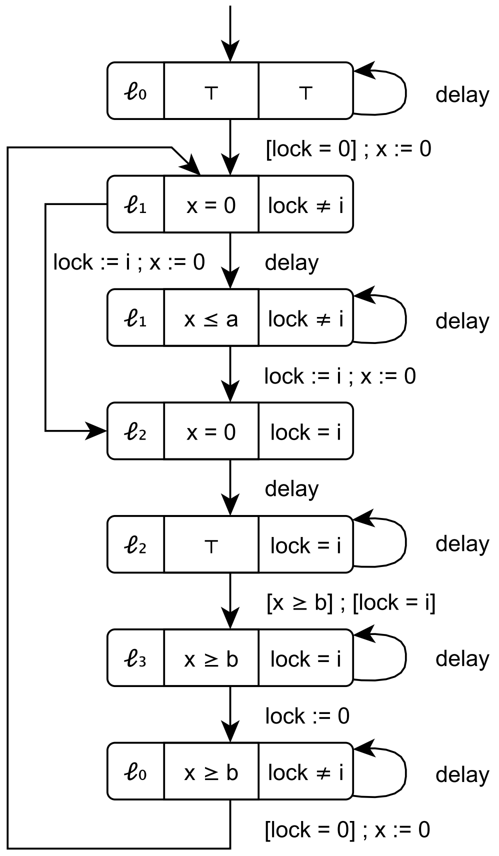

**Fig. 4** Abstract Reachability Graph for Fischer's protocol

As hinted earlier, in simple cases like this, the direct product provides the strongest combination, as statements and atomic formulas occurring in invariants are pure in the sense that they do not mix clock and data variables. However, this restriction doesn't apply to the formalism itself, and in principle analyses can be defined for cases where data and clock variables are in some way interdependent (e.g. a clock is reset to the value stored in a data variable, the current value of a clock is stored in a data variable, clocks are allowed to be bound by data values in assumptions and invariants). Such complex cases are however out of the scope of this paper.

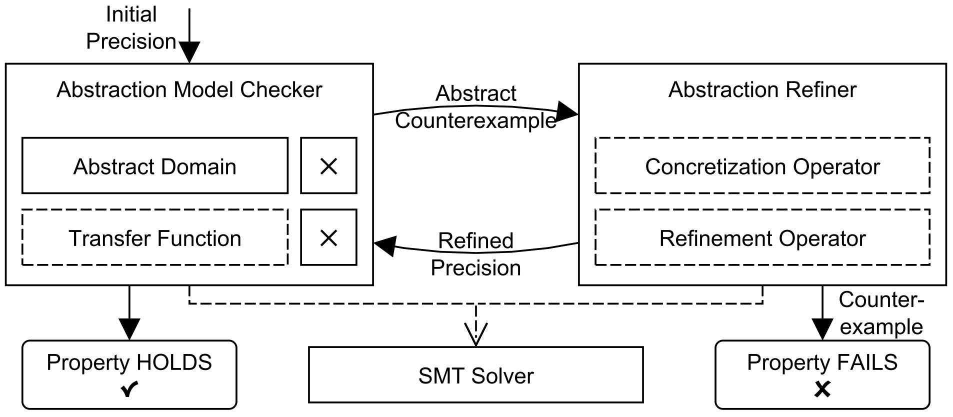

**Fig. 5** Architecture of the Analysis Framework

## 4 Implementation

As a proof of concept, we implemented several analyses in our verification framework to enable analysis of TCFAs. We also evaluated our implementation on two models.

### 4.1 A Framework for Abstract Model Checking

The basis for our implementation is a framework that supports formal verification based on abstraction and abstraction refinement. The simplified architecture of the framework is depicted on Fig. 5.

Our framework provides the following components out-of-the box:

- *Abstraction Model Checker*. An abstraction model checker is implemented that builds the abstract reachability tree (ART) of a system w.r.t. a given abstraction and precision.
- *Abstraction Refiner*. Optionally, the model checker can be augmented with an abstraction refiner component. The model checker and the refiner then can be organized to an abstraction refinement loop. In this case, an abstract counterexample found by the model checker can be passed to the refiner to either concretize it, or exclude it from later search by refining the abstraction. After successful refinement, the model checker can continue with the expansion of the reachability tree.
- *Abstract Domains*. Supported domains include predicate domain, zone domain and explicit value domain. Moreover, since the framework is not specialized to any particular formalism, a domain is defined to track the current control location of an automaton (as in [31]) to enable flow-sensitive analysis of such formalisms.
- *Combination of Domains*. The generic direct product construction is implemented both for abstract domains and transfer relations. Combined domains can optionally be augmented with a reduction operator, thus providing a stronger abstraction.
- *SMT Solvers*. The framework defines a common interface for SMT solvers, with support for generation of interpolants [32]. The solvers can be used for implementing transfer relations or abstraction refiner components.

### 4.2 Analysis Support for TCFAs

We implemented the modeling formalism of TCFAs as described in the paper, including the interleaving operator to support the modeling of networks of TCFAs. Moreover, to enable the formal verification of TCFAs, we extended the framework with the following components:

- *Transfer Relations*. Transfer relations for TCFAs interpret delays and statements over a given abstract domain. We implemented transfer relations for all the abstract domains mentioned above.

The framework and the extensions implemented for TCFAs thus enable abstract model checking for networks of TCFAs, with several different combinations that differ in the handling of data and timing (e.g. explicit tracking of values, the application of predicates, or both for data variables). As for a given system some of the many configurations might be more fitting than others, more efficient verification can be achieved, either by using selection heuristics based on the input model or by running diverse configurations in parallel as part of a portfolio.

### 4.3 Evaluation

We evaluated our implementation of abstractions for networks of TCFAs by generating abstract reachability trees over different abstract domains for two models: Fischer’s mutual exclusion protocol, and a protocol from an industrial railway supervisory control and data acquisition (SCADA) system.

**Example.** For Fischer’s protocol, we compared two combinations of abstractions: predicate abstraction and explicit value analysis, combined with zone abstraction in both cases. Using predicate analysis over the single predicate $lock > 0$, our implementation generates the ART in 4 seconds for 3 processes, and reaches the timeout (set to 5 minutes) for 4 processes due to a large state space. However using explicit value analysis, the generation of the ART takes 1 second for 3 processes, 10 seconds for 4 processes, and results in timeout for 5 processes. For comparison, our implementation is less efficient yet than the sophisticated and optimized implementation of Uppaal forward search (without state space reduction and extrapolation), but the limits are similar: Uppaal is able to generate the state space of 5 processes in 4 seconds and reaches timeout for 6 processes.

**Example.** Our other example is a protocol from an industrial SCADA system. Its primary function is dependable connection handling between the field and control units of the railway control system with given timing conditions. For a detailed description of the protocol, see [33].

We modeled the protocol as a network of TCFAs, under a fault model that permits the loss of a single message. The formalism was well applicable to the task, as in the protocol both real-time (e.g. sending synchronization messages periodically) and data related (e.g. storing messages) behavior was present. Fig. 6 depicts the TCFA model of a component in the protocol.

Due to the presence of clock variables and the small number of possible values for data variables in the TCFA model of the protocol, we applied the combination of zone abstraction and explicit value analysis for its verification. The abstract reachability tree generation for the model executed in one second.

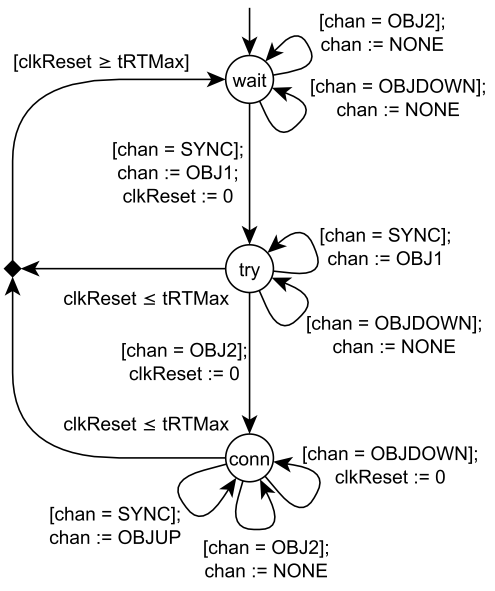

**Fig. 6** TCFA Model of SCAN Connection Handling (Field Side)

## 5 Conclusions and Future Work

In this paper, we described the *formalism* of timed control flow automata that is an extension of control flow automata with notions of timed automata. We compared it to the original formalisms, and demonstrated how *modular verifiers* can be built for the formalism by combining abstractions used for CFAs and TAs using standard combination techniques.

The main advantage of the method is that it enables the use of many different abstraction techniques for both data and timing, some of which may be more efficient than the others for the given model. The price of such an approach is the overhead caused by the indirection introduced for the generic handling of abstractions.

In the future, we plan to implement further analyses for the formalism and augment them with abstraction refinement techniques known from the area of software model checking. Moreover, to enable modeling of industrial systems, we plan to investigate analyses that enable the verification of parametric systems and cases where clock and data variables are interdependent.

## Acknowledgment

This work was partially supported by Gedeon Richter's Talentum Foundation (Gyömrői út 19-21, 1103 Budapest, Hungary).

## References

[1] Alur, R., Dill, D. L. "A theory of timed automata." *Theoretical Computer Science*. 126(2), pp. 183-235. 1994. https://doi.org/10.1016/0304-3975(94)90010-8

[2] Behrmann, G., David, A., Larsen, K. G., Håkansson, J., Petterson, P., Wang, Y., Hendriks, M. "UPPAAL 4.0." In: Third International Conference on the Quantitative Evaluation of Systems, Riverside, CA, USA, Sept. 11-14, 2006, pp. 125-126. https://doi.org/10.1109/QEST.2006.59

[3] Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S. "Kronos: A model-checking tool for real-time systems." In: Ravn, A.P., Rischel, H. (eds.) Formal Techniques in Real-Time and Fault-Tolerant Systems. FTRTFT 1998. Lecture Notes in Computer Science, Vol. 1486. Springer, Berlin, Heidelberg. 1998 https://doi.org/10.1007/BFb0055357

[4] Beyer, D. "Software Verification and Verifiable Witnesses." In: Taier C., Tinelli C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2015. Lecture Notes in Computer Science, Vol. 9035. Springer, Berlin, Heidelberg, 2015. https://doi.org/10.1007/978-3-662-46681-0_31

[5] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H. "Counter example guided abstraction refinement for symbolic model checking." *Journal of the ACM*. 50(5), pp. 752-794, 2003. https://doi.org/10.1145/876638.876643

[6] Tóth, T., Majzik, I. "Formal modeling of real-time systems with data processing." In: Proceedings of the 23rd PhD Mini-Symposium, pp. 46-49, 2016.

[7] Daws, C., Tripakis, S. "Model checking of real-time reachability properties using abstractions." In: Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 1998. Lecture Notes in Computer Science, Vol. 1384, Springer, Berlin, Heidelberg, 1998. https://doi.org/10.1007/BFb0054180

[8] Behrmann, G., Bouyer, P., Larsen, K. G., Pelánek, R. "Lower and Upper Bounds in Zone Based Abstractions of Timed Automata." In: Jensen, K., Podelski A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2004. Lecture Notes in Computer Science, Vol. 2988, Springer, Berlin, Heidelberg, 2004. https://doi.org/10.1007/978-3-540-24730-2_25

[9] Herbreteau, F., Srivathsan, B., Walukiewicz, I. "Better abstractions for timed automata." In: Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science, LICS, pp. 375-384, IEEE, 2012. https://doi.org/10.1109/LICS.2012.48

[10] Dill, D. L. "Timing assumptions and verification of finite-state concurrent systems." In Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science, Vol. 407, pp. 197-212, Springer, Berlin, Heidelberg, 1990.
https://doi.org/10.1007/3-540-52148-8_17

[11] Dierks, H., Kupferschmid, S., Larsen, K. G. "Automatic Abstraction Refinement for Timed Automata." In: Raskin, J. F., Thiagarajan, P. S. (eds.) Formal Modeling and Analysis of Timed Systems, Lecture Notes in Computer Science. FORMATS 2007, Vol. 4763, pp. 114-129, Springer, Berlin, Heidelberg, 2007.
https://doi.org/10.1007/978-3-540-75454-1_10

[12] Okano, K., Bordbar, B., Nagaoka, T. "Clock Number Reduction Abstraction on CEGAR Loop Approach to Timed Automaton." In: 2011 Second International Conference on Networking and Computing, Osaka, 2011, pp. 235-241. https://doi.org/10.1109/ICNC.2011.42

[13] Herbreteau, F., Srivathsan, B., Walukiewicz, I. "Lazy abstractions for timed automata." In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification. CAV 2013. Lecture Notes in Computer Science, Vol. 8044. Springer, Berlin, Heidelberg, 2013.
https://doi.org/10.1007/978-3-642-39799-8_71

[14] Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C. "Satisfiability modulo theories." In: *Handbook of Satisfiability*. Vol. 185 of Frontiers in Artificial Intelligence and Applications, ch. 26, pp. 825-885, IOS Press, 2009.
https://doi.org/10.3233/978-1-58603-929-5-825

[15] Möller, M. O., Rueß, H., Sorea, M. "Predicate abstraction for dense realtime systems." *Electronic Notes in Theoretical Computer Science*. 65(6), pp. 218-237. 2002. https://doi.org/10.1016/S1571-0661(04)80478-X

[16] Sorea, M. "Lazy Approximation for Dense Real-Time Systems." In: Lakhnech, Y., Yovine, S. (eds.) Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems. Lecture Notes in Computer Science, Vol. 3253, Springer, Berlin, Heidelberg, 2004.
https://doi.org/10.1007/978-3-540-30206-3_25

[17] Carioni, A., Ghilardi, S., Ranise, S. "MCMT in the Land of Parameterized Timed Automata." In: Proceedings of VERIFY, pp. 1-16, 2010.

[18] Morbé, G., Pigorsch, F., Scholl, C. "Fully Symbolic Model Checking for Timed Automata." In: Gopalakrishnan G., Qadeer S. (eds.) Computer Aided Verification. CAV 2011. Lecture Notes in Computer Science, Vol. 6806. Springer, Berlin, Heidelberg, 2011.
https://doi.org/10.1007/978-3-642-22110-1_50

[19] Kindermann, R., Junttila, T., Niemela, I. "Beyond Lassos: Complete SMT-Based Bounded Model Checking for Timed Automata." In: Giese, H., Rosu, G. (eds) Formal Techniques for Distributed Systems. Lecture Notes in Computer Science, Vol. 7273. Springer, Berlin, Heidelberg, 2012. https://doi.org/10.1007/978-3-642-30793-5_6

[20] Kindermann, R., Junttila, T., Niemela, I. "SMT-based Induction Methods for Timed Systems." In: Jurdziński, M., Ničković, D. (eds.) Formal Modeling and Analysis of Timed Systems. FORMATS 2012. Lecture Notes in Computer Science, vol 7595. Springer, Berlin, Heidelberg, 2012.
https://doi.org/10.1007/978-3-642-33365-1_13

[21] Kemper, S. "SAT-based Abstraction Refinement for Realtime Systems." *Electronic Notes in Theoretical Computer Science*. 182, pp. 107-122, 2007. https://doi.org/10.1016/j.entcs.2006.09.034

[22] Hoder, K., Bjorner, N. "Generalized Property Directed Reachability." In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing – SAT 2012. SAT 2012. Lecture Notes in Computer Science, Vol. 7317, Springer, Berlin, Heidelberg, 2012.
https://doi.org/10.1007/978-3-642-31612-8_13

[23] Isenberg, T., Wehrheim, H. "Timed Automata Verification via IC3 with Zones." In: Merz, S., Pang, J. (eds) Formal Methods and Software Engineering. ICFEM 2014. Lecture Notes in Computer Science, Vol. 8829, Springer, Cham, 2014. https://doi.org/10.1007/978-3-319-11737-9_14

[24] Isenberg, T. "Incremental Inductive Verification of Parameterized Timed Systems." In: 2015 15th International Conference on Application of Concurrency to System Design, IEEE, Brussels, 2015, pp. 1-9.
https://doi.org/10.1109/ACSD.2015.13

[25] Hojjat, H., Rümmer, P., Subotic, P., Yi, W. "Horn Clauses for Communicating Timed Systems." *Electronic Proceedings in Theoretical Computer Science*. 169, pp. 39–52. 2014.
https://doi.org/10.4204/EPTCS.169.6

[26] Cousot, P., Cousot, R. "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints." In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77, pp. 238-252, 1977. https://doi.org/10.1145/512950.512973

[27] Cousot, P., Cousot, R. "Systematic design of program analysis frameworks." In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '79, pp. 269-282, 1979. https://doi.org/10.1145/567752.567778

[28] Gulwani, S., Tiwari, A. "Combining abstract interpreters." In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '06, pp. 376-386, 2006.
https://doi.org/10.1145/1133255.1134026

[29] Graf, S., Saidi, H. "Construction of abstract state graphs with PVS." In: Grumberg, O. (ed.) Computer Aided Verification. CAV 1997. Lecture Notes in Computer Science, Vol. 1254, Springer, Berlin, Heidelberg, 1997. https://doi.org/10.1007/3-540-63166-6_10

[30] Lamport, L. "A fast mutual exclusion algorithm." *ACM Transactions on Computer Systems*. 5(1), pp. 1-11. 1987.
https://doi.org/10.1145/7351.7352

[31] Beyer, D., Henzinger, T. A., Théoduloz, G. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis." In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification. CAV 2007. Lecture Notes in Computer Science, Vol. 4590, Springer, Berlin, Heidelberg, 2007.
https://doi.org/10.1007/978-3-540-73368-3_51

[32] Cimatti, A., Griggio, A., Sebastiani, R. "Efficient interpolant generation in satisfiability modulo theories." In: Ramakrishnan, C. R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2008. Lecture Notes in Computer Science, Vol. 4963, Springer, Berlin, Heidelberg, 2008.
https://doi.org/10.1007/978-3-540-78800-3_30

[33] Tóth, T., Vörös, A., Majzik, I. "Verification of a real-time safety-critical protocol using a modelling language with formal data and behaviour semantics." In: Bondavalli, A., Ceccarelli, A., Ortmeier, F. (eds.) Computer Safety, Reliability, and Security. SAFECOMP 2014. Lecture Notes in Computer Science, vol 8696. Springer, Cham, 2014.
https://doi.org/10.1007/978-3-319-10557-4_24