Journal of Computing and Information Technology - CIT 17, 2009, 3, 265–284
doi:10.2498/cit.1001272

Verification of Complex Real-time Systems Using Rewriting Logic

Mustapha Bourahla

Computer Science Department, University of Biskra, Algeria

This paper presents a method for model checking dense complex real-time systems. This approach is implemented at the meta level of the Rewriting Logic system Maude. The dense complex real-time system is specified using a syntax which has the semantics of timed automata and the property is specified with the temporal logic TLTL (Timed LTL). The well known timed automata model checkers Kronos and Uppaal only support TCTL model checking (a very limited fragment in the case of Uppaal). Specification of the TLTL property is reduced to LTL and its temporal constraints are captured in a new timed automaton. This timed automaton will be composed with the original timed automaton representing the semantics of the complex real-time system under analysis. Then, the product-timed automaton will be abstracted using partition refinement of state space based on strong bi-simulation. The result is an untimed automaton modulo the TLTL property which represents an equivalent finite state system to be model-checked using Maude LTL model checking. This approach is successfully tested on industrial designs.

Keywords: complex real-time systems, rewriting logic, Maude LTL model checking, timed automaton, strong bi-simulation, partition refinement

1. Introduction

Many formal frameworks that have been proposed to reason about complex real-time systems are based on timed automata [2]. These automata are equipped with clocks, variables used to measure time, ranging over the non negative real numbers (R^+). Consequently, the state space is infinite and cannot be explicitly represented by enumerating all states. Among the different description languages for specifying real-time requirements, we are particularly interested in the temporal logic TLTL [16, 28]. Real-time model checking techniques based on partition refinement [30, 33] build a symbolic state space that is as coarse as possible. Starting from some (implicit) initial partition, the partition is iteratively refined until the verification problem can be decided.

In this paper, we have augmented the LTL syntax used by Maude LTL model-checker [17] by operators to specify TLTL properties. Then, we have proposed a reduction technique from TLTL model-checking to LTL model-checking. This reduction will help us to analyze our system using Maude LTL model checker. Given a complex real-time system specification, a written parser with Maude [12, 13], will generate an equivalent timed automaton $\mathcal{A}$. For a TLTL property specification ψ, we construct an equivalent LTL formula φ and a new timed automaton capturing its temporal behavior. Then, the captured behavior will be composed with the original timed automaton. This is noted by $\mathcal{A}^+$. We prove that $\mathcal{A}$ satisfies ψ if and only if $\mathcal{A}^+$ satisfies φ.

The labeled transition system modeling the behavior of the constructed timed automaton $\mathcal{A}^+$ comprises two kinds of transitions, namely timeless actions representing the discrete evolutions of the system, and time lapses corresponding to the passage of time. Due to density of time, there are infinitely many time transitions. A finite model can be obtained by defining an appropriate equivalence relation inducing a finite number of equivalence classes. The main idea behind these relations is that they abstract away from the exact amount of time elapsed. An important problem consists in constructing the quotient of a labeled transition system (representing a timed automaton) with respect to an equivalence relation.

In this paper we have defined an equivalence relation based on strong bi-simulation [24], which is used by our algorithm to generate the quotient graph. Each edge in the timed automaton represents a discrete transition which has information concerning the source and target states, the enabling condition and the set of clocks to be reset after making this transition. Initially, the timed automaton represents the states of complex timed system as blocks (zones) of states (also called symbolic states). We call this the initial partition of states. We refine any source block of states if there is an outgoing edge with an enabling condition (which is a constraint) formula different from *true*, using the invariant of the block of states and the enabling condition of this transition. The produced sub-blocks represent classes of equivalent states where each sub-block has new invariant that either satisfies or does not satisfy the enabling condition. The refinement process will terminate if there is no block of states to be refined.

1.1. Related work

While LTL model checking is PSPACE complete, the TLTL model checking is undecidable [5]. To our knowledge, it doesn't exist until now a tool for TLTL model checking. However, there are different techniques [6, 22] using TLTL for the diagnosis of reactive systems and runtime verification. The problem is less severe in the case of branching-time timed logics, where TCTL model checking is PSPACE complete [4, 1] (whereas CTL model checking is possible in polynomial time). In contrast to TLTL model checking, there are industrial tools for TCTL model checking (KRONOS and UPPAAL) used successfully.

In our previous work [10], an approach is proposed to reduce TCTL model checking to CTL model checking. This approach is implemented and tested using the SMV tool. Another similar work can be found in [8], where the model checking is based on the on-the-fly exploration of a simulation graph. The simulation graph is the graph reachable, generated from the region graph [1] and from an initial region. A region is a set of states with the same location and a convex set of clock valuations. This forward-reachability approach is used in tools such as KRONOS [15] and UPPAAL [23]. Thus, because the nodes in the simulation graph are region sets and only discrete transitions are explicit, while time passes implicitly inside the nodes, the simulation graph is much smaller than the region graph. The simulation graph is used to solve the model checking problem for a proposed automata-based branching-time temporal logic ($TECTL^*_\exists$). The on-the-fly model checking procedure consists in solving the emptiness problem, that is, in checking whether an automaton (the automaton product of the system automaton and the property automaton) has an infinite execution sequence that satisfies a given acceptance condition. In our work, the property automaton capturing the temporal constraints, is automatically generated from the TLTL specification. Our quotient graph is produced directly from the initial automaton of timed system specification, which resembles the simulation graph, without passing by the region graph. On the other hand, as it has been shown in [8], the simulation graph preserves only linear-time properties and it is used in practice mainly for reachability properties. Moreover, symbolic states in the simulation graph are not necessarily disjoint, so that this graph can be much larger than the quotient graph. The quotient graph is coarser than the initial automaton but finer, and therefore bigger, than the initial graph. Another algorithm that also combines the on-the-fly and the symbolic approaches has been proposed in [29]. In that work, a symbolic graph is dynamically constructed by the verification procedure, according to the formula (specified in an extended temporal logic of μ-Calculus) to be checked. A similar reduction for a derivate of dense time TCTL (TCTL with freeze quantifiers [3]) is given in [18]. This approach augments the region graph used in [1] by a new atomic proposition and new transitions to handle the reset quantifier. Another related work can be found in [11], where verification is performed by translating TCTL (interpreted over discrete time) into CTL by adding an additional specification clock to the model. So, to model-check the augmented model, the CTL logic is extended, and thus the model-checker, too.

The closest work to ours for the time abstraction based on equivalence can be found in [32]. Where the algorithm in [7] for minimal-model generation (which is an enhancement of the algorithm of Paige and Tarjan [27] to avoid refining unreachable classes) is adapted to infinite state space of timed automaton. This new algorithm which generates a finite region graph using partitioning, uses decision procedures for computing intersection, set difference and predecessors of classes, and testing whether a class is empty. Also, the TCTL specification is reduced to CTL logic extended with new atomic propositions to deal with the specification constraints. Then, a TCTL model checker has been developed based on techniques of the classic CTL model-checker. The generated region graph has size exponential in the number of clocks and the highest constant used in the definition of timing constraints. The other closest work is in [21], where the authors propose an approach to produce a compact reachability graph from a timed automaton. In this work, a state is defined as a history: execution upto the state. It is defined as a pair (location, timed history) instead of (location, clock valuation). A timed history is a set of pairs (transition, time) upto the location of the state. An execution is defined as the transitions between the states (defined as histories). To generate an infinite state space, the algorithm uses the notion of history equivalence (states with the same untimed histories are merged into an equivalence class). To generate a finite state space, a transition bisimulation technique (the states that have the same future behaviors are further collapsed) is used to produce equivalent classes. The resultant state space is finite and can be used to analyze real-time properties. The authors have implemented this approach and analyzed applications, where the real-time properties to be verified are expressed as timed automata to be composed with the system timed automata. Other techniques are based on abstraction of the constraints specified in the system and in the property, using the framework of predicate abstractions as abstract interpretation [9, 25].

The rest of the paper is organized as follows. Section 2 presents a background about Rewriting Logic and Maude LTL Model-Checker. In Section 3, we present the semantics based on Rewriting Logic for specification of complex real-time systems and their semantics based on the formalism of timed automata. Our approach for transformation of the TLTL specifications to LTL specifications is presented in Section 4. In Section 5, we present our method for generating finite bi-similar graphs of the complex timed systems. In Section 6, we explain how to use these graphs for Maude LTL model checking and how the results can be projected back to original complex timed systems. Complexity and implementation results of our approach are presented in Section 7. At the end, a conclusion is given.

2. Rewriting Logic and Maude LTL Model-Checker

Maude specifications are executable logical theories in Rewriting Logic [12, 13], a logic that is a flexible logical framework for expressing a very wide range of concurrency models and distributed systems.

A term is constructed by function and constant symbols. Each term belongs to one or several sorts. Equations specify equivalent terms. Rewriting rules specify how to transform a term into another. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ consists of equations and rewriting rules for terms. If a rewrite theory does not contain any rewriting rules, we also call it an equational theory (Σ, E).

In Rewriting Logic, function and constant symbols are declared by the keyword `op`. Sorts are declared by the keyword `sort`. Equations are specified by `eq lhs = rhs`; conditional equations are specified by `ceq lhs = rhs if cond`. Similarly, rewriting rules and conditional rewriting rules are defined by `rl [l] : lhs => rhs` and `crl [l] : lhs => rhs if cond` respectively, where `l` is the label of the rule. The left-hand side of equations and rewriting rules allows pattern matching. Since there may be several ways to match a term, applying a rewriting rule to a given term may yield multiple results. All results obtained by any of these applications are admissible in Rewriting Logic.

Two terms are equivalent if they can be reduced to the same normal form by the equations of a rewrite theory. Equations therefore define equivalence classes of terms. For any term t, we write $[t]$ for its equivalence class. Let $\mathcal{R}$ be a rewrite theory and t, t' two terms in $\mathcal{R}$. We write

$$\mathcal{R} \vdash_l [t] \rightarrow [t']$$

if there is a rule labeled l in $\mathcal{R}$ that rewrites $[t]$ to $[t']$.

In Rewriting Logic, there is a universal theory $\mathcal{U}$ such that any rewrite theory $\mathcal{R}$ and a term t can be presented as meta-level terms $\overline{\mathcal{R}}$ and $\overline{t}$ in $\mathcal{U}$ respectively. Furthermore, we have

$$\mathcal{R} \vdash_l [t] \rightarrow [t'] \Leftrightarrow \mathcal{U} \vdash_{l,n} [\overline{\mathcal{R}}, \overline{t}] \rightarrow [\overline{\mathcal{R}}, \overline{t'}]$$

if t' is the n-th result obtained by applying the rewriting rule labeled l to t. By the universal theory $\mathcal{U}$, we can manipulate meta-level terms at object level. We call the feature that can represent meta-level objects at object level as reflection. We have used this feature during the implementation of our approach. The system Maude has metalevel operators for moving between reflection levels as `upModule`, `upTerm`, `downTerm`, and others. Other operators are used to act on metalevel terms as for parsing (`metaParse`) and pretty-printing (`metaPrettyPrint`) terms.

Since no domain-specific model of concurrency is built into the logic, the range of applications that can be naturally specified is indeed very wide. Another advantage of Maude as the system specification language is that integration of model checking with theorem proving techniques becomes quite seamless. The same rewrite theory $\mathcal{R} = (\Sigma, E, R)$ can be the input to the LTL model checker and to several other proving tools in the Maude environment [17].

Thus, a Maude module is a rewrite theory $\mathcal{R} = (\Sigma, E, R)$. Fixing a distinguished sort *State*, the initial model $\mathcal{T}_{\mathcal{R}}$ of the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has an underlying Kripke structure $\mathcal{K}(\mathcal{R}, State)$ given by the total binary relation extending its one-step sequential rewrites. In the framework, the Kripke structure is specified as a rewrite theory $\mathcal{K}$. The states are equivalence classes of terms defined in $\mathcal{K}$. The transitions of the Kripke structure correspond to rewriting rules in $\mathcal{K}$. Since the Kripke structure is specified as a rewrite theory and system configurations as equivalence classes of terms, the universal theory $\mathcal{U}$ can be used to explore successors of the current system configuration. To the initial algebra of states $T_{\Sigma/E}$ we can likewise associate equationally-defined computable state predicates as atomic predicates for such a Kripke structure. In this way we obtain a language of LTL properties of the rewrite theory $\mathcal{R}$.

Maude supports on-the-fly LTL model checking [17] for initial states $[t]$, say of sort *State*, of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ such that the set $\{[t'] \in T_{\Sigma/E} | \mathcal{R} \vdash_l [t] \rightarrow [t']\}$, of all states reachable from $[t]$ is finite. The rewrite theory $\mathcal{R}$ should satisfy reasonable executability requirements, such as the confluence and termination of the equations E and coherence of the rules R relative to E.

In Maude the rewrite theory $\mathcal{R}$ is specified as a module, say `M`. Then, given an initial state, say `init` of sort $\texttt{State}_{\texttt{M}}$, we can model check different LTL properties beginning at this initial state by doing the following [17]:

- defining a new module, say `CHECK-M`, that includes the modules `M` and the predefined module `MODEL-CHECKER` as submodules;
- giving a subsort declaration,

  ```
  subsort StateM < State .
  ```

  where `State` is one of the key sorts in the module `MODEL-CHECKER`;
- defining the syntax of the state predicates we wish to use by means of constants and operators of sort `Prop`, a subsort of the sort `Formula` (i.e., LTL formulas) in the module `MODEL-CHECKER`; we can define parameterless state predicates as constants of sort `Prop`, and parameterized state predicates by operators from the sorts of their parameters to the `Prop` sort.
- defining the semantics of the state predicates by means of equations.

Once the semantics of each of the state predicates has been defined, we are then ready, given an initial state `init`, to model check any LTL formula, say `form`, involving such predicates. We do so by evaluating in Maude, the expression `init |= form .` Two things can then happen: if the property `form` holds, then we get the result `true`; if it doesn't, we get a counterexample expressed as a finite path followed by a cycle.

3. Complex Real-time Systems Specification

A complex real-time system can be the composition of many timed sub-systems called processes. The semantics of each process will be represented by a timed automaton. The processes can communicate by sending and receiving messages via channels. This mechanism of communication will be used as synchronization between the different processes to compute their composition. The following is the overall system specification semantics.

```
op system_:'prop_;'chan_;_|=_. :
Token NeTokenList
   NeTokenList GProcesses Bubble ->
   GTProblem .
```

The first argument is the system identifier. A list of atomic propositions is the second argument. The third argument is a list of channels identifiers. The fourth argument will represent the semantics of system processes. The last argument is for the semantics of the TLTL formula.

The semantics of a process is defined with the following operator. It has an identifier, a list of clock variables, a list of states, an initial state and a list of transitions.

```
op process_:'clocks_state_init_trans_; :
Token
   NeTokenList Bubble Token Bubble ->
   GProcess .
```

Clocks are real-valued variables increasing uniformly with time. Several independent clocks may be defined for the same process. A state can have an invariant formula and a list of atomic propositions holding at this state.

```
op _:_{_} : Token NeTokenList
NeTokenList -> GTState .
```

A transition is between two states (source and target, first and second argument, respectively). The transition is conditioned by a temporal constraint. As actions, a transition can reset clocks and it can send and receive messages via the specified channels. This mechanism of sending and receiving messages will be used for synchronization.

```
op _->_:_{_}{_} : Token Token
NeTokenList NeTokenList
   NeTokenList -> GTTransition .
```

A temporal constraint has the following semantics.

```
ops True False : -> TConstraint .
op _=_ : Token Time -> TConstraint .
op _>_ : Token Time -> TConstraint .
op _>=_ : Token Time -> TConstraint .
op _<_ : Token Time -> TConstraint .
op _<=_ : Token Time -> TConstraint .
op _/\_ : TConstraint TConstraint ->
          TConstraint [id: True] .
op _\/_ : TConstraint TConstraint ->
          TConstraint [id: False] .
```

The following is an example, which has two atomic propositions, `p` and `r`. One communication channel `C`. This complex real-time system is composed of one process with three states: `a`, `b`, and `c` and three transitions. The real-time specification is followed by specification of a TLTL property which can be omitted and given separately to allow the specification of different TLTL properties.

```
system Example :
     prop p r ;
     chan C ;
     process P :
        clocks x
        state
          a  :          { }
          b  : x <= 1 { p }
          c  :          { r }
        init a
        trans
          a -> b : x = 1  { x } { }
          a -> c : x >= 2 { }   { !C }
          b -> c : x = 1  { }   { } ;
     |= True U { >= 2 } r
```

The semantics of a complex real-time system is represented by timed automata [2] which extend the automata formalism by adding clocks. These semantics will be defined by the following two main operators. The first partial operator (`getSystem`) is called when the parsing of a complete specification is succeeded (the result of the parsing is a `Term`), to generate needed semantics as system identifier, atomic propositions, and channels identifiers. This operator calls the second (`solveProcesses`) to construct the timed automaton for the whole complex real-time system by composition of the processes timed automata.

```
op getSystem : Term ~> TimedSystem .
op solveProcesses : Term ~>
TimedAutomaton .
```

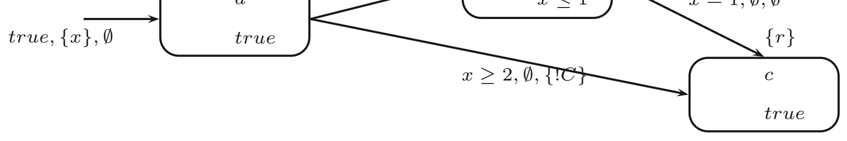
A timed automaton $\mathcal{A}$ is a tuple $\langle \mathcal{Q}, \mathcal{X}, \Sigma, \mathcal{E}, \mathcal{L}^1, \mathcal{L}^2, \mathcal{I} \rangle$, where:

- $\mathcal{Q}$ is a finite set of locations. We denote by $q_0 \in \mathcal{Q}$ the initial location.
- $\mathcal{X}$ is a finite set of clocks. A valuation v is a function that assigns a non negative real-value $v(x) \in R^+$ to each clock $x \in \mathcal{X}$. The valuation $v[X := \delta]$ assigns the value δ to all clocks in the set X. The set of valuations is denoted $\mathcal{V}_\mathcal{X}$. For $\delta \in R^+$, $v + \delta$ denotes the valuation v' such that $v'(x) = v(x) + \delta$ for all $x \in \mathcal{X}$.
- Σ is a finite set of labels (message channels).
- $\mathcal{E}$ is a finite set of edges. Each edge $e \in \mathcal{E}$ is a tuple $\langle q, \theta, X, \sigma, q' \rangle$ where
  - $q, q' \in Q$ are the source and the target locations respectively,
  - $\theta \in \Theta$ is an associated clock constraint which governs the triggering of the transition. It is called its enabling condition or its guard. We denote the set of constraints over $\mathcal{X}$ by Θ. A constraint is defined as a conjunction of atoms of the form $x \sim c$, where $x \in \mathcal{X}$, $\sim \in \{=, >, \geq, <, \leq\}$ and c is a natural constant.
  - $X \subseteq \mathcal{X}$ is the set of clocks to be reset after making this transition.
  - σ is a subset of synchronization events from the set Σ. A synchronization event is the combination of a channel name preceded by the symbol `!` to indicate send event, or `?` for receive event.
- $\mathcal{L}^1 : \mathcal{Q} \rightarrow 2^{AP}$ is a function that associates to each location a set of atomic propositions from the set AP.
- $\mathcal{L}^2 : \mathcal{E} \rightarrow 2^\Sigma$ is a function that associates to each edge a set of synchronization events from the set Σ. We have two kinds of synchronization events (send events and receive events).
- $\mathcal{I}$ is a function that associates a condition $\mathcal{I}(q) \in \Theta$ to every location $q \in \mathcal{Q}$ called the invariant of q.

Figure 1 shows an example of a timed automaton representing the semantics of the example above. $AP = \{p, r\}$ and $\mathcal{Q} = \{a, b, c\}$. A state of $\mathcal{A}$ is a pair $\langle q, v \rangle \in \mathcal{Q} \times \mathcal{V}_\mathcal{X}$ such that v satisfies $\mathcal{I}(q)$. The initial state is the pair $\langle q_0, v_0 \rangle$ such that $v_0(x) = 0$ for all $x \in \mathcal{X}$. Let $\mathcal{S}$ denote the set of states of $\mathcal{A}$. We will refer to $\mathcal{L}^1(s)$ by $\mathcal{L}^1(q)$, for all $s \in \mathcal{S}$, where $s = \langle q, v \rangle$. The set $\mathcal{S}$ can be partitioned to zones (symbolic states). A zone $z = (q, \mathcal{V}_z)$ is a set of states from $\mathcal{S}$ which are associated with the same discrete state $q \in \mathcal{Q}$ and a convex set of valuations $\mathcal{V}_z = \{v \mid \exists \langle q, v \rangle \in \mathcal{S}\}$. The state of a timed system can be changed through an edge that changes the location and resets some of the clocks (discrete transition), or by letting time pass without changing the location (time transition).

Let $e = \langle q, \theta, X, \sigma, q' \rangle \in \mathcal{E}$ be an edge. The state $\langle q, v \rangle$ has a discrete transition to $\langle q', v' \rangle$, denoted $\langle q, v \rangle \xrightarrow{d} \langle q', v' \rangle$, if v satisfies θ and $v' = v[X := 0]$ (we should note that the set of valuations respecting θ is always in the set of valuations respecting $\mathcal{I}(q)$). Let $\delta \in R^+$. The state $\langle q, v \rangle$ has a time transition to $\langle q, v + \delta \rangle$, denoted $\langle q, v \rangle \xrightarrow{\tau} \langle q, v + \delta \rangle$, if for all $\delta' \leq \delta$, $v + \delta'$ satisfies the invariant $\mathcal{I}(q)$.

For timed automata, we label each discrete transition with a label (or an action) a. The label is composed of a temporal constraint to execute the transition when it holds and an action of resetting clocks. The time transitions



$\{p\}$
$\emptyset$
b
$x \leq 1$
$x = 1, \{x\}, \emptyset$
a
$true$
$true, \{x\}, \emptyset$
$x = 1, \emptyset, \emptyset$
$\{r\}$
$x \geq 2, \emptyset, \{!C\}$
c
$true$

Figure 1. Timed automaton.

have a particular label named τ denoting time elapse which is considered as an internal or hidden action. Let A be the set of actions and $A_\tau = A \cup \{\tau\}$. We note $\mathcal{M} = (\mathcal{S}, A_\tau, \mathcal{T}, s_0, \mathcal{L})$ the labeled transition system of $\mathcal{A}$ (its semantics), $\mathcal{S}$ is the set of reachable states from s_0 with respect to $\mathcal{T}$. $\mathcal{L} : \mathcal{S} \to 2^{AP}$ and $\mathcal{L}(s) = \mathcal{L}^1(q)$, for each $s \in q$. $\mathcal{T} \subseteq \mathcal{S} \times A_\tau \times \mathcal{S}$ the (discrete or time) transition relation and s_0 the initial state. For each label a and each state s, we consider the image set $\mathcal{T}_a(s) = \{s' \in \mathcal{S} \mid (s, a, s') \in \mathcal{T}\}$. We extend this notation for sets of states: $\mathcal{T}_a(B) = \cup\{\mathcal{T}_a(s) \mid s \in B\}$. $\mathcal{T}^{-1}$ denotes the inverse relation.

A run r of $\mathcal{M}$ is an infinite sequence of states and transitions. We denote $\mathcal{R}$ the set of runs of $\mathcal{M}$. A run is divergent if $\sum_{i=0}^{\infty} \delta_i$ (the sum of all delays δ_i on this run) diverges. We denote $\mathcal{R}_\infty$ the set of divergent runs of $\mathcal{M}$. In the following, we will consider timed automata with only divergent runs (if the automaton has non-divergent runs, called also zeno runs, it is possible to restrict the behavior to divergent runs [19]).

A complex real-time system is composed of many processes using the operator `Compose`. Their different timed automata will be composed to construct the timed automaton of the overall complex real-time system.

```
op Compose : TimedAutomaton
TimedAutomaton ->
              TimedAutomaton .
```

The parallel composition of two timed automata ($\|$) is defined as follows. Let $\mathcal{A}_i$ be $\langle\, \mathcal{Q}_i,\ \mathcal{X}_i,\ \Sigma_i,\ \mathcal{E}_i,\ \mathcal{L}^1_i,\ \mathcal{L}^2_i,\ \mathcal{I}_i \rangle$, for $i = 1, 2$. We assume that $\mathcal{Q}_1 \cap \mathcal{Q}_2 = \emptyset$ and $\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$. The product-timed automaton $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2 = \langle\, \mathcal{Q}, \mathcal{X}, \Sigma, \mathcal{E}, \mathcal{L}^1, \mathcal{L}^2, \mathcal{I} \rangle$ is such that: $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2$, $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $\mathcal{I}(\langle\, q_1, q_2 \rangle\,) = \mathcal{I}_1(q_1) \wedge \mathcal{I}_2(q_2)$, $\mathcal{L}^1(\langle\, q_1, q_2 \rangle\,) = \mathcal{L}^1_1(q_1) \cup \mathcal{L}^1_2(q_2)$, $\mathcal{L}^2$ is defined during the composition of edges. The set $\mathcal{E}$ of edges is obtained as follows.

1. for all $\langle\, q_1, \theta_1, X_1, \sigma_1, q'_1 \rangle \in \mathcal{E}_1$ and $\langle\, q_2, \theta_2, X_2, \sigma_2, q'_2 \rangle \in \mathcal{E}_2$:
   - if $\mathit{receive}(\sigma_1) \subseteq \mathit{send}(\sigma_2) \wedge \mathit{receive}(\sigma_2) \subseteq \mathit{send}(\sigma_1)$ then $\mathcal{E}$ includes $\langle\, (q_1, q_2), \theta_1 \wedge \theta_2, X_1 \cup X_2, \mathit{send}(\sigma_1 \cup \sigma_2), (q'_1, q'_2) \rangle$.
   - if $\mathit{receive}(\sigma_1) \subseteq \mathit{send}(\sigma_2) \wedge (\mathit{receive}(\sigma_2) = \emptyset \vee \mathit{receive}(\sigma_2) \not\subseteq \mathit{send}(\sigma_1))$ then $\mathcal{E}$ includes $\langle\, (q_1, q_2), \theta_1, X_1, \mathit{send}(\sigma_1 \cup \sigma_2) \cup \mathit{receive}(\sigma_2) \setminus \mathit{send}(\sigma_1), (q'_1, q_2) \rangle$.
   - if $\mathit{receive}(\sigma_2) \subseteq \mathit{send}(\sigma_1) \wedge (\mathit{receive}(\sigma_1) = \emptyset \vee \mathit{receive}(\sigma_1) \not\subseteq \mathit{send}(\sigma_2))$ then $\mathcal{E}$ includes $\langle\, (q_1, q_2), \theta_2, X_2, \mathit{send}(\sigma_1 \cup \sigma_2) \cup \mathit{receive}(\sigma_1) \setminus \mathit{send}(\sigma_2), (q_1, q'_2) \rangle$.
2. for all $\langle\, q_1, \theta_1, X_1, \sigma_1, q'_1 \rangle \in \mathcal{E}_1$, if $\mathit{channel}(\sigma_1) \notin \Sigma_1 \cap \Sigma_2$ then $\forall q_2 \in \mathcal{Q}_2$, $\mathcal{E}$ includes $\langle\, (q_1, q_2), \theta_1, X_1, \sigma_1, (q'_1, q_2) \rangle$.
3. for all $\langle\, q_2, \theta_2, X_2, \sigma_2, q'_2 \rangle \in \mathcal{E}_2$, if $\mathit{channel}(\sigma_2) \notin \Sigma_1 \cap \Sigma_2$ then $\forall q_1 \in \mathcal{Q}_1$, $\mathcal{E}$ includes $\langle\, (q_1, q_2), \theta_2, X_2, \sigma_2, (q_1, q'_2) \rangle$.

$\mathit{receive}(\sigma)$ and $\mathit{send}(\sigma)$ are used to extract channel names used by receive and send events, respectively. $\mathit{channel}(\sigma)$ returns the name of the communication channel used by the event (receive or send) σ. Note that $\mathit{receive}(\emptyset) = \mathit{send}(\emptyset) = \mathit{channel}(\emptyset) = \emptyset$. A transition waiting reception of messages from transitions in other processes via defined transmission channels, will be composed only with those transitions delivering these messages.

The first case of (1) indicates that the receive events on both transitions are sent mutually by these two transitions. The second case of (1) indicates that the receive events of the first transition are all sent by the second transition, but the receive events of the second are empty or not all sent by the first transition. The third condition of (1) is the symmetry of the second. The cases (2) and (3) are for automata not sharing communication channels. The composition of two automata without synchronization events equals to their Cartesian product. This composition of timed automata representing different processes of a system, terminates by producing one timed automaton without synchronization events. The produced system timed automaton will be composed with the property timed automaton generated from the TLTL specification.

4. Transformation of TLTL Specifications

Many important properties of complex timed systems find a natural expression in the real-time temporal logic TLTL, which extends the linear time logic LTL [16, 3, 28]. This extension either augments temporal operators with time bounds, or uses reset quantifiers. We use a version of TLTL with time bounds.

Maude LTL model checker [17] provides the common LTL operators U, W, R, $\Box$, $\diamond$ and O, written as until, until weak, releases, always, eventually, and next in addition to the well known set of Boolean operators : $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$. The formulas φ of the linear temporal logic LTL are defined inductively by the grammar:

$\varphi ::= true \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi U \varphi \mid \varphi W \varphi \mid \varphi R \varphi \mid \Box\varphi \mid \diamond\varphi \mid O\varphi$.

Where $p \in AP$ is an atomic proposition. The LTL semantics over a labeled transition system, are defined using the satisfaction relation, denoted by $s \models \varphi$, on the syntax of φ:

- $s \models true$
- $s \models p$ iff $p \in \mathcal{L}(s)$
- $s \models \neg\varphi$ iff $s \not\models \varphi$
- $s \models \varphi' \wedge \varphi''$ iff $s \models \varphi' \wedge s \models \varphi''$
- $s \models \varphi' U \varphi''$ iff $\forall r \in \mathcal{R}_\infty$ and $r(0) = s, \exists i$ and $r(i) \models \varphi''$ and $\forall j < i.r(j) \models \varphi'$
- $s \models \varphi' W \varphi''$ iff $s \models \varphi' U \varphi'' \vee s \models \Box\varphi'$
- $s \models \varphi' R \varphi''$ iff $s \models \varphi'' W \varphi'$
- $s \models \Box\varphi$ iff $\forall r \in \mathcal{R}_\infty$ and $s = r(0), \forall i \geq 0.r(i) \models \varphi$
- $s \models \diamond\varphi$ iff $s \models true U \varphi$
- $s \models O\varphi$ iff $\forall r \in \mathcal{R}_\infty$ and $r(0) = s, r(1) \models \varphi$

The added TLTL (Timed LTL) operators are: $O_{\sim c}\varphi$ states that the next occurrence of φ is within the time bounds $\sim c$. $\varphi U_{\sim c}\psi$ states that φ is true until the next occurrence of ψ, and that this occurrence of ψ is within the time bounds $\sim c$. $\Box_{\sim c}\varphi$ states that φ must always be true within the time bounds $\sim c$. $\diamond_{\sim c}\varphi$ states that φ must be true at some point within the time bounds $\sim c$, where $\sim \in \{=, >, \geq, <, \leq\}$ and $c \in N$ (N is the set of natural numbers).

The formulas ψ of the timed linear temporal logic TLTL are defined inductively by the grammar:

$$\psi ::= \varphi \mid O_{\sim c}\psi \mid \psi U_{\sim c}\psi \mid \Box_{\sim c}\psi \mid \diamond_{\sim c}\psi.$$

Where φ is an LTL formula. The formulas of TLTL are interpreted over the set of states of a timed automaton represented by a transition system $\mathcal{M}$. Let $\langle q, v\rangle \in \mathcal{S}$ be a state reachable in $\mathcal{M}$ and let a TLTL-formula ψ. The satisfaction relation, denoted by $\langle q, v\rangle \models_{\mathcal{M}} \psi$, is defined inductively on the syntax of ψ:

- $\langle q, v\rangle \models_{\mathcal{M}} \varphi$ Its semantics is defined using the semantics of the logic LTL.
- $\langle q, v\rangle \models_{\mathcal{M}} O_{\sim c}\psi$ iff $\forall r \in \mathcal{R}_\infty$ and $r(0) = \langle q, v\rangle, \exists i.\Sigma_{j\leq i}\delta_j \sim c$ and $r(i) \models_{\mathcal{M}} \psi$ and $\forall j < i.r(j) \not\models_{\mathcal{M}} \psi$
- $\langle q, v\rangle \models_{\mathcal{M}} \psi' U_{\sim c}\psi''$ iff $\forall r \in \mathcal{R}_\infty$ and $r(0) = \langle q, v\rangle, \exists i.\Sigma_{j\leq i}\delta_j \sim c$ and $r(i) \models_{\mathcal{M}} \psi''$ and $\forall j < i.r(j) \models_{\mathcal{M}} \psi'$ if $\sim \in \{=, \geq, >\}$ then $r(j) \not\models_{\mathcal{M}} \psi''$
- $\langle q, v\rangle \models_{\mathcal{M}} \diamond_{\sim c}\psi$ iff $\langle q, v\rangle \models_{\mathcal{M}} True U_{\sim c}\psi$
- $\langle q, v\rangle \models_{\mathcal{M}} \Box_{\sim c}\psi$ iff $\forall r \in \mathcal{R}_\infty$ and $r(0) = \langle q, v\rangle, \exists i.\Sigma_{j\leq i}\delta_j \sim c$ and $\forall j \sim i, r(j) \models_{\mathcal{M}} \psi$

We augmented the LTL syntax used by Maude LTL model-checker by the following operators to specify TLTL properties.

```
op O'{=_}_ : Time Formula -> Formula .
op O'{>_}_ : Time Formula -> Formula .
op O'{>=_}_ : Time Formula -> Formula .
op O'{<_}_ : Time Formula -> Formula .
op O'{<=_}_ : Time Formula -> Formula .
op _U'{=_}_ : Formula Time Formula ->
Formula .
op _U'{>_}_ : Formula Time Formula ->
Formula .
op _U'{>=_}_ : Formula Time Formula ->
Formula .
op _U'{<_}_ : Formula Time Formula ->
Formula .
op _U'{<=_}_ : Formula Time Formula ->
Formula .
op <>'{=_}_ : Time Formula -> Formula .
op <>'{>_}_ : Time Formula -> Formula .
op <>'{>=_}_ : Time Formula -> Formula .
op <>'{<_}_ : Time Formula -> Formula .
op <>'{<=_}_ : Time Formula -> Formula .
op []'{=_}_ : Time Formula -> Formula .
op []'{>_}_ : Time Formula -> Formula .
op []'{>=_}_ : Time Formula -> Formula .
op []'{<_}_ : Time Formula -> Formula .
op []'{<=_}_ : Time Formula -> Formula .
```

Our objective is to transform a TLTL formula ψ to an LTL formula φ. Any TLTL formula ψ will introduce a new set of specification clocks $\mathcal{X}_{\psi}$. This set of specification clocks does not control the behavior of any system under consideration. The transformation process reduces the TLTL formula ψ recursively by decomposing ψ. At the end, it generates an equivalent LTL formula φ and a timed automaton $\mathcal{A}_{\psi}$, capturing the timed behavior specified in the TLTL formula ψ. If the formula does not contain temporal constraint (it is already an LTL formula), the transformation process returns this formula and an empty timed automaton.

On the other hand, if the TLTL formula contains temporal constraints, these can be of one of the forms presented below. The constructed timed automaton $\mathcal{A}_{\psi}$, is almost the same for all the forms which have a set of one clock variable $\mathcal{X}_{\psi} = \{z\}$, two discrete states $Q_{\psi} = \{q_0^{\psi}, q_1^{\psi}\}$ with invariants $z \leq c$ and *true* respectively (these two discrete states are labeled according to the formula), and one edge $\mathcal{E}_{\psi} = \{(q_0^{\psi}, z = c, \emptyset \text{ } or \text{ } \{z\}, \emptyset, q_1^{\psi})\}$. This timed automaton will be composed (using the operator `TCompose` of linear composition) with the product of the two timed automata $\mathcal{A}_{\psi'}$ and $\mathcal{A}_{\psi''}$ constructed by the recursive call to the functions `Transform`(ψ', Pr) and `Transform`(ψ'', Pr) respectively. The second argument is used to label one of the two discrete states ($Q_{\psi} = \{q_0^{\psi}, q_1^{\psi}\}$). The following is an example for transformation of the formulas of the form $\varphi U_{\geq c} \psi$ and $\varphi U_{\leq c} \psi$ respectively.

```
op Transform : Formula Id ->
AutomatonFormula .
ceq Transform(F U { >= T } F', Pr) =
 {
  TCompose(getCapAutomaton
  (AF1:AutomatonFormula),
   TCompose(
    ("U" : { "z" } ; { "q1" } ;
     { ("q1" : "z" <= T { empty }) ,
       ("q2" : True { Pr }) } ;
     { ("q1" -> "q2" : "z" = T { empty }
     { empty }) }
    ),
    getCapAutomaton
    (AF2:AutomatonFormula))) ;
  (getTrsFormula
  (AF1:AutomatonFormula) /\
   ~ getTrsFormula
   (AF2:AutomatonFormula)) U
  (getTrsFormula
  (AF2:AutomatonFormula) /\
   Prop(qid(Pr)))
 }
 if AF1:AutomatonFormula :=
 Transform(F, Pr + "1") /\
    AF2:AutomatonFormula :=
    Transform(F', Pr + "2") .
ceq Transform(F U { <= T } F', Pr) =
 {
  TCompose(getCapAutomaton
  (AF1:AutomatonFormula),
   TCompose(
    ("U" : { "z" } ; { "q1" } ;
     { ("q1" : "z" <= T { Pr }) ,
       ("q2" : True { empty }) } ;
     { ("q1" -> "q2" : "z" = T { empty }
     { empty }) }
    ),
    getCapAutomaton
    (AF2:AutomatonFormula))) ;
   (getTrsFormula
   (AF1:AutomatonFormula) /\
    Prop(qid(Pr))) U
   (getTrsFormula
   (AF2:AutomatonFormula)) }
 if AF1:AutomatonFormula :=
 Transform(F, Pr + "1") /\
    AF2:AutomatonFormula :=
    Transform(F', Pr + "2") .
```

The transformations for the other TLTL operators are coded almost by the same manner. The formulas using the timed operators O and $\diamond$ are converted to equivalent formulas before transformation as follows.

```
*** Operator timed O
eq Transform(O { = T } F, Pr) =
   Transform(([] { < T } (~ F)) /\
             (True U { <= T } F), Pr) .
eq Transform(O { > T } F, Pr) =
   Transform((~ F) /\ ((~ F) U { > T }
   F), Pr) .
eq Transform(O { >= T } F, Pr) =
   Transform((~ F) /\ ((~ F) U { >= T }
   F), Pr) .
eq Transform(O { < T } F, Pr) =
   Transform((~ F) U { < T } F, Pr) .
eq Transform(O { <= T } F, Pr) =
   Transform((~ F) U { <= T } F, Pr) .
```

```
*** Operator timed <>
eq Transform(<> { > T } F, Pr) =
   Transform(True U { = T } F, Pr ) .
eq Transform(<> { > T } F, Pr) =
   Transform(True U { > T } F, Pr ) .
eq Transform(<> { >= T } F, Pr) =
   Transform(True U { >= T } F, Pr ) .
eq Transform(<> { < T } F, Pr) =
   Transform(True U { < T } F, Pr ) .
eq Transform(<> { <= T } F, Pr ) =
   Transform(True U { <= T } F, Pr ) .
```

Example 1. *For the TLTL formula $\psi = \diamond_{\geq 2} r$ (which is equivalent to $trueU_{\geq 2}r$), the function* `Transform` *generates the equivalent LTL formula φ and the timed automaton $\mathcal{A}_\psi$ shown in Figure 2.*

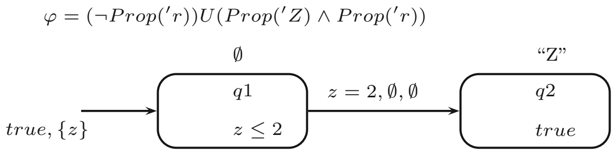

$\varphi = (\neg Prop('r))U(Prop('Z) \wedge Prop('r))$
$\emptyset$
$q1$
$z \leq 2$
$z = 2, \emptyset, \emptyset$
"Z"
$q2$
$true$
$true, \{z\}$

Figure 2. Generated timed automaton $\mathcal{A}$ with LTL formula φ for $\psi = \diamond_{\geq 2} r$.

The following is the result of the Maude command.

```
reduce in VRTS : Transform(<>{>= 2}Prop
('r), "Z") .
rewrites: 13 in -157072442571ms cpu
(0ms real)
          (~ rewrites/second)
result AutomatonFormula: {"U" :{"z"};
{"q1"};
    {("q1" : "z" <= 2{empty}),"q2" :
    True{"Z"}};{"q1" -> "q2" : "z" =
    2{empty}{empty}} ;
    (~ Prop('r)) U (Prop('Z) /\
    Prop('r))}
```

Example 2. *This is another example. Let the TLTL formula $\psi = \diamond_{\leq 2}(a \wedge \diamond_{\leq 1} b)$. The function* `Transform` *generates the equivalent LTL formula φ and the timed automaton $\mathcal{A}_\psi$ shown in Figure 3.*

```
red Transform(<> { <= 2 } (Prop('a) /\
(<> { <= 1 }
    Prop('b))), "Z") .
reduce in VRTS : Transform(<>{<= 2}((<>
{<= 1}Prop('b))
    /\ Prop('a)), "Z") .
rewrites: 79 in -3902117295ms cpu
(0ms real)
    (~ rewrites/second)
result AutomatonFormula: {"UU" :{"z"};
{"q1q1"};
    {("q1q1" : "z" <= 2{"Z","Z2111"}),
    ("q1q2" : "z" <= 2{"Z"}),("q2q1" :
    "z" <= 3{"Z2111"}),"q2q2" :
    True{empty}};
    {"q1q1" -> "q2q1" : "z" =
    2{empty}{empty},
    "q2q1" -> "q2q2" : "z" =
    3{empty}{empty}} ;
    (Prop('Z)) U ((((Prop('Z2111)) U
    (Prop('b)))
    /\ Prop('a)))}
```

The constructed timed automaton $\mathcal{A}_\psi$ will be composed with the original timed automaton $\mathcal{A}$ ($\mathcal{A} \parallel \mathcal{A}_\psi$). The operator `Transform` is using the operator `TCompose` to realize a linear composition (denoted by $\oplus$) of two timed automata. It is defined to compose the constructed timed automata $\mathcal{A}_{\psi'}$ and $\mathcal{A}_{\psi''}$, where ψ' and ψ'' are sub-formulas of ψ, to get only one timed automaton $\mathcal{A}_\psi$ with one clock variable z. Its difference from the operator $\parallel$ is in the construction of the set of edges $\mathcal{E}$, which is obtained as follows. Let $e_i \in \mathcal{E}_i$ of the form $\langle q_i, z_i = c_i, \emptyset, \emptyset, q_i' \rangle$, for $i = 1, 2$. Then, if $c_1 < c_2$, we add $e = \langle (q_1, q_2), z = c_1, \emptyset, \emptyset, (q_1', q_2) \rangle$ to $\mathcal{E}$, where $\mathcal{I}(\langle q_1, q_2 \rangle) = z \leq c_1$. We replace e_2 in $\mathcal{E}_2$ by $\langle q_2, z_2 = c_2 - c_1, \emptyset, \emptyset, q_2' \rangle$ and we remove e_1 from $\mathcal{E}_1$. Else, if $c_2 < c_1$,

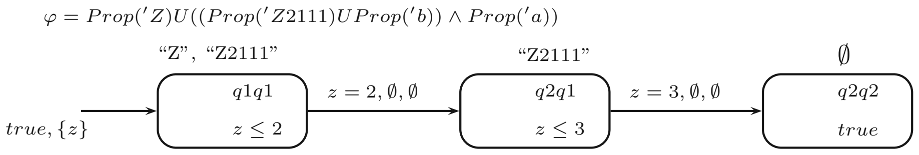

$\varphi = Prop('Z)U((Prop('Z2111)UProp('b)) \wedge Prop('a))$
"Z", "Z2111"
$q1q1$
$z \leq 2$
$z = 2, \emptyset, \emptyset$
"Z2111"
$q2q1$
$z \leq 3$
$z = 3, \emptyset, \emptyset$
$\emptyset$
$q2q2$
$true$
$true, \{z\}$

Figure 3. Generated timed automaton $\mathcal{A}$ with LTL formula φ for $\psi = \diamond_{\leq 2}(a \wedge \diamond_{\leq 1} b)$.

we add $e = \langle (q_1, q_2), z = c_2, \emptyset, \emptyset, (q_1, q_2')\rangle$ to $\mathcal{E}$, where $\mathcal{I}(\langle q_1, q_2\rangle) = z \leq c_2$. We replace e_1 in $\mathcal{E}_1$ by $\langle q_1, z_1 = c_1 - c_2, \emptyset, \emptyset, q_1'\rangle$ and we remove e_2 from $\mathcal{E}_2$. Else, $e = \langle (q_1, q_2), z = c_1, \emptyset, \emptyset, (q_1', q_2')\rangle$, where $\mathcal{I}(\langle q_1, q_2\rangle) = z \leq c_1$ and we remove e_1 from $\mathcal{E}_1$, e_2 from $\mathcal{E}_2$.

This process will continue until $\mathcal{E}_i = \emptyset$, for $i = 1, 2$ or one of the following two cases is satisfied. In the case where $\mathcal{E}_1 = \emptyset$ and $\mathcal{E}_2 = \{\langle q_2, z_2 = c_2, \emptyset, \emptyset, q_2'\rangle\}$, we assume that $q_1 \in \mathcal{Q}_1$ is the discrete state without outgoing edge. Then, we add the edge $e = \langle (q_1, q_2), z = c_2, \emptyset, \emptyset, (q_1, q_2')\rangle$ to $\mathcal{E}$, where $\mathcal{I}(\langle q_1, q_2\rangle) = z \leq c_2$. In the other case where $\mathcal{E}_2 = \emptyset$ and $\mathcal{E}_1 = \{\langle q_1, z_1 = c_1, \emptyset, \emptyset, q_1'\rangle\}$, we assume that $q_2 \in \mathcal{Q}_2$ is the discrete state without outgoing edge. Then, we add the edge $e = \langle (q_1, q_2), z = c_1, \emptyset, \emptyset, (q_1', q_2)\rangle$ to $\mathcal{E}$, where $\mathcal{I}(\langle q_1, q_2\rangle) = z \leq c_1$. At the end, if there are discrete states in the produced timed automaton, without ingoing and outgoing edges, they will be removed from the set of discrete states $\mathcal{Q}$.

Theorem 1. *Let $\mathcal{M}$ be the transition system of the timed automaton $\mathcal{A}$ modeling a real-time system. If the function* `Transform` *produces an LTL formula φ and a timed automaton $\mathcal{A}_\psi$ from a TLTL formula ψ, and if $\langle q, v\rangle \models_\mathcal{M} \psi$ (the state $\langle q, v\rangle$ satisfies the TLTL formula ψ), and $\langle q, v\rangle \models_{\mathcal{M}^+} \varphi$ (the state $\langle q, v\rangle$ satisfies the LTL formula φ ($\mathcal{M}^+$ is the transition system of $\mathcal{A}^+ = \mathcal{A} \parallel \mathcal{A}_\psi$)). Then, $\langle q, v\rangle \models_\mathcal{M} \psi \Leftrightarrow \langle q, v\rangle \models_{\mathcal{M}^+} \varphi$*

Proof. *The proof proceeds by induction on the structure of ψ. The basis case where ψ is of the form φ (an LTL formula) is immediate. In this basis case $\mathcal{A}_\psi = \emptyset$; and $\mathcal{M}^+ = \mathcal{M}$, this means $\langle q, v\rangle \models_\mathcal{M} \psi \Leftrightarrow \langle q, v\rangle \models_{\mathcal{M}^+} \varphi$.*

We will prove the case $\psi = \psi' U_{\geq c} \psi''$ and the other cases can be proved by the same way. Consider a state $\langle q, v\rangle$ in $\mathcal{M}$. Assume that $\langle q, v\rangle \models_\mathcal{M} \psi$. Then, by the semantics of TLTL, for any run $r = \langle q_0, v_0\rangle, \langle q_1, v_1\rangle, \cdots, \langle q_i, v_i\rangle, \cdots \in \mathcal{R}_\infty$ with $\langle q_0, v_0\rangle = \langle q, v\rangle$, where $i \geq 0$ such that $\Sigma_{k=0}^{i} \delta_k \geq c$ and $\langle q_i, v_i\rangle \models_\mathcal{M} \psi''$, and for all $0 \leq j < i$ we have $\langle q_j, v_j\rangle \models_\mathcal{M} \psi' \wedge \neg\psi''$.

If the timed automaton $\mathcal{A}_\psi = \mathcal{A}_{\psi'} \oplus \{\mathcal{Q} = \{q_0^\psi, q_1^\psi\}, \mathcal{X} = \{z\}, \Sigma = \emptyset, e^\psi = (q_0^\psi, z = c, \emptyset, \emptyset, q_1^\psi), \mathcal{L}^1 = \{\mathcal{L}^1(q_0^\psi) = True, \mathcal{L}^1(q_1^\psi) =$ "Z"$\}, \mathcal{L}^2 = \emptyset, \mathcal{I} = \{\mathcal{I}(q_0^\psi) = z \leq c, \mathcal{I}(q_1^\psi) = True\}\} \oplus \mathcal{A}_{\psi''}$ and the TLT formula $\varphi = \varphi' \wedge \neg\varphi'' U \varphi'' \wedge$"Z" are the results of `Transform(`$\psi = \psi' U_{\sim c} \psi''$`)`*, where $\oplus$ is the operator* `TCompose` *as defined before, $\{\mathcal{A}_{\psi'}, \varphi'\}$ and $\{\mathcal{A}_{\psi''}, \varphi''\}$ are the results of* `Transform(`ψ'`)` *and* `Transform (`ψ''`)`*, respectively.*

By the induction hypothesis, $\langle q_i, v_i\rangle \models_\mathcal{M} \psi'' \Leftrightarrow \langle (q_i, q_{\psi''}), v_i\rangle \models_{\mathcal{M}^+_{\psi''}} \varphi''$ ($\mathcal{M}^+_{\psi''}$ is the model of $\mathcal{A} \parallel \mathcal{A}_{\psi''}$) and $\langle q_j, v_j\rangle \models_\mathcal{M} \psi' \wedge \neg\psi'' \Leftrightarrow \langle (q_j, q_{\psi'}), v_j\rangle \models_{\mathcal{M}^+_{\psi'}} \varphi' \wedge \neg\varphi''$ ($\mathcal{M}^+_{\psi'}$ is the model of $\mathcal{A} \parallel \mathcal{A}_{\psi'}$). It is clear from the parallel composition that $\langle (q_i, q_{\psi'}, q_{\psi''}), v_i\rangle \models_{\mathcal{M}^+_{\psi'\psi''}} \varphi''$ and $\langle (q_j, q_{\psi'}, q_{\psi''}), v_j\rangle \models_{\mathcal{M}^+_{\psi'\psi''}} \varphi'$, where $\mathcal{M}^+_{\psi'\psi''}$ is the model of $\mathcal{A} \parallel (\mathcal{A}_{\psi'} \oplus \mathcal{A}_{\psi''})$ ($\oplus$ is the operator `TCompose` *as defined before).*

If $\mathcal{M}^+$ is the model of $\mathcal{A} \parallel \mathcal{A}_\psi$, then it is clear that $v_i(z) \geq c$ and using the property of parallel composition, we have $\langle (q_i, q_{\psi'}, q_{\psi''}, q_1^\psi), v_i\rangle \models_{\mathcal{M}^+} \varphi'' \wedge$"Z" ($\mathcal{L}^1(q_1^\psi) =$ "Z") and for all $0 \leq j < i$ we have $\langle (q_j, q_{\psi'}, q_{\psi''}, q_0^\psi), v_j\rangle \models_{\mathcal{M}^+} \varphi' \wedge \neg\varphi''$. By the semantics of TLTL, we have $\langle q, v\rangle \models_{\mathcal{M}^+} \varphi' \wedge \neg\varphi'' U \varphi'' \wedge$ "Z".

5. Generating Bi-similar Finite System

The model of a timed automaton is an infinite transition-state system due to dense time. Then, it is not possible to perform a model checking. In this section we present our method that generates a strongly bi-similar finite system based on a defined equivalence where exact delays are abstracted away while information on the discrete changes of the system is retained.

5.1. Strong Bi-simulation

For a labeled transition system $\mathcal{M} = (\mathcal{S}, A_\tau, \mathcal{T}, s_0, \mathcal{L})$, a partition $\wp$ (or equivalence relation on $\mathcal{S}$) of the elements of $\mathcal{S}$ is a set of disjoint blocks $\{B_i \mid i \in N\}$ such that $\cup_{i \in N} B_i = \mathcal{S}$. Let $\wp$ and $\wp'$ be partitions of $\mathcal{S}$. $\wp'$ is a refinement of $\wp$ ($\wp' \sqsubseteq \wp$) if and only if $\forall B' \in \wp' : \exists B \in \wp : (B' \subseteq B)$. Intuitively, two states s_1 and s_2 are bi-similar if for each state s_1' reachable from s_1 by execution of an action $a \in A_\tau$ (see Section

3) there is a state s'_2, reachable from s_2 by execution of the action a such that s'_1 and s'_2 are bi-similar.

Definition 1. *Given a labeled transition system $\mathcal{M} = (\mathcal{S}, A_\tau, \mathcal{T}, s_0, \mathcal{L})$, a binary relation $\wp \subseteq \mathcal{S} \times \mathcal{S}$ is a strong bi-simulation if and only if the following conditions hold $\forall (s_1, s_2) \in \wp$ and $\forall a \in A_\tau$:*

1. $\mathcal{L}(s_1) = \mathcal{L}(s_2)$,
2. $\forall s_3 (s_3 = \mathcal{T}_a(s_1) \Rightarrow \exists s_4 (s_4 = \mathcal{T}_a(s_2) \wedge (s_3, s_4) \in \wp))$ *and*
3. $\forall s_4 (s_4 = \mathcal{T}_a(s_2) \Rightarrow \exists s_3 (s_3 = \mathcal{T}_a(s_1) \wedge (s_3, s_4) \in \wp))$.

The set of bi-simulations on $\mathcal{S}$, ordered by inclusion has a minimal element which is the identity relation denoted by $\wp_0$ and it has a maximal element denoted by $\wp_{max}$ which is an equivalence relation on (or a partition of) $\mathcal{S}$. We will be interested in the maximal element which induces the smallest number of equivalence classes in terms of relation inclusion. $\wp_{max}$ (which is unique) may be obtained as the limit of a decreasing sequence of relations $\wp_i$.

Most algorithms used to solve the bi-simulation problem are based on some form of partition refinement, i.e. they perform successive iterations in which blocks of the current partition are split into smaller blocks, until no block can be split any more. While splitting a block, states that cannot be distinguished are kept in the same block. Two states can be distinguished if one of the states allows a transition with a certain label to a state in a certain block and the other state does not have a transition with the same label to a state in the same block. This means that in our case of timed automata the time associated to a state doesn't satisfy the temporal constraint labeling the transition.

Let $\wp$ be a partition of $\mathcal{S}$. $\wp$ is compatible with $\mathcal{T}$ (it is also called stable) if and only if the following property $\mathcal{P}$ holds:

$$\mathcal{P}(\wp) \equiv \forall a \in A_\tau : \forall B, B' \in \wp :$$
$$(B' \subseteq \mathcal{T}_a^{-1}(B) \vee B' \cap \mathcal{T}_a^{-1}(B) = \emptyset).$$

Correctness of a partition refinement algorithm follows from two facts. First, a stable partition is a bi-simulation relation (states are equivalent if they are in the same block). Second, each computed partition by the refinement of the previous one respects the property $\mathcal{P}$.

Definition 2. *Let $\mathcal{M} = (\mathcal{S}, A_\tau, \mathcal{T}, s_0, \mathcal{L})$ be a labeled transition system and $\wp$ an equivalence relation which is a strong bi-simulation, the quotient labeled transition system denoted by $\mathcal{M}/\wp$ is defined as follows: $\mathcal{M}/\wp = (\mathcal{S}/\wp, A_\tau, \mathcal{T}/\wp, s_0, \mathcal{L}/\wp)$, where:*

- *$\mathcal{S}/\wp$ is the set of equivalence classes noted $\mathcal{C}$, $\mathcal{C} = B \subseteq \mathcal{S} \mid \forall s_1, s_2 \in B : (s_1, s_2) \in \wp)$*
- *$((B = \mathcal{T}_a(B')) \in \mathcal{T}/\wp)$ if and only if $\mathcal{T}_a^{-1}(B) \cap B' \neq \emptyset$*
- *$\forall C \in \mathcal{C} : \mathcal{L}/\wp(C) = \mathcal{L}(s)$, where $s \in C$.*
- *$C_0 = [s_0]$ is the equivalence class of s_0.*

$\mathcal{M}/\wp_{max}$ is the normal form of $\mathcal{M}$ with respect to $\wp_{max}$. We present below the implementation with Maude of our partition-refinement algorithm based on strong bi-simulation. We start from an initial partition of the state space in zones. Each time a zone Z is to be refined, it is split with respect to all its discrete successors by some edge e. We can prove that if all successors are zones, then the result of the split is also a set of zones, that is, convexity is preserved by the split operation.

5.2. Partition-refinement Algorithm

A product timed automaton $\mathcal{A} = \langle \mathcal{Q}, \mathcal{X}, \Sigma, \mathcal{E}, \mathcal{L}^1, \mathcal{L}^2, \mathcal{I} \rangle$ can contain spurious behaviors. This means that there are paths in the product-timed automaton that will never be executed. These spurious behaviors are due to parallel composition which doesn't predict them. Thus, it is necessary to get rid of them before the process of (time abstraction) partition refinement. If not, these spurious behaviors will be part of the overall system behavior and will yield false negative counterexamples. There are techniques to remove spurious behaviors. We have used simulation of the product-timed automaton. The unfired transitions during simulation will be removed from the timed automaton. The result of simulation is a timed graph $\mathcal{G} = \langle \mathcal{Q}, \mathcal{X}, \mathcal{E}, \mathcal{L}, \mathcal{I} \rangle$, where $\mathcal{E}$ is the set of edges without event labels, and $\mathcal{L}$ is defined exactly as $\mathcal{L}^1$. Let $e = \langle q, \theta, X, q' \rangle \in \mathcal{E}$ be an edge such that its guard is θ different from *true*. We will refine the block of source states (q, v) of e represented as a convex zone $Z = (q, \mathcal{V}_Z)$.

The objective of refinement is to abstract the quantitative aspect of time needed to measure the constraint θ. So, this block of states (zone) is refined into sub-zones. The invariant of one of these sub-zones satisfies the constraint θ. But, the invariants of the other sub-zones don't satisfy this constraint. This process of refinement will continue until there are no blocks to refine.

The operators over temporal constraints, used in the algorithm of partition refinement are defined as follows.

1. `Var`(θ) is the set of clock variables in the formula θ.
2. `With`(θ, x) is the constraint θ reduced to a constraint defined only on the clock variable x (e.g. `With`$(x = 1 \wedge y < 2 \wedge z \leq 3, x) \equiv x = 1)$
3. `Without`(θ, x) is the constraint θ reduced to a constraint defined without the clock variable x (e.g. `Without`$(x = 1 \wedge y < 2 \wedge z \leq 3, x) \equiv y < 2 \wedge z \leq 3)$
4. `&` is the intersection operator (e.g. $x \leq 2$ `&` $x \geq 2 \equiv x = 2$). θ_1 `&` $\theta_2 = \emptyset$ if `Var`$(\theta_1) \cap$ `Var`$(\theta_2) = \emptyset$.
5. $\setminus$ is the set difference operator (e.g. $x \leq 2 \setminus x = 1 \equiv x < 1 \vee (x > 1 \wedge x \leq 2)$, it is not convex).
6. `floor`(θ) if θ is convex, then this operator will return θ itself, else it returns the constraint representing the lower convex valuations. The constraint θ is defined on one clock variable (e.g `floor`$(x < 1 \vee (x > 1 \wedge x \leq 2)) \equiv x < 1$).
7. `ceil`(θ) if θ is convex, then this operator will return $\emptyset$, else it returns the constraint representing the upper convex valuations. The constraint θ is defined on one clock variable (e.g `ceil`$(x < 1 \vee (x > 1 \wedge x \leq 2)) \equiv x > 1 \wedge x \leq 2$).

Our defined Maude operator `split` splits a zone that is a source of an edge $e = \langle q, \theta, X, q' \rangle$, taken arbitrarily from the set $\mathcal{E}$ of the current partition, where $\theta \neq true$. The refinement (splitting) is based on a clock variable x taken also arbitrarily from the set of clock variables in the constraint θ. The zone is split into at most three sub-zones. These sub-zones have the same location q, but with different invariants. Their union equals to $\mathcal{I}(q)$. Because their invariants are different and for algorithm simplicity, we will denote their location q differently to distinguish them. This will not have any effect on the algorithm results.

The first sub-zeno (with discrete state q_x) has the invariant $\mathcal{I}(q_x) =$ `With`$(\theta, x) \wedge$ `Without`$(\mathcal{I}(q), x)$ and an outgoing edge $\langle q_x,$ `Without`$(\theta, x), \emptyset, q' \rangle$.

If `floor(With`$(\mathcal{I}(q), x) \setminus$ `With`$(\theta, x)) \neq \emptyset$, we have a second sub-zone with a discrete state q_l with an invariant $\mathcal{I}(q_l) =$ `floor(With`$(\mathcal{I}(q), x) \setminus$ `With`$(\theta, x)) \wedge$ `Without`$(\mathcal{I}(q), x)$. This sub-zone has an outgoing edge $\langle q_l, true, \emptyset, q_x \rangle$.

If `ceil(With`$(\mathcal{I}(q), x) \setminus$ `With`$(\theta, x)) \neq \emptyset$, then we have a third sub-zone with a discrete state q_u and an invariant $\mathcal{I}(q_u) =$ `ceil(With`$(\mathcal{I}(q), x) \setminus$ `With`$(\theta, x)) \wedge$ `Without`$(\mathcal{I}(q), x)$. This sub-zone has an ingoing edge $\langle q_x, true, \emptyset, q_u \rangle$. The three new sub-zones will be marked by the same set of atomic propositions $\mathcal{L}(q)$.

At the end of this iteration, the edge e and the zone Z will be removed and replaced by the new edges and the new sub-zones. The other outgoing and incoming edges from and to the zone Z will be updated according to the new partition.

The non-zenoness of the timed automaton and the convexity of its constraints guarantee that the produced partition has zones preserving the convexity and the non-zenoness. Moreover, the algorithm terminates.

5.3. Quotient Graph

The partition-refinement algorithm generates a stable partition $\wp_{max}$ which is the coarsest. Each block in this partition is characterized by an invariant and a unique discrete state. These blocks are reachable and their invariants are convex. The edges of this partition are of the form $\langle q, true, \emptyset, q' \rangle$. This partition can be easily represented by a graph, we call it the quotient graph $G_{\wp_{max}}$. The set $\mathcal{C}$ of nodes of $G_{\wp_{max}}$ is the set of the partition blocks. Thus, a node corresponding to block B_i is denoted C_i. The edges of $G_{\wp_{max}}$ are the edges in the partition $\wp_{max}$ between the different blocks in addition to edges of the form $\langle q, true, \emptyset, q \rangle$ if the invariant $\mathcal{I}(q)$ is bounded only from below or a state doesn't have an outgoing edge. The strong bisimulation quotient graph $(G_{\wp_{max}})$ is generated by the algorithm of partition refinement and, as it is defined, has the following properties:

$G_{\wp max}$**-Property 1**: $G_{\wp max}$ is stable which means that $\forall C_1, C_2 \in G_{\wp max}$, then by definition, if $C_1 \xrightarrow{\tau} C_2$ then $\forall s_1 \in C_1$ there exists $s_2 \in C_2$, such that $s_1 \xrightarrow{\delta} s_2$, for some $\delta \in R^+$ and if $C_1 \xrightarrow{e} C_2$, for some edge e, then $\forall s_1 \in C_1$ there exists $s_2 \in C_2$, such that $s_1 \xrightarrow{e} s_2$.

$G_{\wp max}$**-Property 2**: Given a path $\rho = C_1 \Rightarrow C_2 \Rightarrow \cdots$ of $G_{\wp max}$ ($\Rightarrow$ means discrete or time transition) and a run $r = s_1 \Rightarrow s_2 \Rightarrow \cdots$, we say that r is inscribed in ρ if for all $i \geq 1 : s_i \in C_i$ and, if $C_i \xrightarrow{\tau} C_{i+1}$ then there exists $\delta > 0$ such that $s_i \xrightarrow{\delta} s_{i+1}$, if $C_i \xrightarrow{e} C_{i+1}$ then $s_i \xrightarrow{e} s_{i+1}$. It is easy to conclude that every run r is inscribed in a unique path ρ in $G_{\wp max}$. And inversely, if $\rho = C_1 \Rightarrow C_2 \Rightarrow \cdots$ is a path in $G_{\wp max}$ then for all $s_1 \in C_1$ there exists a run r starting from s_1 and inscribed in ρ.

$G_{\wp max}$**-Property 3**: Any time transition traverses a unique (finite) set of classes. Also, if $(s, s') \in \wp_{max}$ then for any time transition $s \xrightarrow{\delta} s + \delta$, there exists a time transition $s' \xrightarrow{\delta'} s' + \delta'$ such that $(s + \delta, s' + \delta') \in \wp_{max}$ and the two transitions traverse the same classes.

Example 3. *The following is the generated Maude module representing the quotient graph of the example (the specification of the complex real-time system and its property) presented in Section 3. Each name (*`"a-q2x"`*, for example) of an equivalent class is separated to two names. One is known from the system specification (*`a`*), and the other is transparent produced by the transformation of the TLTL formula and by the bi-simulation process (*`q2x`*). The equations are used to mark the different states by propositions and the rules are used to represent the transitions.*

```
mod 'MC-Model is
 including 'MODEL-CHECKER .
 sorts none .
 subsort 'String < 'State .
 op 'Z : nil -> 'Prop [none] .
 op 'p : nil -> 'Prop [none] .
 op 'r : nil -> 'Prop [none] .
 none
 eq '_|=_['"a-q2x".String,'Z.Prop]=
 'true.Bool [none] .
eq '_|=_['"a-q2xl".String,'Z.Prop]=
'true.Bool [none] .
eq '_|=_['"b-q1z".String,'p.Prop]=
'true.Bool [none] .
eq '_|=_['"b-q1zl".String,'p.Prop]=
'true.Bool [none] .
eq '_|=_['"b-q2x".String,'Z.Prop]=
'true.Bool [none] .
eq '_|=_['"b-q2x".String,'p.Prop]=
'true.Bool [none] .
eq '_|=_['"b-q2xl".String,'Z.Prop]=
'true.Bool [none] .
eq '_|=_['"b-q2xl".String,'p.Prop]=
'true.Bool [none] .
eq '_|=_['"c-q2".String,'Z.Prop]=
'true.Bool [none] .
eq '_|=_['"c-q2".String,'r.Prop]=
'true.Bool [none] .
rl '"a-q1zlx".String => '"a-q1zlxu"
.String [none] .
rl '"a-q1zlx".String => '"a-q1zxux"
.String [none] .
rl '"a-q1zlx".String => '"b-q1zl"
.String [none] .
rl '"a-q1zlxl".String => '"a-q1zlx"
.String [none] .
rl '"a-q1zlxl".String => '"a-q1zxux"
.String [none] .
rl '"a-q1zlxu".String => '"a-q1zlxu"
.String [none] .
rl '"a-q1zlxu".String => '"a-q1zxux"
.String [none] .
rl '"a-q1zx".String => '"a-q1zxuxl"
.String [none] .
rl '"a-q1zx".String => '"a-q2xl"
.String [none] .
rl '"a-q1zx".String => '"b-q1zl"
.String [none] .
rl '"a-q1zxl".String => '"a-q1zx"
.String [none] .
rl '"a-q1zxl".String => '"a-q2xl"
.String [none] .
rl '"a-q1zxux".String => '"a-q1zxux"
.String [none] .
rl '"a-q1zxux".String => '"a-q2xl"
.String [none] .
rl '"a-q1zxux".String => '"c-q2"
.String [none] .
rl '"a-q1zxuxl".String => '"a-q1zxux"
.String [none] .
rl '"a-q1zxuxl".String => '"a-q2xl"
.String [none] .
rl '"a-q2x".String => '"a-q2x"
```

```
 .String [none] .
 rl '"a-q2x".String => '"c-q2"
 .String [none] .
 rl '"a-q2xl".String => '"a-q2x"
 .String [none] .
 rl '"b-q1z".String => '"b-q2xl"
 .String [none] .
 rl '"b-q1zl".String => '"b-q1z"
 .String [none] .
 rl '"b-q2x".String => '"c-q2"
 .String [none] .
 rl '"b-q2xl".String => '"b-q2x"
 .String [none] .
 rl '"c-q2".String => '"c-q2"
 .String [none] .
endm
```

6. Maude LTL Model Checking

In this section we show that the strong bi-simulation $\wp_{max}$ preserves the LTL properties. The timed automaton model checking can be reduced to model checking a finite graph, the strong bi-simulation quotient graph (G_{max}) generated by the algorithm of partition refinement.

Consider a labeled transition system $\mathcal{M} = (\mathcal{S}, A_\tau, \mathcal{T}, s_0, \mathcal{L})$ modeling a strongly non-zeno timed automaton $\mathcal{A}$ and an LTL formula φ. We want to check whether $\mathcal{M}$ satisfies φ. Let $\wp_{max}$ be a strong bi-simulation on $\mathcal{M}$. From $G_{\wp_{max}}$-Property 3 of $G_{\wp_{max}}$, we can conclude that for any LTL formula φ and any pair of states $(s, s') \in \wp_{max}$, $s \models_{\mathcal{M}} \varphi$ if and only if $s' \models_{\mathcal{M}} \varphi$.

A formula is said to hold in a node C of $G_{\wp_{max}}$ if it is satisfied in some state of C (this implies that the formula is satisfied in any state of C). Now, the problem of verifying if a state $s \in \mathcal{S}$ satisfies the LTL formula φ ($s \models_{\mathcal{M}} \varphi$) is reduced to checking if the node $C \in \mathcal{C}$ containing the state s satisfies the formula φ ($C \models \varphi$). The following lemma gives the correctness of the model checking.

Lemma 1. *Let $\mathcal{M} = (\mathcal{S}, A_\tau, \mathcal{T}, s_0, \mathcal{L})$ be a labeled transition system modeling a strongly non-zeno timed automaton, $\mathcal{L}$ is a labeling function associating to each discrete state a set of atomic propositions from AP. Let $\wp_{max}$ be a strong bi-simulation on $\mathcal{M}$ and $G_{\wp_{max}}$ is its quotient graph with the set of nodes $\mathcal{C}$. Let C be in $\mathcal{C}$ and φ an LTL formula. $C \models \varphi$ if and only if $\forall s \in C$, $s \models_{\mathcal{M}} \varphi$.*

Proof. *The proof is by induction on the syntax of φ. The basis (φ is an atomic proposition) comes from the fact that $\wp_{max}$ respects $\mathcal{L}$.*

Consider the case where $\varphi = \neg\varphi_1$. By the semantics of LTL, $C \models \neg\varphi_1$ if and only if $C \not\models \varphi_1$. Now using the induction hypothesis, $C \not\models \varphi_1$ if and only if $\forall s \in C, s \not\models_{\mathcal{M}} \varphi_1$ (i.e. $\forall s \in C, s \models_{\mathcal{M}} \neg\varphi_1$).

Consider the case where $\varphi = \varphi_1 \wedge \varphi_2$. By the semantics of LTL $C \models \varphi_1 \wedge \varphi_2 \Leftrightarrow C \models \varphi_1 \wedge C \models \varphi_2$. By induction hypothesis, $C \models \varphi_1 \Leftrightarrow \forall s \in C, s \models_{\mathcal{M}} \varphi_1$ and $C \models \varphi_2 \Leftrightarrow \forall s \in C, s \models_{\mathcal{M}} \varphi_2$. Using the semantics of LTL, $\forall s \in C, s \models_{\mathcal{M}} \varphi_1 \wedge \varphi_2$.

The case where φ is of the form $\varphi_1 U \varphi_2$ can be proved by the fact that if $C \not\models \varphi$, we can extract a run r which falsifies φ, from the path ρ starting from the node C using the property $G_{\wp_{max}}$-Property 2.

Consider the case where $\varphi = \Box\varphi_1$. By the semantics of LTL, $C \models \Box\varphi_1$ if and only if any node C' on any path starting from C, $C' \models \varphi_1$. By the induction hypothesis, $C' \models \varphi_1$ if and only if $\forall s \in C', s \models_{\mathcal{M}} \varphi_1$. Then, using $G_{\wp_{max}}$-Property 1, $G_{\wp_{max}}$-Property 2, and the LTL semantics, $C \models \varphi$ if and only if $\forall s \in C, s \models_{\mathcal{M}} \varphi$.

Example 4. The TLTL model checking of the problem $\langle a, x = 0\rangle \models \diamond_{\geq 2} r$ on the model of the timed automaton of Figure 1 is then reduced to Maude LTL model checking of $C_0 \models (\neg Prop('r)) U (Prop('Z) \wedge Prop('r))$ on the model represented by the graph shown as a Maude module in the previous section, where $C_0 =$ `a-q1zlxl`. This LTL formula is not satisfied and the model checking returns a trace as a counterexample.

```
The property is not satisfied ...
This is a counter example:
    a-q1zlxl -> a-q1zlx -> a-q1zlxu -> a-
q1zxux ->
    a-q2xl -> a-q2x
```

By mapping to the concrete timed automaton, the discrete states of the nodes (classes) `a-q1zlxl`, `a-q1zlx`, `a-q1zlxu`, `a-q1zxux`, `a-q2xl`, and `a-q2x` are `a`. Thus, the concrete trace is $\langle a, x = 0\rangle \overset{\delta_0}{\rightarrow} \langle a, x = \delta_0\rangle \overset{\delta_1}{\rightarrow} \langle a, x = \delta_0 + \delta_1\rangle \overset{\delta_2}{\rightarrow} \cdots$.

7. Complexity and Implementation Results

We denote the size of a timed automaton $\mathcal{A} = \langle \mathcal{Q}, \mathcal{X}, \Sigma, \mathcal{E}, \mathcal{L}^1, \mathcal{L}^2, \mathcal{I} \rangle$ by the pair $(|\mathcal{Q}|, |\mathcal{E}|)$, where $|\mathcal{Q}|$ is the number of discrete states and $|\mathcal{E}|$ is the number of edges. For a TLTL formula ψ with n temporal constraints, the algorithm `Transform` generates a timed automaton $\mathcal{A}_\psi$ with one clock variable, $|\mathcal{Q}_\psi| \leq n+1$ and $|\mathcal{E}_\psi| \leq n$. The size of $\mathcal{A} \parallel \mathcal{A}_\psi$ is at most $(|\mathcal{Q}| \times |\mathcal{Q}_\psi|, |\mathcal{E}| \times |\mathcal{E}_\psi| + |\mathcal{E}| + |\mathcal{E}_\psi| - 1)$.

The size of the quotient graph $G_{\wp_{max}}$ is defined by the pair of the number of its nodes and number of its edges, which are at most $(3 \times (|\mathcal{E}| \times |\mathcal{E}_\psi| + |\mathcal{E}| + |\mathcal{E}_\psi| - 1), |\mathcal{Q}| \times |\mathcal{Q}_\psi| + 3 \times (|\mathcal{E}| \times |\mathcal{E}_\psi| + |\mathcal{E}| + |\mathcal{E}_\psi| - 1))$. The partition-refinement algorithm generates the quotient graph in a time of $\mathcal{O}(|\mathcal{Q}| \times |\mathcal{Q}_\psi| + 3 \times (|\mathcal{E}| \times |\mathcal{E}_\psi| + |\mathcal{E}| - 1))$.

To obtain confidence in the correctness of the implementation, our first experiments concentrated on existing case studies taken from real-time model checking literature. The first case study is the analysis of The CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) protocol which works as follows [31, 20]. When a station has data to send, it first listens to the channel. If it is idle (i.e., no other station is transmitting) the station begins sending its message. However, if it detects a busy channel, it waits a random amount of time and then repeats the operation. When a collision occurs, because several stations transmit simultaneously, then all of them detect it, abort their transmissions immediately and wait a random time to start all over again. If two messages collide then they are both lost.

The propagation delay of the channel plays an important role in the performance of the CSMA/CD protocol. It is possible that just after a station begins sending, another one becomes ready to send. If it senses the channel before the signal of the former arrives, it will find the channel idle and will start sending too. Hence, a collision will happen. Let σ be the time for a signal to propagate between the two farthest stations. Suppose that at time t_0 a station S_0 begins sending a message. Thus, within the time interval $[t_0, t_0 + \sigma)$, it is still possible that some station S_i transmits if it has data, causing a collision. However, after time $t_0 + \sigma$, the channel will be sensed busy by all the stations until the current message is delivered. Hence, the maximum time the channel could be sensed idle by any station after the beginning of a transmission is σ.

Based on the fact above, we might think that a station that does not hear a collision for a time equal to σ, could be sure that no other station would interfere. However, this conclusion is wrong. Due to the propagation delay, the noise burst caused by the collision could take a time σ to arrive. In fact, in the worst case it would take $2 \times \sigma$ for a station to detect a collision.

In case of collision, each station waits randomly a time between 0 and $2 \times \sigma$ before trying again. In general, after i collisions, a station waits a random time between 0 and $2^i \times \sigma$. Moreover, after too many retrials (e.g., 16 as in the 802.3 standard [20]) a failure signal could be reported to the higher layers.

Assume that only messages of equal length are sent and let λ be the time to send a message. Then if no collision occurs, a message will be completely delivered in a time equal to $\lambda + \sigma$. For instance, for a 10Mbps Ethernet with a typical worst case round trip propagation delay of 51.2us, we set $2 \times \sigma$ to be 51.2us and for standard frames of 1024bytes, λ is approximately 782us.

The system consists of n stations $S_1, \cdots, S_n$ and the medium M. The behavior of the medium is as follows. Initially, it is ready and it can accept a message from any station. Suppose that one station begins transmitting (TRANSMIT). There is a time interval of length σ within which the medium can accept data from the other station, causing a collision (CD). This is modeled with a watchdog which is canceled when a collision occurs. In the case of a collision, it takes time σ to the medium to propagate it. This is naturally modeled with a timeout. If no collision occurs, the medium waits for the termination signal. When it arrives, the medium returns to the initial state. The overall specification is obtained putting all the above (stations and the medium) in parallel. $CSMA/CD_n = Medium \parallel S_1 \parallel \cdots \parallel S_n$. The following is the specification of a system composed of two senders (the process specification of the second sender is the same as for Sender1) and a medium.

```
(
 system CSMA-CD :
  prop INIT1 SEND1 TRANSM1 CD1
       INIT2 SEND2 TRANSM2 CD2
       INITM TRANSMM CDM ;
  chan BEGIN BUSY CD END ;
  process Sender1 :
     clocks x1
     state      s0 :
     { INIT1 }
                s1 : x1 = 0
{ SEND1 }
                s2 : x1 <= 782
{ TRANSM1 }
                s3 : x1 <= 52
{ CD1 }
     init s0
     trans
         s0 -> s1 :          { x1 }
 { SEND1 }
         s0 -> s0 :          { }
 { ?CD }
         s1 -> s2 :          { x1 }
 { !BEGIN }
         s1 -> s3 :          { x1 }
 { ?BUSY }
         s1 -> s3 :          { x1 }
 { ?CD }
         s2 -> s3 :          { x1 }
 { ?CD }
         s2 -> s0 : x1 = 782 { x1 }
 { !END }
         s3 -> s3 :          { }
 { ?CD }
         s3 -> s1 :          { x1 }
 { TAU1 }
           ;
   process Medium :
     clocks m
     state      m0 :
     { INITM }
                m1 :
{ TRANSMM }
                m2 :  m <= 26
{ CDM }
     init m0
     trans
         m0 -> m1 :         { m }
 { ?BEGIN }
         m1 -> m2 : m < 26  { m }
 { ?BEGIN }
         m1 -> m1 : m >= 26 { }
 { #BUSY }
         m1 -> m0 :         { m }
 { ?END }
         m2 -> m0 : m <= 26 { m }
 { #CD }
         ;
 |= [] ( ( TRANSM1 /\ TRANSM2 ) =>
    <> { <= 26 } ( CD1 /\ CD2 ) ) . )
```

#BUSY means broadcast send via channel BUSY. We have verified the following real-time property expressed in the logic TLTL. When a collision occurs, because two stations $k \neq j$ transmit simultaneously, they both detect it at most σus later:

$$\psi \equiv \Box(TRANSMIT_k \wedge TRANSMIT_j \Rightarrow \diamond_{\leq\sigma} CD_k \wedge CD_j).$$

The second case study is the analysis of FDDI (Fiber Distributed Data Interface) (example taken from [14]). FDDI is a high performance fiber optic token ring Local Area Network. We consider a network composed by n identical stations $S_1, \cdots, S_n$ and a ring, where the stations can communicate by synchronous messages with high priority and asynchronous messages with low priority. The timed automaton that models the protocol is obtained as the parallel composition $FDDI_n = Ring \parallel S_1 \parallel \cdots \parallel S_n$, where the automata synchronize through actions. The following is the specification of a system composed of two stations (the process specification of the second station is the same as for Station1) and a ring.

```
(
 system FDDI :
  prop ZIDLE1 ZSYNC1 ZASYNC1 YIDLE1 YSYNC1
  YASYNC1
       ZIDLE2 ZSYNC2 ZASYNC2 YIDLE2
       YSYNC2 YASYNC2
       RTO1 RING1 RTO2 RING2 ;
  chan TT1 TT2 RT1 RT2 ;
  process Station1 :
     clocks x y z
     state      s1_z_idle :
     { ZIDLE1 }
                s1_z_sync : x <= 20
{ ZSYNC1 }
                s1_z_async : z <= 120
{ ZASYNC1 }
                s1_y_idle :
{ YIDLE1 }
                s1_y_sync : x <= 20
{ YSYNC1 }
                s1_y_async : y <= 120
{ YASYNC1 }

     init s1_z_idle
     trans
         s1_z_idle -> s1_z_sync :
 { x , y } { ?TT1 }
         s1_z_sync -> s1_z_async :
 x >= 20 /\ z < 120

   { } { }
         s1_z_sync -> s1_y_idle :
 x >= 20 /\ z >= 120

  { } { !RT1 }
         s1_z_async -> s1_y_idle :
 { } { !RT1 }
         s1_y_idle -> s1_y_sync :
 { x , z } { ?TT1 }
         s1_y_sync -> s1_z_idle :
 x >= 20 /\ y >= 120

  { } { !RT1 }
         s1_y_sync -> s1_y_async :
 x >= 20 /\ y < 120

   { } { }
         s1_y_async -> s1_z_idle :
 { } { !RT1 }
           ;
   process Ring :
     clocks x
     state      ring_to_1 : x <= 0
     { RTO1 }
                ring_1 :
{ RING1 }
                ring_to_2 : x <= 0
{ RTO2 }
                ring_2 :
{ RING2 }

     init ring_to_1
     trans
         ring_to_1 -> ring_1 : x <= 0
 { } { !TT1 }
         ring_1 -> ring_to_2 :
 { x } { ?RT1 }
         ring_to_2 -> ring_2 : x <= 0
 { } { !TT2 }
         ring_2 -> ring_to_1 :
 { x } { ?RT2 }
         ;
 |= [] ( ( ZIDLE1 => <> { <= 120 }
 ZASYNC1 ) /\
         ( YIDLE1 => <> { <= 120 }
 YASYNC1 ) /\
         ( ZIDLE2 => <> { <= 120 }
 ZASYNC2 ) /\
         ( YIDLE2 => <> { <= 120 }
 YASYNC2 ) ) . )
```

The formula of TLTL 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:

$$\psi = \Box((S_i = idle) \Rightarrow \diamond_{\leq c}(S_i = async)),$$

where $S_i = idle$ is any state $s \in \mathcal{S}$ verifying the condition that the automaton corresponding to station number i is in the location idle.

The experiments were done on a Pentium IV at 1GHz with 512MB of memory. Our approach has been able to generate the quotient graph for up to 5 processes (for the CSMA/CD protocol, including the medium) and 6 processes (for the FDDI protocol, including the ring).

Table 1 presents the results of the different experiments. The reported results are expressed

number of stations	*CSMA/CD*			*FDDI*		
	nodes	edges	time (s)	nodes	edges	time (s)
2	48	298	95	36	196	50
3	92	1012	200	72	624	170
4	264	4896	4200	212	3048	3700
5	–	–	–	416	7012	6900
6	–	–	–	–	–	–

Table 1. Experimental results.

by number of nodes, number of edges for each quotient graph generated using different configurations and the time consumed (in seconds). The symbol - means the tool fails due to lack of memory.

First experiments thus show that our real-time model checking technique performs relatively well. Although these positive results are only based on limited experience with the tool, we believe that further experiments will show that they are of a more general character. This would show that combination of transformation and partition refinement is a useful approach to real-time model checking, that could be at least as valuable as the currently followed approaches. To our knowledge, it doesn't exist a tool for TLTL model checking to compare with our approach (it is known that the model checking for the logic TLTL is undecidable).

8. Conclusion

In this paper, we have presented a technique for model checking dense complex real-time systems implemented with Maude. This method is based on the reduction of TLTL specifications to LTL. The timed behavior of the TLTL specification is captured and represented as a timed automaton. This timed automaton is composed with the original timed automaton modeling the timed system. Then, a time abstraction technique based on strong bi-simulation, is used to generate a finite graph modulo the TLTL specification. Then, the Maude LTL model checker is used for performing LTL model checking on this graph. We have taken advantage of the reflective aspect of Rewriting Logic to implement this tool.

The correctness of this technique is mathematically proved and tested on many small examples. Relatively complex specifications of the protocols CSMA/CD and FDDI were also successfully tested. Our future work is to generalize this technique to accept real-time semantics defined in Real-Time Maude (RT-Maude) [26]. Thus, a real-time system specified with RT-Maude could be analyzed using this method.

References

[1] R. Alur, C. Courcoubetis and D. L. Dill, Model checking in dense real time. *Information and Computation*, 104(1):2–34, 1993.

[2] R. Alur and D. Dill, Automaton for modeling real-time systems. In *Lecture Notes in Computer Science (17th ICALP)*, number 443. Springer-Verlag, 1990.

[3] R. Alur and T. A. Henzinger, Logics and models of real time: a survey. In *Lecture Notes in Computer Science (Real Time: Theory in Practice)*, number 600, pages 74–106. Springer-Verlag, 1992.

[4] R. Alur and T. A. Henzinger, Real time logics: Complexity and expressiveness. *Information and Computation*, 104:35–77, 1993.

[5] R. Alur and T. A. Henzinger, A really temporal logic. *J. Assoc. Comput. Mach.*, 41:181–204, 1994.

[6] A. K. Bauer, *Model-based runtime analysis of distributed reactive systems.* PhD thesis, University of Munchen, Germany, 2007.

[7] A. Bouajjani, J. C. Fernandez, N. Halbwachs, P. Raymond and C. Ratel, Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[8] A. Bouajjani, S. Tripakis and S. Yovine, On-the-fly symbolic model-checking for real-time systems. In *IEEE RTSS'97*. IEEE Computer Society Press, 1997.

[9] M. Bourahla and M. Benmohamed, Verification of real-time systems by abstraction of time constraints. In *IPDPS(FMPPTA)*. IEEE Computer Society Press, 2003.

[10] M. Bourahla and M. Benmohamed, Analysis of real-time systems with ctl model checkers. *Electronic Notes in Theoretical Computer Science, Elsevier*, 133:41–60, 2005.

[11] U. Brockmeyer and G. Wittich, Real-time verification of statemate designs. In *Lecture Notes in Computer Science (Computer Aided Verification)*, number 1427, pages 537–541. Springer-Verlag, 1998.

[12] M. Clavel, Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[13] M. Clavel, *Maude 2.3 Manual.* http://maude.cs.uiuc.edu/manual, 2005.

[14] C. Daws, A. Olivero, S. Tripakis and S. Yovine, The tool kronos. In *Lecture Notes in Computer Science (Verification and Control of Hybrid Systems)*, number 1066. Springer-Verlag, 1995.

[15] C. Daws and S. Tripakis, Model checking of real-time reachability properties using abstractions. In *Lecture Notes in Computer Science (Tools and Algorithms for the Construction and Analysis of Systems)*, number 1384. Springer-Verlag, 1998.

[16] D. D'SOUZA, A logical characterisation of event clock automata. *Int. Journ. Found. Comp. Sci.*, 14(4):625–639, 2003.

[17] S. EKER, J. MESEGUER AND A. SRIDHARANARAYANAN, The Maude LTL model checker and its implementation. In *Lecture Notes in Computer Science (10th Intl. SPIN Workshop)*, number 2648, pages 230–234. Springer-Verlag, 2003.

[18] T. A. HENZINGER AND O. KUPFERMAN, From quantity to quality. In *Lecture Notes in Computer Science (Workshop on Hybrid and Real-Time Systems)*, number 1201, pages 48–62. Springer-Verlag, 1997.

[19] T. A. HENZINGER, X. NICOLLIN, J. SIFAKIS AND S. YOVINE, Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[20] IEEE, ANSI/IEEE 802.3, ISO/DIS 8802/3. IEEE Computer Society Press, 1985.

[21] I. KANG, I. LEE, AND Y. S. KIM, An efficient state space generation for the analysis of real-time systems. *IEEE Transactions on Software Engineering*, 26(5):453–477, 2000.

[22] F. LAROUSSINIE, N. MARKEY AND PH. SCHNOEBELEN, Efficient timed model checking for discrete-time systems. *Theoretical Computer Science*, 353:249–271, 2006.

[23] K. G. LARSEN, P. PETTERSON AND W. YI, Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1), 1997.

[24] R. MILNER, A calculus of communicating systems. In *Lecture Notes in Computer Science*, number 92. Springer-Verlag, 1980.

[25] M. O. MOLLER, H. RUEB AND M. SOREA, Predicate abstraction for dense real-time systems. *Electronic Notes in Theoretical Computer Science, Elsevier*, 65, 2002.

[26] P. C. ÖLVECZKY AND J. MESEGUER, Semantics and pragmatics of real-time maude. *Higher-Order and Symbolic Computation*, 2006.

[27] R. PAIGE AND R. TARJAN, Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6), 1987.

[28] J. F. RASKIN AND P. Y. SCHOBBENS, The logic of event clocks - decidability, complexity and expressiveness. *Journ. of Autom. Lang. and Comb.*, 4(3):247–286, 1999.

[29] O. SOKOLSKY AND S. A. SMOLKA, Local model checking for real-time systems. In *Lecture Notes in Computer Science (CAV)*, number 939. Springer-Verlag, 1995.

[30] R. SPELBERG, H. TOETENEL AND M. AMMERLAAN, Partition refinement in real-time model checking. In *Lecture Notes in Computer Science (Formal Techniques in Real-Time and Fault-Tolerant Systems)*, number 1486. Springer-Verlag, 1998.

[31] A. S. TANENBAUM, *Computer Networks*. Prentice-Hall, Englewood Cliffs, 1989.

[32] S. TRIPAKIS AND S. YOVINE, Analysis of timed systems using time-abstracting bi-simulations. *Formal Methods in System Design*, 18:25–68, 2001.

[33] M. YANNAKAKIS AND D. LEE, An efficient algorithm for minimizing real-time transition systems. In *Lecture Notes in Computer Science*, number 697, pages 210–224. Springer-Verlag, 1993.

Received: May, 2008
Revised: May, 2008
Accepted: February, 2009

Contact address:
Dr. Mustapha Bourahla
Computer Science Department
University of Biskra
BP 145 RP, Biskra 07000, Algeria
e-mail: `mbourahla@hotmail.com`

MUSTAPHA BOURAHLA has PhD degree in computer science from the University of Biskra, Algeria (2007) and he has the Master degree in computer science from the University of Montreal, Canada (1989). He was a member of the VHDL group at Bell-Northern Research, Ottawa, Canada (1989-1993). He worked for Bell Canada for one year. Now, he is teacher-researcher at the University of Biskra (Algeria). He has publications in the domains of VLSI and formal methods. His current research interests are formal methods, especially model checking critical systems. Dr. Bourahla is a member of a research group working in the domains of VLSI and formal methods at the University of Biskra (Algeria).