

Formal Modeling of Real-Time Systems with Data Processing

Tamás Tóth* and István Majzik†

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

Email: *totht@mit.bme.hu, †majzik@mit.bme.hu

Abstract—The behavior of practical safety critical systems usually 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 is able to model both aspects of such systems and enables the combination of formal verification techniques for real-time systems and software. We also outline a straightforward method for building efficient verifiers for the formalism based on the combination of analyses for the respective aspects.

I. 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 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] can efficiently verify models using dedicated data structures that represent abstractions over real-valued clock variables [3]. Usually, data variables are handled by encoding the data flow into the control flow [2], which only admits variables of finite domains, or alternatively by using a logical encoding [4]–[11] and then performing model checking by calling to decision procedures. In the latter case, the information about time-related behavior becomes implicit and efficiency depends mostly on the underlying solver.

On the other hand, state-of-the-art program verifiers [12] are designed to handle complex data flow, described in terms of a control flow automaton, and often use abstraction-refinement techniques [13] to handle variables of possibly infinite domains. However, they are not directly capable of verifying timed systems.

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

†This work was partially supported by the ARTEMIS JU and the Hungarian National Research, Development and Innovation Fund in the frame of the R5-COP project.

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 explicitly and in a way that is similar to the original formalisms, thus enables the application 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 a simple method for combining analyses for the two formalisms to build efficient verifiers for TCFA.

II. BACKGROUND AND NOTATIONS

In this section, we describe the notations used in the paper.

A. Types

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

B. Variables

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

C. Expressions

Let *Expr* be a set of well-typed expressions over *Var*. An expressions can contain program variables $v \in \text{Var}$, 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 $e \in \text{Expr}$ and a type $\tau \in \text{Type}$, we denote by $e : \tau$ iff e has type τ . Naturally, $v : \tau$ iff $\text{type}(v) = \tau$ for all variables $v \in \text{Var}$ and types $\tau \in \text{Type}$. The set of formulas is denoted by $\text{Form} = \{\varphi \in \text{Expr} \mid \varphi : \mathbf{bool}\}$.

D. 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 $\varphi \in Form$, we denote by $\mathcal{S} \models \varphi$ iff $\mathcal{S}(\varphi) = 1$.

E. Statements

Let $Stmt$ denote 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 , **havoc** v is an assignment of an arbitrary value of a suitable type to v , and $s ; s$ is a sequential statement.

The semantics of statements can be expressed by the (not necessarily total) semantic function $Succ : State \times Stmt \rightarrow \mathcal{P}(State)$ 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

- $\{\mathcal{S}\}$ if $s = [\varphi]$ and $\mathcal{S} \models \varphi$
- \emptyset if $s = [\varphi]$ and $\mathcal{S} \not\models \varphi$
- $\{\mathcal{S}' \in State \mid \mathcal{S}' = \mathcal{S}[v \leftarrow \mathcal{S}(e)]\}$ if $s = v := e$
- $\{\mathcal{S}' \in State \mid \mathcal{S}' = \mathcal{S}[v \leftarrow x] \text{ for some } x \in Dom(v)\}$ if $s = \mathbf{havoc} \ v$
- $\{\mathcal{S}'' \in State \mid \mathcal{S}' \in Succ(\mathcal{S}, s_1) \text{ and } \mathcal{S}'' \in Succ(\mathcal{S}', s_2) \text{ for some } \mathcal{S}' \in State\}$ if $s = s_1 ; s_2$

F. Timed Automata

Timed automata [1] is a widely used formalism for modeling real-time systems. A TA is a tuple $(Loc, Clock, \hookrightarrow, Inv, \ell_0)$ where

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

Here, $ClockConstr$ denotes the set of clock constraints, that is, formulas of the form $x_i \sim 0$ and $x_i - x_j \sim c$ where $x_i, x_j \in Clock$, $\sim \in \{<, \leq, =\}$ and c is an integer literal.

The operational semantics of a TA can be defined as a labeled transition system (S, Act, \rightarrow, I) where

- $S = Loc \times State$ is the set of states,
- $I = \{\ell_0\} \times \{\mathcal{S} \in State \mid \mathcal{S}(x) = 0 \text{ for all } x \in Clock \text{ and } \mathcal{S} \models Inv(\ell_0)\}$ is the set of initial states,
- $Act = \mathbb{R}_{\geq 0} \cup \{\alpha\}$, where α denotes discrete transitions,
- and a transition $t \in \rightarrow$ of the transition relation $\rightarrow \subseteq S \times Act \times S$ 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:

$$\frac{\ell \xrightarrow{g, R} \ell' \quad \mathcal{S} \models g \quad \mathcal{S}' = Reset(\mathcal{S}, R) \quad \mathcal{S}' \models Inv(\ell')}{(\ell, \mathcal{S}) \xrightarrow{\alpha} (\ell', \mathcal{S}')}$$

Here, $Delay : State \times \mathbb{R}_{\geq 0} \rightarrow State$ 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, $Reset(\mathcal{S}, R)$ models the effect of resetting clocks in R to 0 in state $\mathcal{S} \in State$:

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

G. Control Flow Automata

In program analysis, programs are modeled in terms of control flow automata. Syntactically, a CFA is a tuple $(Loc, Var, \hookrightarrow, \ell_0)$ 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.

The operational semantics of a CFA then can be conveniently expressed in terms of a labeled transition system (S, Act, \rightarrow, I) where

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

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

H. Abstract Semantics

To ensure efficiency or termination, modern model checkers and program analyzers check abstractions of systems, expressed in terms of abstract domains. An abstract domain is a triple (S, \mathcal{E}, γ) where

- S is the set of concrete states,
- $\mathcal{E} = (E, \top, \perp, \sqsubseteq, \sqcup)$ is a semi-lattice over the set of abstract states E with a top element $\top \in E$, a bottom element $\perp \in E$, a preorder $\sqsubseteq \subseteq E \times E$ and a join operator $\sqcup : E \times E \rightarrow E$, and
- $\gamma : E \rightarrow \mathcal{P}(S)$ is the concretization function that assigns to each abstract state the set of concrete states it represents.

Given a transition system (S, Act, \rightarrow, I) for the concrete semantics, the abstract semantics w.r.t. \mathcal{E} and γ can be expressed as a transition system $(E, Act, \rightsquigarrow, \gamma(I))$. For soundness of the analysis, the following properties must hold:

- $\gamma(\top) = S$ and $\gamma(\perp) = \emptyset$,
- $\gamma(e_1) \cup \gamma(e_2) \subseteq \gamma(e_1 \sqcup e_2)$ for all $e_1, e_2 \in E$, and
- $\bigcup_{s \in \gamma(e)} \{s' \in S \mid s \xrightarrow{\alpha} s'\} \subseteq \bigcup_{e' \in \alpha_e} \gamma(e')$ for all $e \in E$ and $\alpha \in Act$.

The abstract transition relation $\rightsquigarrow \subseteq E \times Act \times E$ is also called a transfer relation.

A verifier can then analyze the system by exploring the abstract state space and applying abstraction refinement [13] in case of a spurious error path that cannot be simulated according to the concrete semantics.

III. TIMED CONTROL FLOW AUTOMATA

To extend CFAs with timed behavior, we assume $\mathbf{clock} \in Type$ for a distinguished type \mathbf{clock} such that $Dom(\mathbf{clock}) = \mathbb{R}_{\geq 0}$. This enables modeling of a clock variable as a regular program variable of type \mathbf{clock} . In his context, $Clock = Var(\mathbf{clock})$.

A. Syntax

A TCFA is a tuple $(Loc, Urg, Var, \hookrightarrow, Inv, \ell_0)$ where

- $(Loc, Var, \hookrightarrow, \ell_0)$ is a CFA (with $\mathbf{clock} \in Type$),
- $Urg \subseteq Loc$ is a set of urgent locations that model locations where time shouldn't pass, and
- $Inv : Loc \rightarrow Form$ is a function that maps invariants to locations.

Moreover, we assume that all atomic formulas that contain clock variables are clock constraints.¹

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 where guards and clock resets are represented as statements. As a consequence, optimizations from both areas (e.g. large block encoding [14]) might be applicable.

As an example, Figure 1 depicts Fischer's protocol as a TCFA. Here, a, b and i are constant values of type \mathbf{int} .

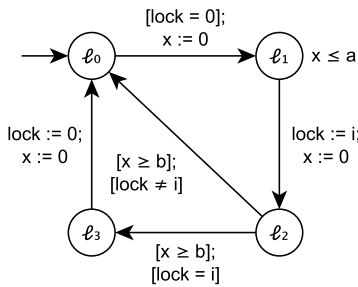


Fig. 1. Fischer's protocol as a TCFA

¹Note that the formalism is sensible even if this assumption is not made. However, in order to apply the theory of timed automata for verification, it has to be assumed.

B. Concrete Semantics

The semantics of a TCFA is (S, Act, \rightarrow, I) where

- $S = \{\ell_0\} \times State$,
- $I = \{(\ell_0, \mathcal{S}_0) \in S \mid \mathcal{S}_0 \models Inv(\ell_0)\}$,
- $Act = Stmt \cup \mathbb{R}_{\geq 0}$,
- and a transition $t \in \rightarrow$ of the transition relation $\rightarrow \subseteq S \times Act \times S$ is either a delay transition that increases all clocks with a value $\delta \geq 0$:

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

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

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

C. Abstract semantics

The abstract semantics of a TCFA can simply be defined by extending the transfer relation with transitions that abstract time delay. For TCFA, the transfer relation is of the form $\rightsquigarrow \subseteq E \times (Stmt \cup \{\mathbf{delay}\}) \times E$, and the following additional property holds:

- $\bigcup_{s \in \gamma(e)} \{s' \in S \mid s \xrightarrow{\delta} s'\} \subseteq \bigcup_{e' \in \mathbf{delay}_e} \gamma(e')$ for all $e \in E$ and $\delta \in \mathbb{R}_{\geq 0}$.

Alternatively, for the analysis of reachability properties, an abstract combined step semantics [7] can be defined where a transition is a combination of a single delay and a discrete transition.

D. Connection to TAs and CFAs

The formulation above admits a simple description of both CFAs and TAs. A timed automaton can be considered a TCFA where $Clock = Var$, $Urg = \emptyset$ and only statements of the form $[\varphi](; x := 0)^*$ are allowed where $x \in Clock$. Here, $[\varphi]$ is a guard and $x := 0$ is a clock reset. A CFA on the other hand is a TCFA where $Clock = \emptyset$, $Urg = Loc$ and $Inv(\ell) = \mathbf{true}$ for all $\ell \in Loc$.

Moreover, given a TCFA with $Inv(\ell_0) = \mathbf{true}$, it can be transformed to a semantically equivalent CFA by applying the following simple steps:

- *Eliminating clock variables.* For all variables $x : \mathbf{clock}$ of the TCFA, the CFA has a variable $x : \mathbf{real}_{\geq 0}$ such that $Dom(\mathbf{real}_{\geq 0}) = Dom(\mathbf{clock}) = \mathbb{R}_{\geq 0}$.
- *Eliminating location invariants.* For all edges (ℓ, s, ℓ') of the TCFA, the CFA has an edge $(\ell, [Inv(\ell)]; s; [Inv(\ell')], \ell')$. Naturally, invariants equivalent to \mathbf{true} can be omitted.
- *Simulating delay.* For all locations $\ell \in Loc \setminus Urg$ of the TCFA, the CFA has an edge $(\ell, \mathbf{delay}, \ell)$ that simulates delay steps. Here, \mathbf{delay} stands for the statement

$$\mathbf{havoc} \delta; x_1 := x_1 + \delta; \dots; x_n := x_n + \delta; [Inv(\ell)]$$

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

Figure 2 shows the resulting CFA for Fischer’s protocol.

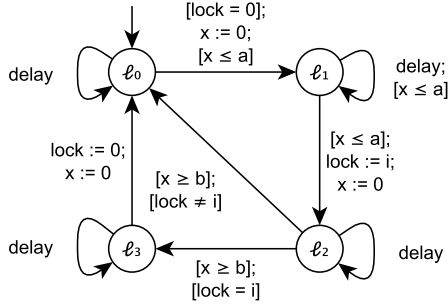


Fig. 2. Fischer’s protocol as a CFA

E. Reachability Analysis of TCFAs

The main advantage of the above formulation is that it admits verifiers to be built compositionally, in the spirit of configurable program analysis [15]. More precisely, given abstractions \mathcal{E}_{data} for data variables and \mathcal{E}_{time} for clock variables with respective transfer relations \rightsquigarrow_{data} and \rightsquigarrow_{time} , a simple analysis can be built that explores the two aspects independently and is a full-fledged verifier for the complete system. Here, \mathcal{E}_{data} is basically a verifier for software that operates on CFAs, and \mathcal{E}_{time} a verifier for timed automata.

As a simple example, Figure 3 illustrates the abstract state space of Fischer’s protocol where \mathcal{E}_{data} is predicate abstraction over a single predicate $lock = i$ (for $i \neq 0$), and \mathcal{E}_{time} is zone abstraction. 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.

IV. CONCLUSIONS AND FUTURE WORK

In this paper, we have described the formalism of timed control flow automata that is an extension of control flow automata with notions of timed automata. We have compared it to the original formalisms, and highlighted a simple method to build verifiers for the formalism by combining verifiers for CFAs and TAs.

In the future, we plan to implement such combined analyses and investigate them in depth. Moreover, to enable modeling of industrial systems, the formalism can be extended with syntax and semantics for parametric behavior and concurrency based on shared variables and handshake synchronization.

REFERENCES

- [1] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.
- [2] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, Y. Wang, and M. Hendriks, “UPPAAL 4.0,” in *Third International Conference on the Quantitative Evaluation of Systems - (QEST’06)*. IEEE, 2006, pp. 125–126.
- [3] J. Bengtsson, J. Bengtsson, W. Yi, and W. Yi, “Timed automata: Semantics, algorithms and tools,” in *Lectures on Concurrency and Petri Nets*, 2004, vol. 3098 LNCS, pp. 87–124.

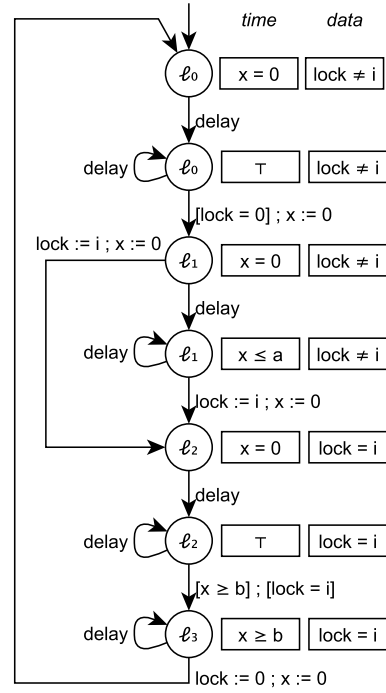


Fig. 3. Abstract Reachability Graph for Fischer’s protocol

- [4] G. Morb e, F. Pigorsch, and C. Scholl, “Fully Symbolic Model Checking for Timed Automata,” in *Computer Aided Verification*, 2011, vol. 6806 LNCS, pp. 616–632.
- [5] A. Carioni, S. Ghilardi, and S. Ranise, “MCMT in the Land of Parameterized Timed Automata,” *Proceedings of VERIFY*, pp. 1–16, 2010.
- [6] R. Kindermann, T. Junttila, and I. Niemel a, “Beyond Lassos: Complete SMT-Based Bounded Model Checking for Timed Automata,” in *Formal Techniques for Distributed Systems*, 2012, pp. 84–100.
- [7] —, “SMT-based Induction Methods for Timed Systems,” *Formal Modeling and Analysis of Timed Systems*, vol. 7595 LNCS, pp. 171–187, 2012.
- [8] T. Isenberg and H. Wehrheim, “Timed Automata Verification via IC3 with Zones,” in *Formal Methods and Software Engineering*, 2014, pp. 203–218.
- [9] T. Isenberg, “Incremental Inductive Verification of Parameterized Timed Systems,” in *Application of Concurrency to System Design (ACSD)*, 2015, pp. 1–9.
- [10] K. Hoder and N. Bj orner, “Generalized Property Directed Reachability,” in *Theory and Applications of Satisfiability Testing SAT 2012*, vol. 7317 LNCS, 2012, pp. 157–171.
- [11] H. Hojjat, P. R ummer, P. Subotic, and Wang Yi, “Horn Clauses for Communicating Timed Systems,” *Electronic Proceedings in Theoretical Computer Science*, vol. 169, pp. 39–52, 2014.
- [12] D. Beyer, “Software Verification and Verifiable Witnesses,” in *Tools and Algorithms for the Construction and Analysis of Systems*, 2015, vol. 9035 LNCS, pp. 401–416.
- [13] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement for symbolic model checking,” *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.
- [14] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, S. F. Univers, and R. Sebastiani, “Software Model Checking via Large-Block Encoding,” in *Formal Methods in Computer-Aided Design*, 2009, pp. 25–32.
- [15] D. Beyer, T. A. Henzinger, and G. Th eoduloz, “Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis,” in *Computer Aided Verification*, 2007, vol. 4590 LNCS, pp. 504–518.