Lot 3.1

Animation

Symbolic Simulation and Formal Verification of Updatable Timed Automata using a Rewrite System

Description :	Simulateur d'automates Temporisés étendus AVERROES.
Auteur(s) :	Olivier BOURNEZ, Térence SOUSSAN, Bertrand TAVERNIER,
Référence :	AVERROES / Lot 3.1 / Fourniture 1 / V1.0
Date :	14 mars 2006
Statut :	validé
Version :	1.0

Réseau National des Technologies Logicielles
Projet subventionné par le Ministère de la Recherche et des Nouvelles Technologies

CRIL Technology, France Télécom R&D, INRIA-Futurs, LaBRI (Univ. de Bordeaux – CNRS), LIX (École Polytechnique, CNRS) LORIA, LRI (Univ. de Paris Sud – CNRS), LSV (ENS de Cachan – CNRS)

avril 2003	V 0.1	version initiale
14 mars 2006	V 1.0	mise au format averroes

Table des matières

Résumé

In the paper we present a tool, fully developed using rewrite rules and strategies in the ELAN system, which supports symbolic simulation and formal verification of reachability properties on reactive timed systems. Systems proved or simulated are modeled as a synchronized product of Updatable Timed Automata (Timed Automata extended with bounded variables and non-deterministic updates).
The tool is connected to the CALIFE plateform which allows the design of systems in a graphical way. A dedicated graphical user interface is connected to the plateform for specifying reachability requests by simple clicks. Using that interface, a simulation trace can be built by switching between "step by step" and reachability requests. This process allows one to debug a specification with a kind of breakpoint modeled by a reachability formula.

1 Introduction

Overview. The CALIFE v3.0 plateform is an environment developed under the GPL license (freely downloadable at http ://calife.criltechnology.com) allowing the specification and formal validation of systems described as a synchronized product of (extended) timed automata [14].
The goal of this plateform is not to provide another verification tool but to interface existing tools working on automata in a unique graphical and powerful environment.
The CALIFE plateform is composed of several layers :

1. a system editor which allows the user to model in graphic form a system described as a synchronized product of timed automata ;
2. a model compiler that checks the consistency of the input ;
3. a script engine that generates datas and executes one of the interfaced tool.

Among the interfaced tools there are CMC[5], COQ[6], HyTech[9], KRONOS[10], UPPAAL[16], and a model-checker built using ELAN rewrite system [3].
Considering that a simulator was missing to CALIFE graphical interface, it was proposed to build this simulator by adapting this latter model-checker. Indeed, the code needed to realize a simulator is close to the code used by model-checkers, and unlike all the other tools, the code of this latter model checker is not hard-coded in a classical imperative programming language but written using a set of high-level rules that are executed using the ELAN rewrite system, hence providing clearly a great flexibility.
That is why it was decided to start from the model-checker of [3] (about 1000 lines, tested only few simple timed automata) to build a new model-checker fully usable as a simulator tool by the CALIFE plateform. The new tool is now about 5000 lines.

This paper. This paper focus on describing

1. the extensions that have been added to the previous model-checker concerning the class of automata that can now be model-checked ;
2. the extensions that have been added to the previous model-checker concerning possible queries that allow one to use it as simulator ; for the CALIFE plateform
3. the connexion of the new model-checker with the CALIFE plateform.

The new model-checker. Compared to the tool described in [3], the new model-checker/simulator now offers :

1. the possibility of defining products of automata synchronized by synchronization vectors (the most general and powerful way to define synchronization) ;
2. the possibility of qualifying some locations as urgent ;
3. the possibility of doing depth-first, breadth-first, concentric explorations in order to get for example all the states reachable in some fixed number of discrete transitions ;
4. the possibility of obtaining a trace of a given execution as well as the list of all labels used in that transitions ;
5. the possibility to use wildcards in the description of states tested for reachability as well as detecting deadlocks ;
6. the possibility of considering automata with integer bounded variables and with very general updates.

The latter point, concerning integer variables is very important, since it transforms our tools not only in a model checker for classical timed automata [2], but also in a model checker for the *updatable timed automata* model presented in [4]. As far as we know, this is the first simulator which is really able to deal with this very general model.

Organization. In Section 2 we recall briefly the ELAN system. In Section 3 we present the CALIFE plateform. Updatable timed automata and the associated model-checking algorithms for testing reachability properties are recalled in Section 4. In Section 5, we describe the way these algorithms have been implemented using rewrite rules and strategies in the ELAN system. In Section 6, we explain how the model-checker is connected to the CALIFE plateform in order to get a simulator, and we discuss the resulting functionalities for CALIFE plateform users. Section 7 is a conclusion.

2 The ELAN system

The simulator for the CALIFE plateform is built using the rewrite system ELAN.
The ELAN system [8] takes from functional programming the concept of abstract data types and the function evaluation principle based on rewriting. In ELAN, a program is a set of labeled conditional rewrite rules with local affectations

$$\ell : l \Rightarrow r \text{ if } c \text{ where } w$$

Informally, rewriting a ground term t consists of selecting a rule whose left-hand side (also called pattern) matches the current term (t), or a subterm ($t_{|\omega}$), computing a substitution σ that gives the instantiation of rule variable ($l\sigma = t_{|\omega}$), and if instantiated condition c is satisfied ($c\sigma$ reduces to $true$), applying substitution σ enriched by local affectation w to the right-hand side to build the reduced term.
One of the main originalities of the ELAN language is to provide strategies as first class objects of the language. This allows the programmer to specify in a precise and natural way the control on the rule applications.
The full ELAN system includes a preprocessor, an interpreter, a compiler, and standard libraries [8]. The ELAN compiler is able to generate code that applies up to 15 millions rewrite rules per second on typical examples where no non-determinism is involved and typically between 100 000 and one million controlled rewrite per second in presence of associative-commutative operators and non-determinism [11, 12].

3 The CALIFE plateform

The simulator/model-checker presented in this paper is fully integrated to the CALIFE plateform. The CALIFE plateform is a framework for modeling and proving reactive timed systems. It works on several automata models (transition systems, timed automata, counter automata,...) and allows one to define new models and interface new tools. Several tools are currently interfaced with the plateform (UPPAAL [16], HyTech [9], KRONOS [10], CMC [5], COQ [6],...) and a unique timed-automata system modeled under the CALIFE System Editor can be exported to all these tools.

XML Automata. In the plateform, automata are described using the XML standard [17]. The key of interoperability between tools is to represent every predicate as an Abstract Syntax Tree (AST in short) where :
- the root is a Guard, Invariant, Updates or Activity node;
- the leaves are Variable, Parameter or Constant nodes;
- the intermediary nodes are associated with functions (but some usual operators are pretty-printed for flexibility reasons).

Of course, that mechanism is totally hidden to the user who defines automata in a graphical way. These syntactic automata definitions allow one to easily generate target code for interfaced tools. The process of the generation of target code can be slitted into two parts : generating the predicates in the target language and modeling the product of automata in a consistent way with respect to the semantics of synchronization in the tool.

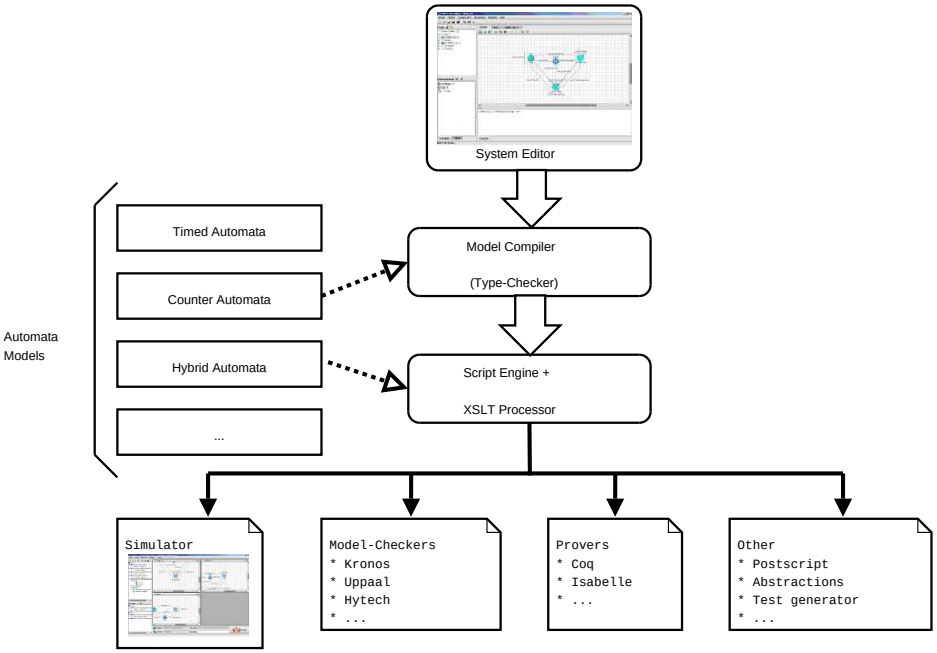

FIG. 1 – Overview of the CALIFE plateform

Translating predicates. That step is performed using an XSL transformation [18]. Since every predicate is represented as an AST, the corresponding code can be generated searching through the tree from the root to the leaves and associating a simple rule for code generation with every node encountered. These rules are defined using an "xsl :template" node [18].

Modeling the synchronized product. Several techniques can be used to generate consistent code :

- direct translation of the synchronization table for tools able to deal directly with synchronization tables (CMC [5], COQ [6] or the simulator/model-checker presented in this paper for example). Code defining the table can be generated using a very simple XSL transformation.
- translation using an injective function which associates a single label with every synchronization vector. Every transition is replicated for each instance of its label in the table. This technique is used for tools like KRONOS [10], HyTech [9] or the previous version of the model-checker developed in ELAN. Code can be generated using an XSL transformation.
- complex translation using XML pattern matching programming with the rewrite based Tom tool [15]. This technique is used to interface the UPPAAL tool [16] where new states must be introduced for every synchronization vector using more than one synchronization label. Complex abstractions are also made using this technique.

4 Updatable Timed Automata

Introduction. Updatable timed automata is a model for reactive systems defined in [4] which extends the timed automata model introduced by Alur and Dill [2].

Let X be a finite set of variables called *clocks*. A *clock valuation* is a mapping $v : X \to \mathbb{R}^+$, where $\mathbb{R}^+$ denotes the set of non-negative reals. The set of clock valuation is denoted $(\mathbb{R}^+)^X$. Given $t \in \mathbb{R}^+$, the valuation $v+t$ is defined by $(v+t)(x) = v(x)+t$ for all $x \in X$. When the cardinality of X is n, we also identify a valuation v with the point $(v(x_i))_{1 \leq i \leq n}$ of $(\mathbb{R}^+)^n$.

Clock constraints. The set of clock constraints denoted by $\mathcal{C}(X)$, is defined by the following grammar $\varphi ::= x \sim c | x - y \sim c | \varphi \wedge \varphi | true$ where $x, y \in X$, $c \in \mathbb{Q}$, $\sim \in \{<, \leq, =, \geq, >\}$, and $\mathbb{Q}$ is the set of rational numbers.
A *k-bounded clock constraint* is a clock constraint that involves only constants between $-k$ and $+k$.

Updates. An *update* is a function which assigns to each valuation a set of valuations. Updates are restricted as follows : a *simple update* over a clock z has one of the following forms $up ::= z :\sim c | z :\sim y + c$ where $c \in \mathbb{Q}$, $y \in X$ and $\sim \in \{<, \leq, =, \geq, >\}$.
Given a valuation v and a simple update up over z, a valuation v' is in $up(v)$ if $v'(y) = v(y)$ for all $y \neq z$, and if $v'(z)$ satisfies ($\sim \in \{<, \leq, =, \geq, >\}$) :

$$\begin{cases} v'(z) \sim c \text{ and } v'(z) \geq 0 & \text{if } up = z :\sim c \\ v'(z) \sim v(y) + d \text{ and } v'(z) \geq 0 & \text{if } up = z :\sim c + d \end{cases}$$

An *update over a set of clocks* X is a collection $up = (up_i)_{1 \leq i \leq k}$ where each up_i is a simple update over some clock $x_i \in X$. Given a valuation v, a valuation v' is in $up(v)$ if and only if, for all i, the clock valuation v_i defined by

$$\begin{cases} v_i(x_i) & = & v'(x_i) \\ v_i(y) & = & v(y) \quad \text{for } y \neq x_i \end{cases}$$

is in $up_i(v)$. The set of updates over the set of clocks X is denoted by $\mathcal{U}(X)$.

Updatable Timed Automata. An *updatable timed automata*[4] is a tuple $\mathcal{A} = (\Sigma, Q, T, I, F, X)$ where Σ is a finite alphabet of *actions*, Q is a finite set of states called *locations*, X is a finite set of clocks, $T \subset Q \times [\mathcal{C}(X) \times \Sigma \times \mathcal{U}(X)] \times Q$ is a finite set of transitions, $I \subset Q$ is the subset of initial states, and $F \subset Q$ is the subset of final states.
Timed Automata correspond to updatable timed automata where the only allowed atomic updates are of the form $up ::= z = 0$, $z \in X$ [2].

Runs. A *path* of $\mathcal{A}$ is a finite sequence of consecutive transitions $P = q_0 \longrightarrow^{\varphi_1, a_1, up_1} q_1 \ldots q_{p-1} \longrightarrow^{\varphi_p, a_p, up_p} q_p$ where $(q_{i-1}, \varphi_i, a_i, up_i, q_i) \in T$ for each $1 \leq i \leq p$.
The path is said to be *accepting* if it starts in an initial state ($q_0 \in I$) and ends in a final state ($q_p \in F$).
A *run* of the automaton through the path P is a sequence of the form $< q_0, v_0 > \longrightarrow_{t_1}^{\varphi_1, a_1, up_1} < q_1, v_1 > \cdots < q_{p-1}, v_{p-1} > \longrightarrow_{t_p}^{\varphi_p, a_p, up_p} q_p$ where $(t_i)_{1 \leq i \leq p}$ is finite non-decreasing sequence of non-negative reals, and $(v_i)_{1 \leq i \leq p}$ are clock valuations defined by

$$\begin{cases} v_0(x) = 0, \ \forall x \in X \\ v_{i-1} + (t_i - t_{i-1}) \text{ satisfies } \varphi_i \\ v_i \in up_i(v_{i-1} + (t_i - t_{i-1})) \end{cases}$$

Building automata from simpler ones. Given $\mathcal{A}_1 = (\Sigma_1, Q_1, T_1, I_1, F_1, X_1)$ and $\mathcal{A}_2 = (\Sigma_2, Q_2, T_2, I_2, F_2, X_2)$, and a set $Synch \subset (\Sigma_1 \cup \{-\}) \times (\Sigma_2 \cup \{-\})$ of *synchronization vectors*, the *product automata* $\mathcal{A}_1 \| \mathcal{A}_2$ is defined as $(\Sigma_1 \cup \Sigma_2, Q_1 \times Q_2, T, I_1 \times I_2, F_1 \times F_2, X_1 \cup X_2)$ where the set of transitions T is defined by

1. For $(a_1, a_2) \in Synch$, $a_1 \in \Sigma_1$, $a_2 \in \Sigma_2$, whenever $(q_1, \varphi_1, a_1, up_1, q_1') \in T_1$ and $(q_2, \varphi_2, a_2, up_2, q_2') \in T_2$, then $((q_1, q_2), a_1, \varphi_1 \wedge \varphi_2, up_1 \cup up_2, (q_1', q_2')) \in T$
2. For $(a_1, -) \in Synch$, whenever $(q_1, \varphi_1, a_1, up_1, q_1') \in T_1$, then for all $q_2 \in Q_2$, $((q_1, q_2), a_1, \varphi_1, up_1, (q_1', q_2)) \in T$
3. Symmetrically for $(-, a_2) \in Synch$.

Model-checking algorithms. A *zone* is a subset of $\mathbb{R}^n$ defined by a clock constraint. A *k-bounded zone* is a zone defined by a k-bounded clock constraint. When Z is a zone, we denote by $Approx_k(Z)$ the smallest k-bounded zone Z_k such that $Z \subset Z_k$.
If $e = q \longrightarrow^{\varphi,a,C:=0} q'$ is a transition of a timed automaton, then $Post(Z,e)$ denotes the set $[C \leftarrow 0](\varphi \cap Future(Z))$ where $Future(Z)$ represents the *future* of Z and is defined by $Future(Z) = \{v+t | v \in Z \text{ and } t \geq 0\}$. In other words, $Post(Z,e)$ is the set of valuations which can be reached by waiting in the current state q and then taking the transition e.
The algorithm implemented in our model-checker for testing reachability properties for timed automata (for e.g. in UPPAAL [16] and KRONOS [10], see the discussion in [4]) is the following[1] :

ZoneAlgorithm($\mathcal{A}$) :
  Compute k *the largest constant appearing in* $\mathcal{A}$;
  $Visited := \emptyset$;
  $Waiting := \{(q_0, Approx_k(Z_0))\}$;
  Repeat
    Get and Remove (q,Z) *from* $Waiting$;
    If q *is final then {Return "Yes" ;}*
      else { *if there is no* $(q,Z') \in Visited$ *such that* $Z \subset Z'$
        then { $Visited := Visited \cup \{(q,Z)\}$;
          $Successor := \{(q', Approx_k(Post(Z,e))) | e$ *transition from* q *to* $q'\}$;
        $Waiting := Waiting \cup Successor$;*}}*
  Until $(Waiting = \emptyset)$;
  Return "No" ; }

As proved in [4], a similar algorithm can be derived for updatable timed automata by replacing the approximation $Approx_k$ operator by a more general one.

DBMs. For timed automata as well as for updatable timed automata, zones can be represented efficiently by matrices, to get the so called *difference bounded matrices (DBM)* [7]. The operations needed on zones (computing $Future(Z)$, $[C \leftarrow 0](Z)$, $Z \cap Z'$ for given zones Z, Z') correspond to elementary manipulations on these matrices [1].

5 Model-checking algorithms implemented in ELAN

The ELAN code of the simulator consists mainly in rewrite rules that implement the previous algorithm *ZoneAlgorithm* for udpatable timed automata.

Representing zones. Observing the grammar of constraints, clock constraints and zones can be parsed directly using the following mix-fix ELAN signature :

```
true   :  clause  ;
false  :  clause  ;

@ <  @  :  ( clock int ) clause  ;
@ <= @  :  ( clock int ) clause  ;

@ - @ <= @  :  ( clock clock int ) clause  ;
@ - @ <  @  :  ( clock clock int ) clause  ;

@      :  ( clause ) clockzone ;
@ ^ @ :  ( clockzone clockzone ) clockzone  assocRight;
```

[1] Given a timed automaton $\mathcal{A}$ it tests if $\mathcal{A}$ has an accepting run by computing step-by-step an over-approximation of the set of reachable states and tests whether this approximation intersects the set of final states.

We also implemented difference bounded matrices [7] : difference bounded matrices correspond in the ELAN code to terms of sort $matrix[bound]$ where $bound$ is a sort for coding DBM entries, and $matrix$ is the built-in module of ELAN system for dealing with matrices.
The pairs of type (q, Z) of algorithm $ZoneAlgorithm$ correspond then to terms of the form q/z where q is a list of locations, and z is a term coding the zone Z using the previous signature or a DBM.

Operations on zones. To implement $ZoneAlgorithm$, the main operations to be realized on zones are the operators that map a zone Z to $Future(Z)$, $[C \leftarrow 0](Z)$ and $Approx_k(Z)$ respectively. Unlike previous prototype where these operations where realized using a Fourier-Motzkin like algorithm [3], these operations are now realized by working on DBMs, transforming constraints to DBM if needed.
In the same spirit, the classical Floyd-Warshal algorithm for computing normal form of DBMs [7, 1] as well as the test of vacuity of a DBM are also fully written by ELAN rewrite rules.

```
// ELAN Signature of main operations on DBMs
TimeDBM(@): (matrix[bound]) matrix[bound]; //Future
@-ApproxDBM(@): (int matrix[bound]) matrix[bound];//Approx_k
InterDBM(@,@): (matrix[bound] matrix[bound]) matrix[bound];
             //Intersection
Project(@,@): (matrix[bound] list[clock]) matrix[bound]; //[C<-0]
Floyd(@,@): (matrix[bound] int) matrix[bound] ; //Canonical Form
IsEmptyDBM(@,@): (matrix[bound] int) bool; //Empty?
```

Transcription of Automata The previous rules are independant of the automata given as input. For rules dependant of the (updatable) timed automaton given as input, the rules are generated using the preprocessor of the ELAN system.
For example, $2n + 1$ named rules and a strategy of ELAN strategy language are used for making an automaton corresponding to a product of n automata do a transition, using the following ELAN code :

```
// Transcription of a synchronization product
FOR EACH N : Int SUCH THAT N:=()valueOf(size_of_Automaton_list(LA)):{
rules for can_sz
  {s_I : state ;}_I=1...N
  {ss_J : state ;}_J=1...N
  Phi, nPhi       : matrix[bound] ;
  llbl            : list[label] ;
  lbl             : label;
  cansz              : can_sz ;
 global
{[r1s_i] DTs(llbl,{s_j.}_j=1...(i-1) s_i.{s_j.}_j=(i+1)...N nil/Phi) =>
        DTs(llbl,{s_j.}_j=1...(i-1) ss_i.{s_j.}_j=(i+1)...N nil/nPhi)
             where lbl:=()i-th elem(llbl)
             if not(eq_label(NOMOVE,lbl))
             where cansz:=()TransitionOperator.lbl(s_i/Phi)
             where ss_i:=()st(cansz)
             where nPhi:=()zn(cansz)
             end
  [r2s_i] DTs(llbl,cansz) => DTs(llbl,cansz) end
}_i=1...N
  [r3s]   DTs(llbl,cansz) => cansz end
end // of rules for can_sz
```

```
}
```

With the following strategy built using the *first one* operator (this ELAN strategy operator applied on a term t returns the result of the first strategy among its arguments on t that does not fail) :

```
strategies for can_sz
 implicit
 FOR EACH N : Int SUCH THAT N:=()valueOf(size_of_Automaton_list(LA)) :{
  [ ] next_sz => {first one(r1s_I,r2s_I);}_I=1...N  r3s end
 }
```

Breadth-First Exploration The first model-checker written in ELAN was implementing a Depth-First exploration of the zone graph. In our case, the goal is to build a model-checker which can be used as a symbolic simulator. Depth-First exploration does not ensure that the trace returned is the shortest. That's why, we implemented a breadth-first exploration compliant to the model-checking algorithm *ZoneAlgorithm*.
From an implementation point of view, the list of visited states is stored using a hashtable principle. This technique divides the time needed for exploration by 3. Requiring the compiled code to use the *aterm library* [13] which allows maximal subterm sharing and automatic garbage collection allows one to divide further the time for exploration by 2.

On-the-fly model-checking. Our implementation using a rewrite engine and our transcription of automata transitions into rewriting rules offers a natural way to implement *On-the-fly* model-checking. This means that the synchronized automata are not statically built before starting to reach the final state, but is constructed on running time and only when needed.

Queries The different queries for reachability that are offered by the tool are the following (notice that wild-cards are authorized when writing starting and final subsets).

- **breadth-First***(starting subset, final subset)*, where starting subset and final subset are of the form s/c where s is a list of locations (with possibly wildcard $*$) and c is a zone. This query returns an execution trace if some state of the final subset is reachable from a state of the starting subset, or *unreachable* otherwise.
- **breadthFirstExploration1s***(starting subset,final subset)* does an exploration similar to the previous query but without returning a trace : it returns only *reachable* or $non - reachable$.
- **ReachableIn-i-stepsFrom***(starting subset)* returns all the traces corresponding to states reachable in exactly i-steps from starting subset ; it returns that the exploration is exhaustive in less than i-steps otherwise. Similarly to the two previous queries, it is implemented using a breadth-first exploration in the spirit of *ZoneAlgorithm*.
- **DsbreadthFirst***(starting subset)* seeks for a deadlock starting from starting subset, and returns a trace that reaches it if there is one. It returns that there is no deadlock otherwise. It is also implemented using a breadth-first exploration.
- **ReachInOneStep***(starting subset, final subset)* returns *true* if some state of final subset can be reach in one step from a starting state, $false$ otherwise

6 The CALIFE Simulator

6.1 Driving ELAN from CALIFE

Generating the ELAN executable. As other tools integrated in the CALIFE plateform, the ELAN simulator is connected using an XML script defined inside a model definition (associated to an automata class like Timed Automata, Extended Timed Automata or Updatable Timed Automata). The XML script executed by the CALIFE Script Engine is in charge of :

– generating the ELAN specification : this step is performed using a simple XSL[18] transformation directly translating the XML tree in a flat textual form. The use of a synchronization table in the ELAN tool simplifies a lot that step by avoiding to calculate synchronization labels or to abstract the system in an equivalent form
– compiling the ELAN executable : in order to have the best performances during model-checking verification, we choose to drive a C executable generated by the ELAN compiler instead of using the ELAN interpreter. To avoid useless compilations (which spend about 1 minute), we calculate a magic number from the specification (using a CRC32) in order to determine when a compilation phase is necessary.

Driving the ELAN executable. In order to make the graphical interface independant of ELAN term forms, we define a Java interface defining a generic exchange protocol for the simulator engine (SimulatorEngine.java).
Any Java class following this interface must implement the following methods :
– public Document getNextStatesMultitracesFrom(Node xmlZone) ;
– public Document findaDeadlockFrom(Node xmlZone) ;
– public Document reachState(Node fromxmlZone, Node toxmlZone) ;
As illustrated in the previous Java function signatures, we use XML (which is generalized within the CALIFE plateform) as a data exchange format between the simulator engine and the GUI. The java functions are in charge of executing the simulator engine and translating the results in XML.

XML representation of an execution trace. The XML grammar used to represent Zones (StateSync) and Traces is defined by the following DTD[17][2] :

```
<!ELEMENT Trace (StateSync,(Action,StateSync)*)>
<!ATTLIST Trace Label CDATA #REQUIRED>
  <!ELEMENT StateSync ((State)*,Diag,Glob)>
    <!ELEMENT State (Loc,Bounds)>
    <!ATTLIST State Id CDATA #REQUIRED>
    <!ATTLIST State Component CDATA #REQUIRED>
      <!ELEMENT Loc EMPTY>
      <!ATTLIST Loc Label CDATA #REQUIRED>
      <!ELEMENT Bounds (Bound)*>
    <!ELEMENT Diag (Bound)*>
    <!ELEMENT Glob (Bound)*>
        <!ELEMENT Bound EMPTY>
        <!ATTLIST Bound Value CDATA #REQUIRED>
```

From a semantical point of view,
– a Bound node defines an atomic clock constraint (cf Section 4).
– a Bounds node contains a sequence of Bound and is local to an automaton (non-diagonal constraints involving only local variables).
– a Glob node contains a sequence of Bound associated to non-diagonal constraints involving global (shared) variables.
– a Diag node contains all diagonal constraints.
An XML trace is made by a sequence of StateSync nodes (defining a state zone) and Action nodes (defining the label of the synchronization vector used to reach the next state).

6.2 A Graphical User Interface for debugging specifications

Overview of the simulator As illustrated by the figure below, the GUI is made of 4 parts showing :

[2] Document Type Definition

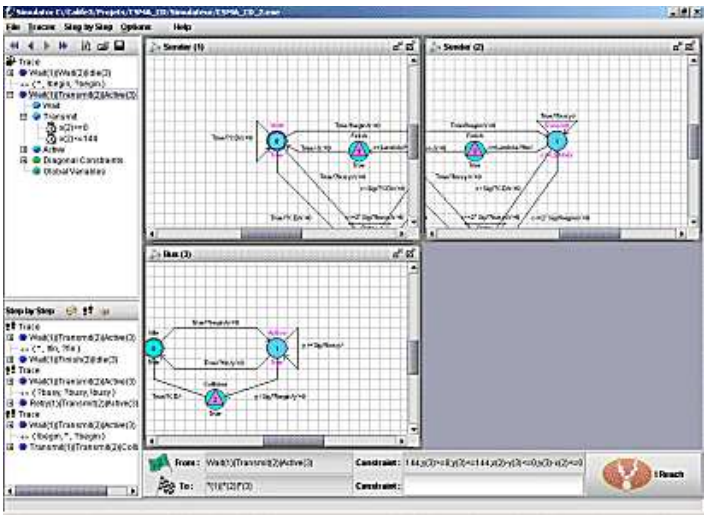

FIG. 2 – Overview of the simulator

- In the central part, a view of all the components defined in the synchronized product. These views are always centered on the current location of all the graphs.
- In the upper-left part of the window, the current execution trace built by the user. Initially, this trace only contains the initial state. The trace is presented as a tree which can be expanded to inspect clock valuations.
- In the lower-left part, all the states reachable in " one step " from the current state (modulo time elapse).
- The bottom part is used to easily construct reachability requests on the model. That can be done by clicking on locations in the graphs associated to automata and (optionally) adding a specific clock constraint.

Step by Step debugging. The first main functionality of a symbolic simulator over automata is to be able to run a specification "Step by Step". For every state reached, a "Reachin-1-Step" term is built and rewritten by the ELAN program in a list of states to present the next states list. That kind of functionality is available in the UPPAAL [16] tool which deals with Timed Automata with bounded variables. Our tool allows one to add non-deterministic updates (to simulate an algorithm with an input between a lower and a upper bound for example).

A breakpoint as a reachability formula. In the simulator GUI, the user can also create a reachability request by simply clicking on locations and adding a constraint over clocks and variables that the reached state must satisfy. A "breadth-First" term is built and rewritten by the ELAN program in an execution trace (playable for the simulator) or a *UNREACHABLE_STATE* term.

Mixing Step by Step and Reachability allows the user :

- to run the system to a specific execution point (breakpoint) and then to inspect the specification step by step.
- to prove formula like *When that state is reached, the system has no deadlock* or *that state is unreachable* , ...
- to orient the model-checker for proving reachability formula in cases where direct model-checking is impossible because of a combinatorial explosion. The user can construct the beginning of a potential execution trace and then try to reach the state from that intermediary point.

7 Conclusion

In this paper we presented a tool fully written using the ELAN rewrite system, and fully integrated to the CALIFE plateform that provides first a model-checker with functionalities available nowhere else (for example the possiblity of model-checking updatable timed automata), and second a simulator that provides a graphical tool for simulating products of (extended) timed automata in the CALIFE plateform with unique features.

Furthermore, if one considers that the whole system was fully developed in ELAN in five months, by a non rewrite-system expert who neither know rewriting nor ELAN system before this exercise, it can be observed that this is also a demonstration of the gain offered by the use of a rewrite system such as ELAN for realizing quickly powerful prototypes.

The full code can be found at http ://calife.criltechnology.com in the CALIFE plateform downloadable area.
From the modeling and verification point of view, with

1. its full integration into the CALIFE plateform,
2. the connection of the CALIFE plateform with all tools for timed automata,
3. the new resulting simulation facilities,
4. in particular with the flexibility offered in the queries,

we believe that the CALIFE plateform with our simulator is one of the most powerful tool for practical verification and simulation of timed automata.

Références

[1] R. Alur. Timed automata. In *NATO-ASI Summer School on Verification of Digital and Hybrid Systems*, 1998.

[2] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *Automata, Languages and Programming, 17th International Colloquium*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 16–20 July 1990.

[3] Emmanuel Beffara, Olivier Bournez, Hassen Kacem, and Claude Kirchner. Verification of timed automata using rewrite rules and strategies. In Nachum Dershowitz and Ariel Frank, editors, *Proceedings BISFAI 2001, Seventh Biennial Bar-Ilan International Symposium on the Foundations of Artificial Intelligence*, Ramat-Gan, Israel, June 25–27, 2001.

[4] Patricia Bouyer. Updatable timed automata, an algorithmic approach. Technical report, LSV, 2001. Available at http ://www.lsv.ens-cachan.fr/Publis.

[5] CMC. *CMC*. Available at http ://www.lsv.ens-cachan.fr/fl/cmcweb.html.

[6] COQ. *COQ*. Available at http ://coq.inria.fr.

[7] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. volume 407 of *Lecture Note in Computer Science*, pages 197–212. Springer-Verlag, 1989.

[8] ELAN. *ELAN*. Available at http ://elan.loria.fr/.

[9] HYTECH. *HyTech*. Available at http ://www-cad.eecs.berkeley.edu/tah/hytech/.

[10] KRONOS. *Kronos*. Available at http ://www-verimag.imag.fr/TEMPORISE/kronos/.

[11] Pierre-Etienne Moreau. REM (Reduce Elan Machine) : Core of the new ELAN compiler. In *Proceedings 11th Conference on Rewriting Techniques and Applications, Norwich (UK)*, volume 1833 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 2000.

[12] Pierre-Etienne Moreau and Hélène Kirchner. A compiler for rewrite programs in associative-commutative theories. In *"Principles of Declarative Programming"*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, September 1998. Report LORIA 98-R-226.

[13] Pierre-Etienne Moreau and Olivier Zendra. GC2 : A Generational Conservative Garbage Collector for the ATerm Library, 2004. To appear in Journal of Logic and Algebraic Programming.

[14] Bertrand Tavernier. Calife : a generic graphical user interface for automata tools. In *Fourth Workshop on Language Descriptions, Tools and Applications (LDTA 2004)*, April 2004.

[15] TOM. *TOM.* Available at http ://tom.loria.fr/.

[16] UPPAAL. *UPPAAL.* Available at http ://www.uppaal.com/.

[17] XML. *Extensible Markup Language (XML) 1.0. W3C Recommendation*, October 6 2000. http ://www.w3.org/TR/2000/REC-xml-20001006.

[18] XSL. *Extensible Stylesheet Language (XSL) Version 1.0. W3C Recommendation*, October 15 2001. http ://www.w3.org/TR/xsl/.