Verification of Timed Automata Using Rewrite Rules and Strategies

Emmanuel Beffara[2], Olivier Bournez[1], Hassen Kacem[1], and Claude Kirchner[1]

[1] LORIA & INRIA, 615 rue du Jardin Botanique, BP 101, 54602 Villers-lès-Nancy Cedex, Nancy, France.
{Olivier.Bournez,Hassen.Kacem,Claude.Kirchner}@loria.fr
[2] ENS-Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07, France.
Emmanuel.Beffara@ens-lyon.fr

Abstract. ELAN is a powerful language and environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. Timed automata is a class of continuous real-time models of reactive systems for which efficient model-checking algorithms have been devised. In this paper, we show that these algorithms can very easily be prototyped in the ELAN system.
This paper argues through this example that rewriting based systems relying on rules *and* strategies are a good framework to prototype, study and test rather efficiently symbolic model-checking algorithms, i.e. algorithms which involve combination of graph exploration rules, deduction rules, constraint solving techniques and decision procedures.

1 Introduction

ELAN is a powerful language and environment for specifying and prototyping deduction systems in a language based on rewrite rules controlled by strategies. It offers a natural and simple framework for the combination of the computation and the deduction paradigms. The logical and semantical foundations of the ELAN system rely respectively on rewriting logic [19] and rewriting calculus [12] and are in particular described in [9, 11].

Timed automata [4] is a particular class of hybrid systems, i.e. systems consisting of a mixture of continuous evolutions and discrete transitions. They can be seen as automata augmented with clock variables, which can be reset to 0 by guarded transitions of some special type. They have proven to be a very useful formalism for describing timed systems, for which verification and synthesis algorithms exist [1, 4], and are implemented in several model-checking tools such as TIMED-COSPAN [5], KRONOS [14] or UPPAAL [18].

This paper describes our experience using the ELAN system to prototype the reachability verification algorithms implemented in the model-checking tools for timed automata.

It is known that rewriting logic is a good framework for unifying the different models of discrete-time reactive systems [19]. Rewriting logic can be extended to

deal with continuous real-time models. Such an extension, called "Timed rewriting logic" has been investigated, and applied to several examples and specification languages [17, 25, 26]. In this approach the time is somehow built in the logic. Another approach is to express continuous real-time models directly in rewriting logic. This has been investigated in [22, 23] and recently Olveczky and Meseguer have conceived "Real-Time Maude" which is a tool for simulating continuous real-time models [24].

Our approach is different. First, we do not intend to conceive a tool for *simulating* real-time systems, but for *verifying* real-time systems. In other words, we do not intend to prototype real-time systems but to prototype verification algorithms for real-time systems.

Second we focus on *Timed Automata*. Since verification of hybrid systems is undecidable in the general case [2], any verification tool must restrict to some decidable class of real-time systems, or must be authorised to diverge for some systems. Timed automata is a class of continuous real-time systems which is known to be decidable [4]. Real-Time Maude falls in the second approach in the sense that the "find" strategy implemented in this tool gives only *partially* correct answers [24].

The implemented model-checking algorithms for timed automata are typical examples where the combination of exploration rules, deduction rules, constraint solving and decision procedures are needed. One aim of this work is to argue and demonstrate through this example that the rewriting calculus is a natural and powerful framework to understand and formalise combinations of proving and constraint solving techniques.

Another aim is to argue the suitability advantages of using a formal tool such as ELAN to specify and prototype a model checking algorithm compared to doing it in a much cumbersome way using a conventional programming language. First this allows a clear flexibility for customisation not available in typical hard-wired model checkers, through for example programmable strategies. Second, using the efficient ELAN compiler, the performances are indeed close to dedicated optimised model-checking tools.

This paper is organised as follows. In Section 2, we describe the ELAN system based on rewriting calculus. In Section 3, we recall what a timed automaton is. In Section 4, we describe our tool for verifying reachability properties of product of timed automata. In Section 5, we discuss the implementation.

2 The ELAN system

The ELAN system 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 labelled 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.

In general, the normalisation of a term may not terminate, or terminate with different results corresponding to different selected rules, selected sub-terms or non-unicity of the substitution σ. So evaluation by rewriting is essentially non-deterministic and backtracking may be needed to generate all results.

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. This is in contrast to many existing rewriting-based languages where the term reduction strategy is hard-wired and not accessible to the designer of an application. The strategy language offers primitives for sequential composition, iteration, deterministic and non-deterministic choices of elementary strategies that are labelled rules. From these primitives, more complex strategies can be expressed, and new strategy operators can be introduced and defined by rewrite rules.

The full ELAN system includes a preprocessor, an interpreter, a compiler, and standard libraries available through the ELAN web page[1]. From the specific techniques developed for compiling strategy controlled rewrite systems [**?**,21], 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.

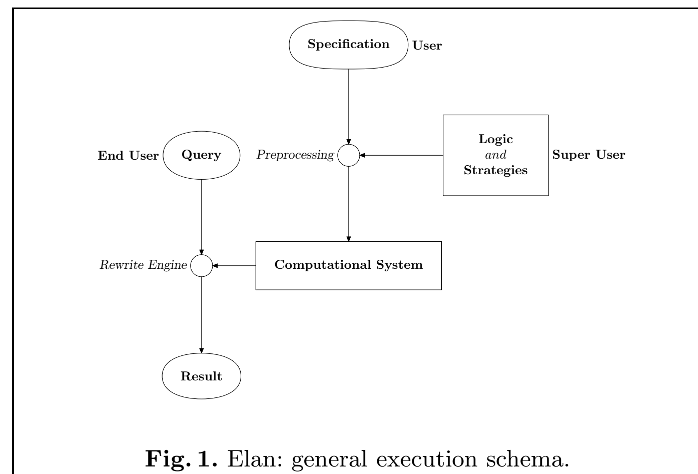

Fig. 1. Elan: general execution schema.

[1] `http://elan.loria.fr`

3 Timed automata

A *clock* is a variable which takes value in the set $\mathbb{R}^+$ of non-negative real numbers. A *clock constraint* is a conjunction of constraints of type $x\#c$ or $x - y\#c$ for some clocks x, y, rational number $c \in \mathbb{Q}$ and $\# \in \{\leq, <, =, >, \geq\}$. Let $TC(K)$ denotes the set of clock constraints over clock set K.

Informally, a *Timed Automaton* is a finite automaton augmented with clock variables, which can be reset to 0 by guarded transitions.

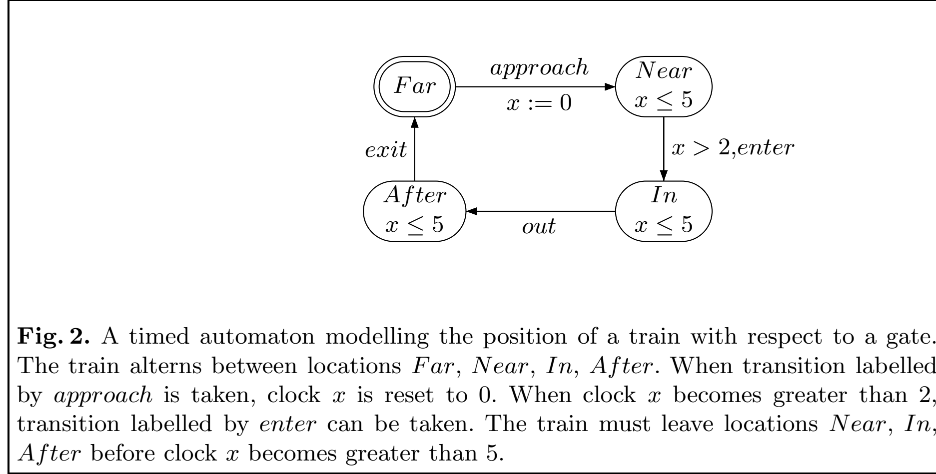

Fig. 2. A timed automaton modelling the position of a train with respect to a gate. The train alterns between locations Far, $Near$, In, $After$. When transition labelled by *approach* is taken, clock x is reset to 0. When clock x becomes greater than 2, transition labelled by *enter* can be taken. The train must leave locations $Near$, In, $After$ before clock x becomes greater than 5.

Formally, a *timed automaton* [4, 1] is a 5-tuple $\mathcal{A} = (\Sigma, L, K, I, \Delta)$ where

1. Σ is a finite alphabet,
2. L is a finite set of *locations*,
3. K is a finite set of clocks. A *state* is given by some location and some valuation of the clocks, i.e. by some element (s, v) of $L \times \mathbb{R}^{+|K|}$.
4. I is a function from L to $TC(K)$ that labels each location s by some *invariant* $I(s)$. Invariant $I(s)$ restricts the possible values of the clocks in location s.
5. Δ is a subset of $\Sigma \times L \times TC(K) \times \mathcal{P}(K) \times L$. *Transition* $(a, s, c, z, s') \in \Delta$ corresponds to a transition from location s to location s', labelled by a, guarded by constraint c on the clocks, and which resets the clocks $k \in z$ to 0.

Timed automaton $\mathcal{A}$ evolves according to two basic types of transitions:

1. *Delay transitions* corresponds to the elapsing of time while staying in some location: write $(s, v) \longrightarrow^d (s, v')$, where $d \in \mathbb{R}^+$, $v' = v + (d, \ldots, d)$ provided for every $0 \leq e \leq d$, state $v + (e, \ldots, e)$ satisfies constraint $I(s)$.
2. *Action transitions* corresponds to the execution of some transition from Δ: write $(s, v) \longrightarrow^a (s', v')$, for $a \in \Sigma$, provided there exists $(a, s, c, z, s') \in \Delta$ such that v satisfies constraint c and $v'_k = v_k$ for $k \notin z$, $v'_k = 0$ for $k \in z$.

A *trajectory* of $\mathcal{A}$ starting from (s, v) is a sequence

$$(s, v) \longrightarrow^{e_0} (s_1, v_1) \longrightarrow^{e_1} (s_2, v_2) \dots$$

for some $e_0, e_1, \dots \in \Sigma \cup \mathbb{R}^+$.

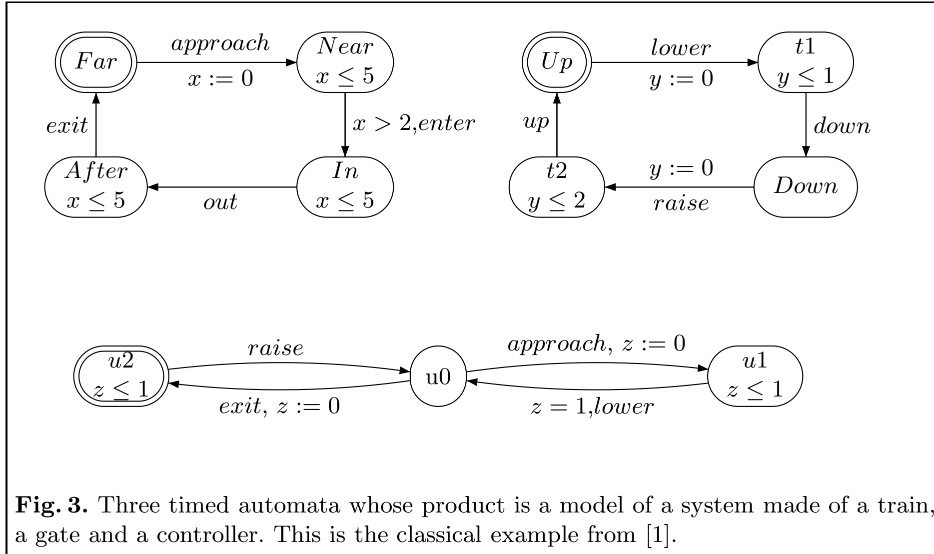

Fig. 3. Three timed automata whose product is a model of a system made of a train, a gate and a controller. This is the classical example from [1].

Product construction. Timed automata can be composed by *synchronisation* [1, 4] (see figure 3 for a classical example). Intuitively, building the product of two timed automata consists in considering a timed automaton whose state space is the product of the state spaces of the automata, where transitions labelled by a same letter in the two automata occur synchronously, and others may occur asynchronously.

4 The tool

Our verification system for timed automata is fully implemented in ELAN. It works according to the schema of Figure 4. Concretely,

1. The tool takes a specification of a product of automata. The specification is given in the form of a text file containing a list of clocks, a list of locations, a list of labels, and a list of automaton descriptions. Each automaton description is in turn a list of list of locations, labels, invariants, transitions: cf Figure 5.

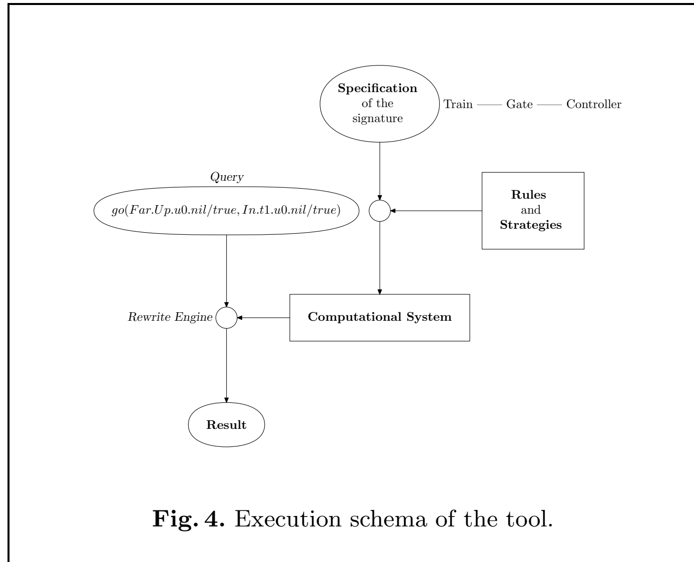

Fig. 4. Execution schema of the tool.

2. The specification is then parsed using the ELAN system, and compiled into an executable program. More precisely, the ELAN preprocessor, manipulates the encapsulated lists of the specification through ELAN rules in order to generate rewrite rules, which are in turn compiled by the ELAN compiler into an executable C program.
3. This C program tests reachability properties of the product of automata:
   (a) it takes as input a query of type $go(s/c, s'/c')$ where s, s' are some locations of the product of automata, and c, c' are some clock constraints.
   (b) it answers $True$ iff there is a trajectory starting from some state (s, v) with v satisfying clock constraint c that reaches some (s', v') with v' satisfying clock constraint c'.

Figure 6 shows some queries and their execution times for the $train \parallel gate \parallel controller$ system, and for the system described in [15] consisting of two robots, a conveyer belt and a processing station[2]. The execution times of system *KRONOS* are shown for comparison. Observe that they are of same magnitude.

5 Implementation

We now describe how the reachability algorithm is implemented using rewrite rules controlled by strategies. We need first to recall the basis of timed automata theory.

[2] See web page `http://www.loria.fr/~kacem/AT` for details.

```
specification train
  Clocks
    X Y Z
    nil
  States
    Far Near In After Up t1 Down t2 u0 u1 u2
    nil
  Labels
    app raise lower up down enter out exit
    nil
  Automata
    (
    Locations
      Far Near In After
      nil
    Labels
      app enter out exit
      nil
    Invariants
      Far   :        true
      Near  : X<=5 ^ true
      In    : X<=5 ^ true
      After : X<=5 ^ true
      nil
    Transitions
      Far  , app  :        true, X nil, Near  .
      Near , enter: X>2  ^ true,   nil, In    .
      In   , out  :        true,   nil, After .
      After, exit :        true,   nil, Far   .
      nil
    ) .
(
    Locations
      Up t1 Down t2
      nil
    Labels
      lower down raise up
      nil
    Invariants
      Up   :        true
      t1   : Y<=1 ^ true
      Down :        true
      t2   : Y<=2 ^ true
      nil
    Transitions
      Up  , lower :        true, Y nil, t1   .
      t1  , down  :        true,   nil, Down .
      Down, raise :        true, Y nil, t2   .
      t2  , up    :        true,   nil, Up   .
      nil
    ) .
    (
    Locations
      u0 u1 u2
      nil
    Labels
      app lower raise exit
      nil
    Invariants
      u0 : true
      u1 : Z<=1 ^ true
      u2 : Z<=1 ^ true
      nil
    Transitions
      u0, app   :               true, Z nil, u1 .
      u1, lower : Z>=1 ^ Z<=1 ^ true,   nil, u0 .
      u0, exit  :               true, Z nil, u2 .
      u2, raise :               true,   nil, u0 .
      nil
    ) .
    nil
end
```

Fig. 5. The specification of the system of Figure 3. Each automaton is described by lists of locations, labels, invariants and transitions. The product is in turn described by a list of automata.

Query	Answer	Kronos	Elan
$Train$: $go(Far.Up.u0.nil/true, In.Down.u0.nil/true)$	$True$	0.03 seconds	0.00 seconds
$Train$: $go(Far.Up.u0.nil/true, In.Up.u0.nil/true)$	$False$	0.03 seconds	0.03 seconds
$Robots$:$go(D-Wait.G-Inspect.S-Empty.B-Mov.nil/true$ $, D-Turn-L.G_Inspect.S-Empty.B-Mov.nil/true)$	$True$	0.04 seconds	0.03 seconds
$Robots$: $go(D-Wait.G-Inspect.S-Empty.B-Mov.nil/true,$ $D-Turn-R.G-Turn-L.S-Empty.B-On-S.nil/true)$	$False$	0.04 seconds	0.14 seconds

Fig. 6. Some queries and their execution times (performed on a PC PIII 733 MHz 128 Mo).

5.1 Region automaton

Recall that a *labelled transition system* is a tuple $(Q, \Sigma, \rightarrow)$ where Q is set of states, Σ is some finite alphabet, and $\rightarrow \in Q \times \Sigma \times Q$ is a set of transitions.

Given some timed automaton $A = (\Sigma, L, K, I, \Delta)$, denoting $(s, v) \leadsto^a (s', v')$ if there exists s'' and v'' such that $(s, v) \longrightarrow^d (s'', v'') \longrightarrow^a (s', v')$ for some $d \in \mathbb{R}^+$, the *time-abstract labelled transition system* associated to A is the labelled transition system $S(A) = (Q_A, \Sigma, \leadsto)$ whose state space Q_A is unchanged, i.e. $Q_A = L \times {\mathbb{R}^+}^{|K|}$, but whose transition relation is given by $\leadsto$.

Although $S(A)$ has uncountably many states, we can associate some equivalence relation $\sim_A$ over the state space Q_A which is stable and which is of finite index [1, 3]. Some equivalence relation $\sim$ over the space of a labelled transition system $(Q, \Sigma, \rightarrow)$ is said to be *stable* iff whenever $q \sim u$ and $q \rightarrow^a q'$ there exists some u' with $u \rightarrow^a u'$ and $u' \sim u$.

The *quotient of* $S(A)$ *with respect to* $\sim_A$, denoted by $[S(A)]$, is the transition system whose state space is made of the equivalence classes of $\sim_A$, called *regions*, and such that there is a transition from region π to region π' labelled by a if for some some $q \in \pi$ and $q' \in \pi'$, $q \leadsto^a q'$.

Since $\sim_A$ is of finite index, $[S(A)]$ is a finite automaton, which is called *the region automaton* of A [1, 3].

Since $\sim_A$ is stable, the set of reachable states from some region s_0 in timed automaton A is equal to the union of the reachable regions in $[S(A)]$ starting from region s_0 [1, 3]. Hence, the reachability problem for A reduces to the reachability problem for finite automaton $[S(A)]$.

This is the basis of all model-checking tools for timed automata. See [1, 4] for details.

5.2 Manipulating regions using states with constraints

Computing the reachable regions in region automaton $[S(A)]$ requires to manipulate regions.

This is can be done by manipulating symbolic representations of these sets, i.e. by manipulating clock constraints [1, 4]. Hence, our program in ELAN manipulates terms of form s/c where s, c is a (term representation of) a state of the product of automata, and c is a clock constraint. Term s/c, represents the (convex) set, denoted by $[[s/c]]$, of all the states $(s, v) \in Q_A$, such that v satisfies constraint c. Such a set is called a *zone* [1, 4].

The heart of the reachability algorithm in ELAN is made of *rewrite* rules which manipulate such terms through symbolic operators on constraints. Figure 7 shows an example of such a rule for the system of Figure 5. This rewrite rule uses "Intersection" and "Reset" operators on clock constraints.

The *Intersection* operator transforms two clock constraints into a representation of their intersection. The *Reset* operator transforms a constraint c, and a list of clocks k, to a constraint representing the set of states reachable from a state satisfying c after the variables of k are reset to 0.

```
[] Post.enter(Near/c) => In /Reset(Intersection(X>2 ^ true,c), nil) end
```

Fig. 7. A rewrite rule manipulating a zone, using 'Intersection" and "Reset" operators on clock constraints.

5.3 Generation of rules from the automaton specification

These rules on zones are generated from the description of the timed automaton given as input using the preprocessor facilities of the ELAN system.

For example, for any transition $e = (a, s, d, z, s')$ of the timed automaton, we need to generate a rewrite rule which transforms any zone s/c into some zone s'/c' which represents all the states reachable from s/c by transition e. Formally, we want to rewrite s/c into s'/c' with

$$[[s'/c']] = \{(s', v')|(s, v) \in [[s/c]] \text{ and } (s, v) \longrightarrow^{a} (s', v')\}$$

This can be done by generating rewrite rule

$$post.e(s/c) \Rightarrow s'/Reset(Intersection(c, d), z)$$

In order to generate this rewrite rule for any transition e of the automaton specification, we just need in the ELAN system, to write the preprocessor rule of Figure 8.

```
FOR EACH A : Automaton SUCH THAT A:=(listExtract) elem(LA):{
rules for statezone
  z : clockzone ;
 global
  FOR EACH tr       : transition ;
           bef, aft : state ;
           label    : label ;
           cond     : clockzone ;
           zero     : list[clock]
  SUCH THAT tr   := (listExtract) elem(lst_trans(A))
        AND bef  :=()tr_before(tr)
        AND label:=()tr_label (tr)
        AND cond :=()tr_cond  (tr)
        AND zero :=()tr_zero  (tr)
        AND aft  :=()tr_after (tr) :
  {
    []  Post.label(bef/z) => aft / Reset(Intersection(cond,z),zero) end
  }
end // of rules for statezone
```

Fig. 8. The preprocessor rule which generated the rule of Figure 7 for the transition from location *Near* to *In* of the system of Figure 3.

5.4 Constraint representation

Implementing the operators on constraints requires to represent clock constraints.

One solution consists in representing clock constraints using *bounded differences matrices* [1, 16]. With this representation scheme, any clock constraint has a normal form which can be computed using a Floyd-Warshal algorithm based technique [1].

This method using bounded differences matrices has been implemented as a rewrite system in our tool. That means in particular that the Floyd-Warshal based technique for computing normal forms of bounded differences matrices is implemented as rewrite rules, as well as all operators required on constraints ($Intersection$, $Reset$, $Is_Empty?$, $Is_Equivalent?$, $Effect_Of_Time_Elapsing$).

Constraint representation alternatives. Other representations of constraints are possible. In particular, we have experimented the representation of clock constraints using classical logical formulas. Clock constraints are closed by quantifier elimination [7, 13]. Denoting by $Exists$ the operator which maps a clock constraint c and a list of variable k to a clock constraint logically equivalent to formula $\exists k\ c$, the above operators on constraints can all be expressed using the $Exists$ operator [7, 13]. For example, the $Reset$ operator can be expressed by the rewrite rule

$$Reset(c, k) \Rightarrow Exists(c, k) \wedge k = 0.$$

This as been implemented in our tool. The $Exists$ operator is computed using a technique based on Fourier-Motzkin algorithm [7].

5.5 Exploration

Once the constraint manipulation is implemented, one interesting part is to implement the exploration of the reachable regions of the automaton. This is done by manipulating terms of the form $Transitions_List(lsz, szs, szc)$ where lsz is a list of already explored zones, szs is the current zone under investigation, and zsc is the objective zone.

One main originality of ELAN is the possibility to express strategies. Hence, the exploration of graph can be guided by the simple strategy language of the ELAN system.

As an example, suppose we want to explore the graph by backtracking. Rule named $SuccessStep$ of Figure 9 does one step of the graph exploration for the particular case when the objective zone is reachable by one step of the graph exploration, and rule named $NextStep$ does one step of the graph exploration for the generic case.

To explore the whole graph, we just need to iterate these rules, taking at each step the first one of the two elementary rules $SuccessStep$ and $NextStep$ which succeeds. This is easily done using the ELAN strategy language as in Figure 10.

Exploration alternatives. Of course, other exploration strategies can also be used and experimented just by modifying the above lines. Graph can easily be traversed depth first, breath first, if one prefers.

```
rules for bool
  s,s0 : state;
  sz      : zone;
  c,c0 : clockzone;
  result  : bool;
  lsz : hashSet[statezone];
global
[SuccessStep] Transitions_List(lsz,s/c,s0/c0) => result
                  where sz := (Post) s/c
                  if sz.state== s0
                  if not Is_Empty?(Intersection(sz.constraint,c0))
                  where result:=() True
                  end
[NextStep]    Transitions_List(lsz,s/c,s0/c0) => result
                  where sz := (Post) s/c
                  if not Is_Empty?(sz.constraint)
                  if not lsz.contains(sz)
                  where result:=(Exploration) Transitions_List(lsz.add(sz),sz,s0/c0)
end
```

Fig. 9. Making one step of the exploration.

```
strategies for bool
 implicit
  []  Exploration => first one(SuccessStep,NextStep) end
  end


rules for bool
  s,s0: state;
  c,c0: clockzone;
  result: bool;
global
[] go(s/c,s0/c0) => result
               choose
                 try where result:=(Exploration) Transitions_List(EmptySet,s/c,s0/c0)
                 try where result:=()False
               end
end
```

Fig. 10. Exploring the whole graph.

5.6 On-fly generation.

The tool implements *on-fly model-checking*. This means that the tool does not need to build the full product of the timed automata before testing reachability properties, but that the transitions of the product of timed automata are generated on-line only when needed. This is in contrast with what happens in some model-checking tools.

This is easily done, in the case of an input consisting of a product of n timed automata, by using a succession of n ELAN $first_one$ strategy operators applied on named rules $ExecuteTransition_i$ and $JumpStep_i$ which compute on-line the transitions to apply in each automaton of the product corresponding to some possible label: cf Figure 11.

```
// Transcription Of The Synchronised Product
FOR EACH N : int SUCH THAT N:=()size_of_Automaton_list(LA):{
rules for statezone
  {s_I : state ;}_I=1...N
  {ss_J : state ;}_J=1...N
  Phi, nPhi       : clockzone ;
  lbl             : label ;
  sz              : statezone ;
 global
{
  [ExecuteTransition_i] SynTransitionOperator(lbl,{s_j.}_j=1...(i-1) s_i.{s_j.}_j=(i+1)...N nil/Phi) =>
         SynTransitionOperator(lbl,{s_j.}_j=1...(i-1) ss_i.{s_j.}_j=(i+1)...N nil/nPhi)
                                      if action(lbl,s_i,i)
                                      where sz:=()TransitionOperator.lbl(s_i/Phi)
                                      where ss_i:=()st(sz)
                                      where nPhi:=()zn(sz)
                                      end
  [JumpStep_i] SynTransitionOperator(lbl,sz) => SynTransitionOperator(lbl,sz) end
}_i=1...N
  [FinishSynTransition]   SynTransitionOperator(lbl,sz) => sz end
end // of rules for statezone
} // End Of The Transcription Of The Synchronised Product

strategies for statezone
 implicit
 FOR EACH N : int SUCH THAT N:=()size_of_Automaton_list(LA) :{
  [] next_sz => {first one(ExecuteTransition_I,JumpStep_I);}_I=1...N  FinishSynTransition end
 }
end // of strategies for statezone
```

Fig. 11. On-line generation of the transitions of the product of automata.

6 Conclusion

This paper presents the use of the rewrite based system ELAN to prototype model-checking algorithms for timed automata. As expected, the performance are a little bit lower than model-checking dedicated tools like KRONOS [14] or UPPAAL [18]. However, using the specific techniques already developed for

compiling strategy controlled rewrite systems implemented in the ELAN system compiler, the performances turns out to be of same magnitude.

The main advantage is the gained flexibility compared to conventional programming languages. The whole ELAN code for the described model-checkers is less than 1100 lines (including comments). Changing the graph exploration technique, or the constraint solving algorithm, for example, turned out to require to modify only a few lines. We presented the direct and classical implementation of the reachability algorithms for timed automata. But other techniques (e.g.: partition refinement techniques [1, 2], alternative representations for constraints [6, 8, 10, 20, 27]) could also be experimented and would require only a few modifications of the existing ELAN code.

Furthemore, the ELAN system offers facilities, such as a strategy language which provides flexibily for customizations that are not available in typical hard-wired model checkers such as programmable strategies.

Moreover, we believe that such a work helps to understand mixture of proving and constraint manipulation techniques by studying them in the same unifying framework. In particular, it clearly helps to delimit the difference between pure computations and deductions in model-checking techniques which are very often presented as relying only on computations on constraints.

More details on the tool together with the full code can be found on web page `http://www.loria.fr/~kacem/AT`.

7 Thanks

The authors would like to thank all members of the PROTHEO group for their discussions. Among them, we would like to address some special thanks to Pierre-Etienne Moreau who helped us a lot through his expertise of the ELAN compiler.

References

1. R. Alur. Timed automata. In *NATO-ASI Summer School on Verification of Digital and Hybrid Systems*, 1998.
2. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 6 February 1995.
3. 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.
4. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 25 April 1994. Fundamental Study.
5. R. Alur and R. P. Kurshan. Timing analysis in COSPAN. *Lecture Notes in Computer Science*, 1066:220, 1996.
6. E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data-structures for the verification of timed automata. In *Proc. HART'97*, volume 1201 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 1997.

7. Emmanuel Beffara. Automates temporisés et calcul de réécriture. Rapport de fin de stage, ENS Lyon, September 2000.
8. G. Behrmann, K. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Proc. CAV'99*. Springer, 1999.
9. Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, volume 15, `http://www.elsevier.nl/locate/entcs/volume15.html`, Pont-à-Mousson (France), September 1998. Electronic Notes in Computer Science.
10. Olivier Bournez and Oded Maler. On the representation of timed polyhedra. In *International Colloquium on Automata Languages and Programming (ICALP'00)*, volume 1853 of *Lecture Notes in Computer Science*, pages 793–807, Geneva, Switzerland, 9–15 July 2000. Springer.
11. Horatiu Cirstea. *Calcul de réécriture : fondements et applications*. Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, 2000.
12. Horatiu Cirstea and Claude Kirchner. Combining higher-order and first-order computation using ρ-calculus: Towards a semantics of ELAN. In Dov Gabbay and Maarten de Rijke, editors, *Frontiers of Combining Systems 2*, Research Studies, ISBN 0863802524, pages 95–120. Wiley, 1999.
13. E. Clarke, O. Grumberg, and D. Peled. Model checking, 1999.
14. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. *Lecture Notes in Computer Science*, 1066:208, 1996.
15. C. Daws and S. Yovine. Two examples of verification of multirate timed automata with kronos. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 66–75. IEEE Computer Society Press, 1995.
16. 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.
17. Piotr Kosiuczenko and Marting Wirsing. Timed rewriting logic with an application to object-based specification. *Science of Computer Programming*, 2-3(28), 1997.
18. K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. In *Software Tools for Technology Transfer*, volume 1, 1997.
19. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
20. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Computer Science Logic*, The IT University of Copenhagen, Denmark, September 1999.
21. 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.
22. P. Olveczky and J. Meseguer. Specifying real-time systems in rewriting logic, 1996.
23. P. Olveczky and J. Meseguer. Specifying real-time and hybrid systems in rewriting logic, December 1999. http://maude.csl.sri.com/papers.
24. P. Olveczky and J. Meseguer. Real-time maude: A tool for simulating and analyzing real-time and hybrid systems. In *Third International Workshop on Rewriting Logic and its Applications (WRLA00)*, Electronic Notes in Theoretical Computer Science, Elsevier Science, Kanazawa Cultural Hall, Kanazawa, Japan, September 2000.

25. P. C. Olveczky, P. Kosiuczenko, and M. Wirsing. An object-oriented algebraic steam-boiler control specification. In J. R. Abrial and al, editors, *Formal Methods for Industrial Application: Specifying and Programming the Steam Boiler Control*, Lecture Notes in Computer Science, pages 379–402. Springer, 1996.
26. L.J. Steggles and P. Kosiuczenko. A timed rewriting logic semantics for sdl: A case study of alternating bit protocol. In Claude Kirchner and Helene Kirchner, editors, *Electronic Notes in Theoretical Computer Science*, volume 15. Elsevier Science Publishers, 2000.
27. F. Wang. Efficient data structure for fully symbolic verification of real-time software systems. In *Proc. TACAS'00*, volume 1785 of *Lecture Notes in Computer Science 1785*. Springer, 2000.