Verification of Real-time Systems by Abstraction of Time Constraints

Mustapha Bourahla  
Computer Science Department  
University of Biskra  
BP 145 RP, Biskra, 07000, Algeria  
mbourahla@hotmail.com

Mohamed Benmohamed  
Computer Science Department  
University of Constantine  
Constantine 25000, Algeria  
ibnm@yahoo.fr

Abstract

This paper presents a new methodology for model checking real-time systems based on the abstraction of time predicates. A real-time system is modeled with a timed automaton which is translated to a real-time program. The properties are specified with the temporal logic TCTL (Timed Computational Tree Logic). The real-time program and the TCTL property are used first, for producing a new automaton which augments the original with auxiliary clocks capturing the timing constraints in the TCTL specification that is reduced to an equivalent CTL specification. Second, the augmented real-time program is converted to a well timed system by removing the zeno runs (that are executions in which time does not diverge). Then the time predicates in the augmented automaton which is represented by an augmented and no-zeno real-time program will be abstracted to generate an untimed automaton representing an equivalent finite state system to be model checked using existing tools. 

Keywords: Real-time systems, Formal Verification, Model Checking, Timed Automaton, Predicate Abstraction.

1. Introduction

Timed systems are more and more frequently used in critical applications such as traffic controllers, real-time operating systems and so on. These systems must meet real-time constraints. Formal techniques have been proposed to ensure the correctness of such systems which is critical. The objective of this paper is to show how to verify a concrete timed system described by a timed automaton using an equivalent abstract system described by an untimed automaton. This abstract system has a reduced number of states and it can be verified using the existing infrastructure in tools and algorithms for the verification of untimed finite-state systems.

To model timed systems, we use timed automata [2]. These are finite-state automata equipped with clocks used to specify constraints on the amount of time that can elapse between two events. The particularity of this model is that it uses a dense time domain which results in an a-priori infinite state space. However, the so-called region equivalence [1] reduces the state space of a given timed automaton into a finite graph (the region graph) which preserves sufficient information for verification. Untimed verification techniques can be in principle applied on the region graph to check timed properties of the original automaton. The region graph is too large to be of any practical interest. Its size is exponential in the number of clocks of the system as well as in the size of the constants used in the timing constraints (using a combinatorial argument, the authors of [1] have shown that the number of regions is at most \( n! \times 2^n \times (2 \times c + 2)^n \), where \( n \) is the number of clocks).

To overcome this explosion, the authors of [19] have proposed the technique of time-abstracting bi-simulations. This technique produces a finite graph like the region graph called the quotient of the automaton with respect to a time-abstracting bi-simulation and it has more abstracted time delays. Then, use the quotient to perform verification. The quotient graph is still too big (its size is exponential in the number of clocks) and the results shown in [18] for small designs indicated that the state space is blown up.

We propose a method to interpret the behavior of the original system in an abstract system using the approach of predicate abstraction [6] [10] [13] which is based on the abstract interpretation framework to abstract time predicates. The abstract system will reduce abnormally the number of states by abstracting away the time delays. The idea of time predicate abstraction is to represent the atomic predicates representing clock constraints by abstract Boolean variables.

For the verification we use the branching-time logic TCTL as a property-specification language. The user provides a TCTL formula \( \varphi \) which expresses the desired property. Then, we generate the abstract system with respect to the time predicate abstraction. In case \( \varphi \) does not involve any timing constraints (i.e., \( \varphi \) is a CTL formula), the ab-
A timed automaton $\mathcal{A}$ is defined as a tuple $\mathcal{A} = (X, \mathcal{L}, \mathcal{Q}, q_0, E, \Box)$, where:

- $X$ is a finite set of clocks.
- $\mathcal{L}$ is a finite set of labels.
- $\mathcal{Q}$ is a finite set of discrete states, $q_0 \in \mathcal{Q}$ being the initial discrete state.
- $E$ is a finite set of edges of the form $e = (q, \varsigma, a, X, q')$, $q, q' \in \mathcal{Q}$ are the source and target discrete states, $a \in \mathcal{L}$ is a label, $\varsigma$ is a conjunction of atomic constraints on $X$ defining an $X$-polyhedron, called the guard of $e$. $X \subseteq X$ is a set of clocks to be reset upon crossing the edge.

- $\Box$ is a function associating with each discrete state $q$ an $X$-polyhedron called the invariant of $q$. Note that $\zeta \in \Box(q)$ of an edge $e = (q, \varsigma, a, X, q')$.

A clock is a variable ranging in $\mathbb{R}$ (the set of non-negative reals). The state of a system is determined by the values of a finite set $Pr$ of Boolean variables (propositions), representing data and control, and by the values of a finite set $X$ of real-valued variables (clocks). The clocks allow the system to make time-dependent decisions. A state $\sigma$ is an interpretation of all propositions and clocks $\sigma : \{Pr \cup X\} \rightarrow \{true, false\} \cup \mathbb{R}$. A state $\sigma$ satisfies $\phi \iff \sigma[\phi] = true$. Also, $\sigma$ is a Boolean value $\sigma(p) \in \{true, false\}$ and to each clock $x \in X$ a nonnegative real $\sigma(x) \in \mathbb{R}$. We write $\Sigma$ for the set of all states. A state $\sigma$ of $\mathcal{A}$ is a pair $(q, \vartheta)$, where $q \in \mathcal{Q}$ is a discrete state, and $\vartheta \in \Box(q)$ is a clock valuation satisfying the invariant of $q$. We write $\sigma^0$ to denote $q$, the discrete part of $\sigma$. The initial state $\mathcal{A} = (\sigma_0 = (q_0, \vartheta_0)). \vartheta_0$ is the valuation that assigns zero to a clock.

Consider a state $(q, \vartheta)$, and an edge $e = (q, \varsigma, a, X, q')$ such that $\vartheta \in \varsigma$ and $\vartheta' = \vartheta + \delta$ (reset($e$)) if $\delta \leq 0 \in \Box(q)$, $(q, \vartheta) \xrightarrow{\delta'} (q', \vartheta')$ is a discrete transition of $\mathcal{A}$. $(q, \vartheta) \xrightarrow{\delta}$ is called the $\delta$-successor of $(q, \vartheta)$. A time transition from $(q, \vartheta)$ has the form $(q, \vartheta) \xrightarrow{\delta} (q, \vartheta + \delta)$, where $\delta \in \mathbb{R}$ and $\vartheta + \delta \in \Box(q)$.

Definition 2 (Semantics of Timed Automata). We associate two kinds of semantics to a timed automaton: a branching-time semantics in terms of a labeled graph and a linear-time semantics in terms of executions (runs).

The semantic graph of $\mathcal{A}$, denoted $G_\mathcal{A}$, is defined to be the graph which has as nodes the states of $\mathcal{A}$ and two types of edges, corresponding to the discrete and time transitions of $\mathcal{A}$. $G_\mathcal{A}$ has generally an uncountable set of nodes and uncountable branching.

A run of $\mathcal{A}$ starting from state $\sigma$ is a finite or infinite sequence $\rho = \sigma_1 \xrightarrow{\delta_1} \sigma_1 + \delta_1 \xrightarrow{\varepsilon_1} \sigma_2 \xrightarrow{\delta_2} \sigma_2 + \delta_2 \xrightarrow{\varepsilon_2} \cdots$, such that $\sigma = \sigma_1$ and for all $i = 1, 2, \ldots, \sigma_i + \delta_i$ is the $\delta$-successor of $\sigma_i$ and $\sigma_{i+1}$ is the $\varepsilon_i$-successor of $\sigma_i + \delta_i$. That is, a run is an execution along a path in the semantic graph of $\mathcal{A}$ where discrete transitions are taken infinitely often and consecutive time transitions are concatenated. We denote $\sigma_i$ by $\rho(i)$, $\delta_i$ by delay($\rho$, i), and $\sum_{1 \leq j \leq i} \delta_j$ by time($\rho$, i).

The limit of the sequence $\text{time}(\rho, i)$ as $i \rightarrow \infty$ is denoted $\text{time}(\rho)$.

Definition 3 (Non-zeno runs). Consider an infinite run $\rho$ such that $\text{time}(\rho) \neq \infty$, that is, there exists $t \in \mathbb{R}$ such that for all $i$, $\text{time}(\rho, i) < t$. Such a run, called zeno, corresponds to a pathological situation, since it violates the time-progress requirement. A non-zeno run is a run $\rho$ such that $\text{time}(\rho) = \infty$.

Definition 4 (Strongly non-zeno Timed Automaton). Consider a timed automaton $\mathcal{A}$. A structural
loop of $A$ is a sequence of distinct edges $e_1, \cdots, e_m$ such that $\text{target}(e_i) = \text{source}(e_{i+1})$, for all $i = 1, \cdots, m$ (the addition $i + 1$ is modulo $m$). $A$ is called strongly non-zeno if for every structural loop there exists a clock $x$ and some $1 \leq i, j \leq m$ such that:

- $x$ is reset in step $i$, that is, $x \in \text{reset}(e_i)$; and
- $x$ is bounded from below in step $j$, that is, $(x \geq 1) \wedge \text{guard}(e_j) = \text{true}$.

**Corollary 1** A strongly non-zeno timed automaton is also timelock-free.

The notion of non-zenoness was introduced at the same time as timed automata. Strong non-zenoness is interesting since it dispenses us with the burden of ensuring time progress. In particular, since strongly non-zeno timed automaton is also timelock-free, checking progress is reduced to checking deadlock-freedom.

### 3. The FDDI Protocol

FDDI (Fiber Distributed Data Interface) (example taken from [11]) is a high performance fiber optic token ring Local Area Network. We consider a network composed by $N$ identical stations $S_1, \ldots, S_N$ and a ring, where the stations can communicate by synchronous messages with high priority and asynchronous messages with low priority (Figure 1).

Each station $S_i$ can be either waiting for the token (location 0), in possession of the token and executing the synchronous transmission (location 1) or in possession of the token and executing the asynchronous transmission (location 2). The two clocks a station uses to control the possession time of the token are called $TTRT$ (Token Rotation Timer) and $THT_i$ (Token Holding Timer).

- $TTRT$ counts the time since the last reception of the token by the station. This clock is reset to zero each time the station $S_i$ takes the token.
- $THT_i$ counts the time since the last reception of the token, added to the time elapsed since the precedent one, given by the value of the clock $THT_i$ just before it is reinitialized.

When the station $S_i$ receives the token ($TT_i$), the clock $THT_i$ takes the value of the clock $TTRT$, $THT_i$ is reset to zero, and the station $S_i$ starts sending synchronous messages. The duration of the synchronous transmission is given, for each station $S_i$, by a constant $S_{Ai}$ (Synchronous Allocation). When the synchronous transmission ends, the station has the possibility of starting the transmission of asynchronous messages if the current value of $THT_i$ minus the time of synchronous transmission ($S_{Ai}$) is less than a global constant of the system called $TTRT$ (Target Token Rotation Timer). Before $THT_i - S_{Ai}$ reaches the value $TTRT$, the station must release the token ($RT_i$), ending the asynchronous transmission if this one has began. The behavior of the station $S_i$ is described by the timed automaton $Station_i$ of Figure 1 (a).

The ring controls the transmission of the token between two consecutive stations $S_i$ and $S_{i+1}$. Figure 1 (b) shows the timed automaton $Ring$ that models the ring for two stations.

The timed automaton that models the protocol is obtained as the parallel composition $FDDI_N = Ring \mid Station_1 \mid \cdots \mid Station_N$, where the automata synchronize through the actions $TT_i$ and $RT_i$. The set $X \subseteq X'$ of the reset clocks contains an assignment of a clock value to another clock ($THT_i := TRT_i$) this will not affect the decidability of model checking because these details will be abstracted.
4. Real-time Programs

The timed automaton $\mathcal{A}$ will be translated to a real-time program which consists of a set of guarded commands, a program invariant and a program initial state.

Definition 5 (syntax of real-time program). The syntax of a real-time program $\mathcal{P} = (\mathcal{R}, \varphi^i, \varphi^{init})$ consists of

1. $\mathcal{R}$ is the program body, a predicate of the system transition relation. This predicate is a disjunctive formula of a set of conjunctive sub-formulas. Each conjunctive sub-formula representing a transition in the system which is of the form $\psi \wedge A$, for a state predicate $\psi$ (the guard of the command) and a set $A = \{v := a_v \mid v \in Pr \cup X\}$ of simultaneous assignments such that for all propositions $p \in Pr$, the expression $a_p$ is a state predicate, and for all clocks $x \in X$, the expression $a_x$ is either 0 (i.e., the clock $x$ is reset), or assignment of another clock value or $x$ (i.e., the clock $x$ is left unchanged).

2. $\varphi^i$ is the program invariant, a state predicate. We require $\varphi^i$ to verify the condition, that is, for all states $\sigma \in \Sigma$ and all delays $\delta \in R^+$, if $\sigma + \delta \models \varphi_i$ then $\sigma \models \varphi^i$.

3. $\varphi^{init}$ is the predicate of initial state.

Definition 6 (semantics of real-time program). The real-time program $\mathcal{P} = (\mathcal{R}, \varphi^i, \varphi^{init})$ defines the transition relation $\mathcal{R}_P$ such that $(\sigma, \sigma') \in \mathcal{R}_P$ if

1. either $\sigma' = \sigma$ or for some guarded command $g = \psi \wedge A \in \mathcal{R}$, $\sigma \models \psi$ and $\sigma' = [A]$, $\sigma \models \varphi^i$;

2. $\sigma \models \varphi^i$ and $\sigma' \models \varphi^i$.

Example 1 The real-time system of FDDI is defined by the following real-time program $\mathcal{P}$, where $SA_1 = SA_2 = 20$ and $TTRT = 50$.

$$\mathcal{P} = (\mathcal{R}, \varphi^i, \varphi^{init})$$

$$\varphi^i = \{ \sigma^0 \wedge [\sigma^0 := q_0 \wedge THT_1 := T T R T_1 \wedge T R T_1 := 0] \}$$

$$\varphi^{init} = \{ \sigma^0 = [q_0 \wedge T R T_1 = 0 \wedge T H T_1 = 0 \wedge T R T_2 = 0 \wedge T H T_2 = 0] \}$$

This real-time program defines a transition relation for the FDDI system ($St_{ation1} \parallel St_{ation2} \parallel Ring$). The set of discrete states is $Q = \{q_0 \equiv 000, q_1 \equiv 101, q_2 \equiv 201, q_3 \equiv 002, q_4 \equiv 013, q_5 \equiv 023\}$.

5. Timed Computational Tree Logic

Many important properties of systems find a natural expression in the real-time temporal logic TCTL, which extends the branching time logic CTL with clock variables. The formulas of TCTL, which are interpreted over the states of a given model $M$, are built from state predicates by Boolean connectives, the two temporal until operators $\exists U$ (possibly) and $\forall U$ (necessarily). Intuitively, the formula $\exists U q[pUq]$ holds in a state $\sigma$ of $M$ if and only if the proposition $q$ becomes true on some (every) path in $M$ that starts from $\sigma$, and the proposition $p$ is true until $q$ becomes true. The formulas of TCTL contain two kinds of variables. Propositional variables from $Pr$ and clock variables from $X$ occur freely and refer to the states of the given model $M$.

Definition 7 (syntax of TCTL). The formulas $\varphi$ of the timed computation tree logic TCTL are defined inductively by the grammar

$$\varphi ::= true \mid p \mid \neg \varphi \mid \varphi \wedge \varphi \mid \varphi \exists I \varphi \mid \varphi \forall I \varphi,$$

where $p \in Pr$ is an atomic proposition using proposition and/or clock variables and $I \in I$ is an interval in the set of intervals $I$ appearing in $\varphi$.

The temporal operators $\exists \varphi$, $\forall \varphi$, $\exists O \varphi$, $\forall O \varphi$, $\exists I \varphi$, $\forall I \varphi$, $\exists O \varphi$, $\forall O \varphi$, $\exists I \varphi$, $\forall I \varphi$, and $\exists O \varphi$, $\forall O \varphi$, $\exists I \varphi$, $\forall I \varphi$, are evaluated in states. We simplify notation for intervals, for instance, we write $\exists O \varphi$ instead of $\exists O [0, I] \varphi$ and $\forall O \varphi$ instead of $\forall \varphi[0, \infty] \varphi$. Also note that all state predicates are definable in TCTL (for example, the state predicate $x \leq 5$, is using the clock variable $x$). The formulas of TCTL are interpreted over the states of a model. The propositions and the free clock variables of a TCTL-formula $\varphi$ are evaluated in states. We say that the model $M$ of the program $\mathcal{P}$ satisfies a formula $\varphi$ if the initial state of $\mathcal{P}$ satisfies $\varphi$.

Definition 8 (semantics of TCTL). Let $M$ be a model, $\sigma \in \Sigma_M$ be a state reachable in $M$ and let a TCTL-formula $\varphi$. Also let $f_p : Pr \rightarrow 2^\infty$ be a function associating to each atomic proposition a set of discrete states.
The satisfaction relation, denoted by \( \models_{\mathcal{M}} \varphi \), is defined inductively on the syntax of \( \varphi \):

- \( \sigma \models_{\mathcal{M}} \text{true} \)
- \( \sigma \models_{\mathcal{M}} p \) if \( \sigma \in f_{p}(p) \)
- \( \sigma \models_{\mathcal{M}} \neg \varphi \) if not \( \sigma \models_{\mathcal{M}} \varphi \)
- \( \sigma \models_{\mathcal{M}} \varphi_1 \land \varphi_2 \) if \( \sigma \models_{\mathcal{M}} \varphi_1 \) and \( \sigma \models_{\mathcal{M}} \varphi_2 \)
- \( \sigma \models_{\mathcal{M}} \varphi_1 \exists \varphi_2 \) if \( \exists \rho \in \mathcal{M} \) with \( \text{time}(\rho) = \infty \) and \( \rho(0) = \sigma \), \( \exists i. \sum_{j=1}^{i} \delta_j \in I \) and \( \rho(i) + \delta_i \models_{\mathcal{M}} \varphi_2 \) and
- \( \forall j \leq i. \rho(j) \leq \delta_j, \rho(j) + \delta_i \models_{\mathcal{M}} \varphi_1 \)

Example 2 The formula of TCTL that describes the property of the bounded time for sending asynchronous message where each Idle station in the FDDI system will send asynchronous messages before a time \( c \) is:

\[
\varphi = \bigvee \{(S_i = 0) \rightarrow \bigwedge (S_i = 2)\},
\]

where \( S_i = 0 \) is any state \( \sigma \in \Sigma \) verifying the condition that the automaton corresponding to station number \( i \) is in the location 0. \( c \) is equal to \((N - 1) \times TTRT + 2 \times N \times SA_i\).

5.1. TCTL Model Checking

TCTL model checking can be reduced to CTL model checking using the following technique. Given a real-time program \( \mathcal{P} \) to be checked against a TCTL formula \( \varphi \), first we extend \( \mathcal{P} \) with a set of clocks, to obtain a new real-time program \( \mathcal{P}^* \). We may bind certain clock variables to express the timing requirements of a specification. A specification clock \( z \in X_{\varphi} (X_{\varphi} \subseteq X \) is the set of specification clocks) is a clock that does not control the behavior of any system under consideration; that is, we consider only runs \( \rho \) such that for all positions \( i \) of \( \rho \),

\[
\rho(i)(z) = s_{0}(z) + \text{time}(\rho, i).
\]

Then we transform \( \varphi \) to a CTL formula \( \varphi_{\text{CTL}} \); finally, we generate the abstract real-time program \( \mathcal{P}^* \) of \( \mathcal{P} \) after removing the zeno runs, and then model check it against \( \varphi_{\text{CTL}} \). More precisely, let \( Q \) and \( X \) be the set of discrete states and set of clocks of \( \mathcal{P} \). Also let \( I = \{I_1, \ldots, I_n\} \) be the set of non-trivial intervals appearing in \( \varphi \). \( \mathcal{P}' \) has exactly the same structure as \( \mathcal{P} \), except that it has an additional set of clocks \( X_{\varphi} = \{z_1, \ldots, z_m\} \). The set of atomic propositions \( P_{\mathcal{P}} \) is also augmented with two kinds of propositions, namely, \( p_{z_j} = 0 \) and \( p_{z_j} \in I_j \), for each \( j = 1, \ldots, m \). The program body \( \mathcal{P}^* \) and the program initial state \( \mathcal{P}^*_{\text{init}} \) are augmented during the recursive transformation of the formula \( \varphi \) to \( \varphi_{\text{CTL}} \) as follows.

Algorithm 1 (Transform).

{Program, CTL} Transform(Program \( \mathcal{P} \), TCTL \( \varphi \))

\[
\begin{align*}
\text{Program } P_1, P_2 & \leftarrow \{\text{if } \text{true} : \text{true} ;\} \\
\text{CTL } \varphi_{\text{CTL}}, \varphi_{TTL}' & \equiv \text{TCTL } \varphi', \varphi'' ;
\end{align*}
\]

\[
\begin{cases}
\text{switch}(\varphi) & \\
\text{case true } & \text{return } (P_1, \text{true}); \\
\text{case } \rho & \text{return } (P_1, \rho); \\
\text{case } \rho' & \text{return } (P_1, \varphi_{\text{CTL}}'; \rightarrow \text{TCTL}';) \\
\text{case } \varphi' & \text{if } \varphi' \equiv \{P_1, \varphi_{\text{CTL}} ; \rightarrow \text{TCTL} ;\} \\
\text{case } \rho' & \text{return } (P_1, \text{true}); \\
\text{case } \rho & \text{return } (P_1, \rho); \\
\text{case } \rho & \text{return } (P_1, \varphi_{\text{CTL}}'; \rightarrow \text{TCTL}';) \\
\text{case } \varphi' & \text{if } \varphi' \equiv \{P_1, \varphi_{\text{CTL}} ; \rightarrow \text{TCTL} ;\} \\
\text{case } \rho' & \text{return } (P_1, \text{true}); \\
\text{case } \rho & \text{return } (P_1, \rho); \\
\text{case } \rho & \text{return } (P_1, \varphi_{\text{CTL}}'; \rightarrow \text{TCTL}';) \\
\text{case } \varphi' & \text{if } \varphi' \equiv \{P_1, \varphi_{\text{CTL}} ; \rightarrow \text{TCTL} ;\}
\end{cases}
\]

Example 3 As an example, we take the real-time program \( \mathcal{P} \) of the system FDDI presented in the previous section and the TCTL-formula

\[
\varphi = \bigvee \{(S_1 = 0) \rightarrow \bigwedge (S_1 = 2)\},
\]

whose contains only one interval for timing constraint, then \( X_{\varphi} = \{z\} \). The result after running the Transform algorithm is the equivalent CTL formula \( \varphi_{\text{CTL}} \) and the extended program \( \mathcal{P}' \)

\[
\varphi_{\text{CTL}} = \bigvee \{(S_1 = 0) \rightarrow (z_1 = 0 \bigwedge (S_1 = 1))\},
\]

\[
\mathcal{P}' = \mathcal{P} \{(z_1 = 0 \bigwedge S_1 = 2 \bigwedge z \leq 100) \bigwedge (z_1 = 1))\; \\
\text{true} ; (z_1 = 0 \bigwedge z \leq 0)\}
\]

The produced program \( \mathcal{P}' \) represents a model of an infinite state-transition system which means that it is not possible to apply CTL model checking to verify if this model satisfies the property specified as the generated CTL formula \( \varphi_{\text{CTL}} \). To model check this concrete system we will
interpret the behavior of the program \( P' \) with an equivalent abstract finite system using the framework of predicate abstractions. Before this, the concrete system should be converted to a well-timed system, which means a non-zeno real-time system.

6. Removing zenoness

It is important to check if a real-time program \( P \) is non-zeno and, if not, to convert \( P \) into an equivalent non-zeno real-time program. To check if \( P \) is non-zeno, we compute the characteristic predicate of the possibility requirement

\[
\psi_{nz} = \forall \sigma (\sigma \rightarrow ((\exists z \sigma \rightarrow (z \geq 1)) \lor (\neg \exists z \sigma)),
\]

which is true in all states from which time can advance by at least 1 time unit. To do this, we have to add the action \( z := 0 \) to any transition leading to the state \( \sigma \).

To convert a zeno real-time program \( P \) into an equivalent non-zeno program, we need to strengthen the program invariant so that the new invariant rules out states from which time cannot diverge. The zeno states are the states that do not satisfy the above formula which is true in all states that occur on a \( R_P \)-path along which time advances infinitely often by at least 1 time unit; that is, \( \psi_{nz} \) characterizes precisely the states that occur on runs of \( P \).

**Proposition 1** For every real-time program \( P \), \( \psi_{nz} = s \rightarrow \Sigma_P \).

Since the invariant \( \psi^n \) of any non-zeno real-time program \( P \) defines precisely the set \( \Sigma_P \) of states that occur on runs of \( P \), the invariant \( \psi^{nz} \) is equivalent to the characteristic predicate of \( \psi_{nz} \). On the other hand, if the real-time program \( P \) is zeno, then the invariant \( \psi^{nz} \) is weaker than \( C_{\psi_{nz}} \) and can be replaced by \( C_{\psi_{nz}} \) to obtain an equivalent non-zeno program.

**Theorem 1** For every real-time program \( P \) there is an equivalent non-zeno real-time program, which can be obtained from \( P \) by replacing the program invariant with the state predicate represented by \( C_{\psi_{nz}} \).

**Algorithm 2** (Non-zeno real-time program).

Program NonZeno \((R, \psi^n, \psi^{inst})\)

\[
C_{\psi_{nz}} \leftarrow \psi^n; \delta \leftarrow 0; \quad \text{TransList} \leftarrow \emptyset; \quad \sigma \leftarrow \psi^{inst} \land \psi^n;
\]

\[
\text{do} \quad \{ \exists \sigma = \psi \land A \in R \mid \sigma [\psi] = \text{true and } \sigma [A] \in C_{\psi_{nz}} \} \quad \{ \exists \sigma [A] \in \text{TransList} \land \exists \rho \in \text{TransList} \mid \rho = \sigma [A] \land \sum_{i=1}^{m} \sigma_i \land \delta = 0 \} \quad \{ \Sigma_{\sum_{i=1}^{m} \delta_i = 0} \}
\]

\[
\text{TransList} \leftarrow \text{TransList} + \{ \sigma \leftarrow \sigma [\delta]\}; \sigma \leftarrow \sigma [\delta]; \delta \leftarrow 0;
\]

\[
\} \quad \text{else} \{ \delta \leftarrow \delta + 1; \quad \text{Update all the clocks; } \sigma \leftarrow \sigma + \delta; \}
\]

\[
\text{while } \sigma \notin \text{TransList}; \quad \text{return } [R \cap \text{TransList}, C_{\psi_{nz}}, \psi^{inst}];
\]

The operation \( R \cap \text{TransList} \) takes transitions off \( R \) that are never executed or they are part of zeno runs. This algorithm uses only predicates on clock variables \( (X \cup X_{\nu}) \) to transit from state to state. The new non-zeno real-time program \( P^{nz} \) does not contain any zeno run that can be present in the original real-time program. As an example, we will use the produced non-zeno real-time program \( (P^{nz}) \) corresponding to the augmented real-time program \( (P') \) of Section 5. The result of applying this algorithm on \( P' \) has given a non-zeno program with same components of \( R \) and \( \psi^{nz} \), which means that the original program is non-zeno.

7. Interpretation using Predicate Abstraction

The model of the augmented non-zeno real-time program \( (P^{nz}) \), denoted \( M^* \), is interpreted by an abstract model \( (M^*) \) using the framework of predicate abstraction. Predicate abstraction consists of using predicates over clock variables as Boolean abstract variables. It can be defined in the framework of abstraction interpretation [8] [9] using Galois connections.

**Definition 9** (Abstraction by Galois Connection). Let \( \Sigma^i \) and \( \Sigma^a \) represent the concrete and abstract state domains respectively. A Galois connection \([3]\) from \( \Sigma^i \) to \( \Sigma^a \) is a pair of functions \( \alpha : 2^{\Sigma^i} \rightarrow 2^{\Sigma^a} \) and \( \gamma : 2^{\Sigma^a} \rightarrow 2^{\Sigma^i} \) such that:

- \( \alpha \) and \( \gamma \) are total and monotonic.
- \( \forall S \in 2^{\Sigma^i}, \gamma \circ \alpha (S) \supseteq S \), and
- \( \forall S \in 2^{\Sigma^a}, \alpha \circ \gamma (S) \supseteq S \).

Each constraint on a set of clocks \( X \subseteq X' \), will be interpreted by an abstract Boolean variable \( B_i \). The valuations on \( X (R^{i+}) \) are interpreted to affect a value \( \text{(true or false)} \) to each abstract Boolean variable \( B_i \). The valuations \( \theta^i \) and \( \theta + \delta \) (time elapse by \( \delta \)) are considered as actions and they will affect the different values of abstract Boolean variables.

**Definition 10** (Abstract timed automaton). An abstract timed automaton is a tuple \( A^* = (B, \mathcal{L}, \Sigma^a, \sigma^0, E^a, \square^a) \), where:

- \( B \) is a finite set of abstract Boolean variables.
- \( \mathcal{L} \) is a finite set of labels.
\[ \Sigma^a \text{ is a finite set of discrete states, } \sigma^a \in \Sigma^a \text{ being the initial discrete state.} \]

\[ \mathcal{E}^a \text{ is a finite set of abstract edges of the form } e^a = (\sigma^a, \zeta^a, \alpha, \sigma^d). \sigma^a, \sigma^d \in \Sigma^a \text{ are the source and target discrete states. } \alpha \in \mathcal{L} \text{ is a label. } \zeta^a \text{ is an abstract } X\text{-polyhedron, called the guard of } e^a. \]

\[ \Box^a \text{ is the abstract invariant.} \]

A state \( \sigma^a \) of \( \mathcal{A}^a \) is an expression of the form \( (\sigma^e \land B^a) \), where \( \sigma^e \in \Sigma^e \) is a discrete state, and \( B^a \) is a Boolean expression giving different Boolean values to different variables satisfying the invariant of \( \sigma^e \). The discrete transition has the form \( \sigma^e \land B^a \rightarrow \sigma^d \land B^d \) and the time transition has the form \( \sigma^e \land B^a \rightarrow \sigma^d \land B^d \).

**Lemma 1** The abstract timed automaton \( \mathcal{A}^a \) preserves the semantics of the concrete timed automaton \( \mathcal{A} \).

**Lemma 2** The time predicate abstraction does not preserve the zeno runs.

As a conclusion of this lemma, we don’t have timelocks in the abstract real-time program. This is not surprising, since the predicate abstractions are insensitive to exact delays. But, we have to remove the non-zeno runs to avoid the undesired behaviors and then having only strong preservation. The abstract timed automaton is represented by an abstract real-time program which will be computed automatically from the original real-time program representing the original automaton. This abstract program has the same structure as the original, \( \mathcal{P}^a = (B^a, \varphi^a, \varphi_{init}^a) \). If \( P \) is a time predicate over clock variables, a predicate abstraction can be expressed as a Galois connection:

\[
\alpha(P) = \bigwedge \{ B^a | P \Rightarrow \gamma(B^a) \} = P^a,
\]

where \( B^a \) is any Boolean expression over the set \( \{B_1, ..., B_k\} \) which is the set of abstract variables corresponding to the set of concrete time predicates (the atomic constraints on the clocks) taken from the concrete model, \( \{\phi_1, ..., \phi_k\} \). \( \gamma \) is defined as a substitution function, that is, \( \gamma(P^a) = P^a[\phi_1/B_1, ..., \phi_k/B_k] \), where each Boolean variable \( B_i \) is substituted by its corresponding correct predicate \( \phi_i \).

Thus, in order to compute for a concrete real-time system \( \mathcal{M}^e \), an abstract real-time system \( \mathcal{M}^a \), it is sufficient to abstract the initial state \( \varphi_{init}^a \) by computing \( \alpha(\varphi_{init}^a) \), to abstract the invariant \( \varphi^a \) by computing \( \alpha(\varphi^a) \), and to abstract each transition \( \tau \in \mathcal{T}(\mathcal{T}) \) is the set of transitions calculated by the transition relation \( \mathcal{R}_P \) as follows:

\[
\tau^a = \alpha(\tau) = \alpha(\text{action}_P(V, V')) = \bigwedge \{(B^a, B'^a) | \text{post}([\tau](\gamma(B^a))) \Rightarrow \gamma(B'^a)\},
\]

that is, the pair \( (B^a, B'^a) \) characterizing the abstraction of the set of possible predecessors by \( \tau \) and the abstraction of the set of possible successors by \( \tau \), where \( \text{post} \) expresses the strongest post condition by a transition \( \tau \) of a predicate \( P \) over the state variables \( V = Pr \cup X \) (proposition and clock variables), it is defined as follows:

\[
\text{post}([\tau](P)) = \exists V'. \text{action}_P(V', V) \land P(V'),
\]

where \( \text{action}_P(V', V) \) is defined as the relation between the current state and next state, that is the expression:

\[
(\sigma^a = q_j) \land \text{guard} \land \bigwedge_{i=1}^{l} \left( i \equiv a_i \land (\text{next}(\sigma^a) = q_j) \right),
\]

where \( \text{next}(\sigma^a) = q_j \) means that the value of \( \sigma^a \) equals \( q_j \) in the next state of \( \sigma \). Each abstract state is then a conjunction of a subset of the set of Boolean variables which are the codes of the finite abstract domain. The concretization of an abstract state is a set of concrete states that can be represented as a predicate. The discrete states are not abstracted because they are of finite type. The preservation of properties expressed in temporal logic is established via equivalences and preorders between the concrete and abstract models.

Therefore, we have two kinds of preservation: weak preservation and strong preservation.

**Theorem 2** Any time predicate abstraction of \( \mathcal{P} = (\mathcal{R}^{nz}, \varphi_{nz}, \varphi_{init}) \) is a strong preservation.

**Example 4** The following is the abstraction of the timed system representing the FDDL protocol after transformation of the specification \( \varphi \) presented in Example 3 of Section 5, to CTL and removing zeno runs (the function \( \gamma \) is the substitution function).

\[
\varphi \equiv (B_1 \rightarrow z = 0 \land B_2 \rightarrow T \land T = 0 \land B_3 \rightarrow T = 1 \land B_4 \rightarrow T = 2 \land B_5 \rightarrow T = 3 \land B_6 \rightarrow T = 4 \land B_7 \rightarrow T = 5 \land B_8 \rightarrow T < 10 \land B_9 \rightarrow T < 20 \land B_{10} \rightarrow T < 30 \land B_{11} \rightarrow T < 40 \land B_{12} \rightarrow T < 50 \land B_{13} \rightarrow T < 60 \land B_{14} \rightarrow T < 70, T < 80).
\]

\[
\mathcal{R}^a = \{ (\sigma^a = q_0 \land \text{next}(B_2) \land \text{next}(\sigma^a) = q_1) \lor \\
(\sigma^a = q_1 \land B_6 \land B_8 \land \text{next}(\sigma^a) = q_2) \lor \\
(\sigma^a = q_2 \land B_6 \land B_{12} \land \text{next}(\sigma^a) = q_3) \lor \\
(\sigma^a = q_3 \land B_8 \land \text{next}(\sigma^a) = q_4) \lor \\
(\sigma^a = q_4 \land B_8 \land B_6 \land \text{next}(\sigma^a) = q_5) \lor \\
(\sigma^a = q_5 \land B_8 \land \text{next}(\sigma^a) = q_6) \lor \\
(\sigma^a = q_6 \land \text{next}(\sigma^a) = q_7) \}
\]

\[
\varphi^{init} = \{ \sigma^a \in \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\} \land \\
(\sigma^a = q_1 \rightarrow B_1) \land (\sigma^a = q_2 \rightarrow B_12 \lor B_8) \lor \\
\}
\]
\[(\sigma^e = q_0 \rightarrow B(1)3) \land (\sigma^e = q_5 \rightarrow B1 \lor B9)\]
\[\varphi^{ext\alpha} = (\sigma^e = q_0 \land B1 \land B2 \land B3 \land B4 \land B5)\]

Now the abstract model described by this program can be model checked to verify the generated CTL formula. We have written an equivalent model using the SMV language and then used the SMV tool to verify the satisfaction of the specification. The SMV tool has reported an abstract error trace indicating the non-satisfaction of the property by the abstract model. The following is the error trace containing only the discrete states \((S_1 \parallel S_2 \parallel R \parallel Spec)\):

<table>
<thead>
<tr>
<th>0000</th>
<th>0101</th>
<th>1010</th>
<th>0001</th>
<th>0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0130</td>
<td>0000</td>
<td>1010</td>
<td>0020</td>
<td>0130</td>
</tr>
<tr>
<td>0230</td>
<td>0000</td>
<td>1010</td>
<td>0020</td>
<td>0130</td>
</tr>
<tr>
<td>0000</td>
<td>1010</td>
<td>2010</td>
<td>0020</td>
<td>0130</td>
</tr>
</tbody>
</table>

By following the trace and accumulating the time on the specification clock \(z\), we find that the time constraint is violated. The specification \(\varphi\) has been changed to

\[\varphi = \forall\Box((S_1 = 0) \rightarrow \forall\Box_{\leq 130}(S_1 = 2)).\]

By the same way we have generated the equivalent CTL formula and the abstract program. The only difference with the first abstract program is that this one does not contain the transition \((\sigma^e = q_2 \land B8 \land next(\sigma^e) = q_3)\) and the functions \(\alpha\) and \(\gamma\) have their definitions modified where \(B10\) has been changed to \(B10 \leftrightarrow z \leq 130\). This specification is checked by the SMV tool and it is satisfied by the generated abstract model.

8. Conclusion

A novel methodology based on the abstraction of time predicates is presented for model checking real-time systems. This methodology extends the original automaton of the real-time system by auxiliary clocks capturing the timing constraints in the TCTL property. The TCTL specification is then reduced to an equivalent CTL specification. The resulting automaton represents an infinite state-transition system due to the dense real time. Consequently, we have to abstract the real-time system to an equivalent untimed finite abstract state-transition system using the framework of predicate abstraction. Before abstracting the real-time system, it is convenient to remove the zero runs which can occur to produce spurious behaviors. All the processes in this methodology are operating on a defined syntax and semantics for real-time programs.

References