Experiments using XTA and ITS-tools

Yann Thierry-Mieg[1] and Maximilien Colange[2]

[1] LIP6, CNRS UMR 7606, Université P. & M. Curie – Paris 6
4 place Jussieu, F-75252 Paris Cedex 05, France
yann.thierry-mieg@lip6.fr
[2] Centre Universitaire d'Informatique, Université de Genève
7 route de Drize, 1227 Carouge, Switzerland
maximilien.colange@unige.ch

Abstract. This document summarizes experiments we performed using XTA input (via GAL transformation) and its-tools to analyze timed automata using discrete time assumptions. We provide comparisons to reference tool Uppaal.

1 Introduction

Symbolic model-checking using decision diagrams (DD) can be very effective, but expressing a transition relation symbolically is not easy in general. Building upon an existing DD package (such as CuDD [17]), then encoding states and the transition relation with low-level concepts [12] (Boolean functions, union...) requires a high level of expertise, far beyond what can be reasonably expected from an end-user of the technology. This difficult task is further compounded by technicalities that may have a great impact on performance (such as variable ordering). Most symbolic model-checkers such as NuSMV or VIS [2, 11] primarily target gate-level hardware verification, with synchronous semantics. Encoding some semantics in such a setting can be difficult and/or cumbersome, leading to traceability and trace interpretation issues.

To ease the interaction with our efficient symbolic back-end, we recently focused on building a general-purpose language to model concurrent specifications with data (arrays, arithmetic...). The bridge between this language GAL and the back-end notably relies on the expressive symbolic operations defined in [13].

We thus use the Guarded Action Language (GAL), specifically designed to easily express a wide range of concurrent semantics. It is supported natively by the efficient symbolic model-checker ITS-tools. GAL defines no high-level concepts (no explicit notion of process, channel, ...), but it offers a lot of flexibility when defining the atomicity of operations and compactly expresses non determinism.

2 Guarded Action Language

The reader is referred to the GAL semantics definition, provided on our webpage : `http://ddd.lip6.fr/files/gal.pdf` for a definition of GAL and its semantics consistent with notations used in this document.

```
chan take, release;
int [0,1] L;
// a Soldier
process S(const int delay) {
clock y;
state S0, safe, S1, unsafe;
init unsafe;
trans
S0 -> safe { guard y >= delay;
    sync release!; },
safe -> S1 { guard L == 1;
    sync take !; assign y = 0; },
S1 -> unsafe { guard y >= delay;
    sync release!; },
unsafe -> S0 { guard L == 0;
    sync take !; assign y = 0; };
}
// a Torch
process T() {
state one, S0, free, two;
urgent S0;
init free;
trans
free -> S0 { sync take?; },
S0 -> one { },
S0 -> two { sync take?; },
one -> free { sync release?;
    assign L = 1 - L; },
two -> one { sync release?; };
}

const int fastest = 5;
const int fast    = 10;
const int slow    = 20;
const int slowest = 25;

Viking1 = S(fastest);
Viking2 = S(fast);
Viking3 = S(slow);
Viking4 = S(slowest);

system Viking1, Viking2, Viking3,
       Viking4, T;
```

Fig. 1. Uppaal timed automata bridge example (Uppaal native textual .xta format).

3 Uppaal timed automata

We present in this section a transformation of Timed Automata (TA), interpreted with discrete time semantics, to GAL. In a discrete time setting, the semantics of TA is defined as a discrete time transition system, i.e. a labeled transition system where one label (noted *elapse*) represents a delay of one time unit. Note that analysis in the discrete setting has been shown to be equivalent to analysis in a dense time setting provided all constraints in the automata are of the form $x \leq k$ but not $x < k$ [14, 8]. This is due to the fact that if bounds are open, timings in zones with non integer values may equivalently be represented (in terms of future behavior) by an integer timing touching one of its border.

We use this formalism as an example because its semantics are relatively rich, allowing to highlight the characteristics of GAL. The fragment we consider here contains typical features of many DSL.

Uppaal Timed Automata. Uppaal [5, 3] specifications consist in a set of timed automata that may communicate through channels (synchronously) or using shared variables and shared clocks. Each automata may bear local variables and clocks. Each automata defines a set of locations, each of which may carry a clock invariant of the form $x \leq k$ constraining how much time can elapse while in that location. In locations noted as being *urgent* time cannot elapse. Transitions are edges between a source and a target location, bearing boolean combinations of time constraints $x \leq k$ and $x \geq k$ giving a time precondition to the transition, a set of clock reset statements of the form $x = 0$, and an effect that consists in a sequence of assignments of expressions to variables, and/or the send or receive action on a channel. We assume the channels do not carry data but model *rendez-vous* semantics. The only data types manipulated are subsets of integers. Symbolic names for constants can be defined. A full Uppaal specification is given by

```
gal bridge_pop
// constant parameters
($fastest = 5, $fast = 10,
 $slow = 20, $slowest = 25)
{
// range declarations
typedef Sid_t = 0 .. 3 ;
typedef Tid_t = 0 .. 0 ;
int glob_L = 0 ;
array [4] S_state =(3,3,3,3);
array [4] S_delay =
 ($fastest,$fast,$slow,$slowest);
array [4] S_c_y =(0,0,0,0);
array [1] T_state =(2);
transition chantake [true]
  label "dtrans" {
 self."Sendtake" ;
 self."Recvtake" ;
}
transition chanrelease [true]
  label "dtrans" {
 self."Sendrelease" ;
 self."Recvrelease" ;
}
transition elapse [true]
  label "elapseOne" {
 for ($Sid : Sid_t) {
  self."updClkSy_$Sid";
 }
 for ($Tid : Tid_t) {
  self."passUrgentT_$Tid";
 }
}
// S.S0 has a max tracking value.
transition updClkSy_S0 (Sid_t $Sid)
  [S_state[$Sid] == 0]
  label "updClkSy_$Sid" {
 if (! S_c_y[$Sid] >= S_delay[$Sid]) {
  S_c_y[$Sid] = S_c_y[$Sid] + 1 ;
 }
}
// S.safe is inactive.
transition updClkSy_safe (Sid_t $Sid)
  [S_state[$Sid] == 1]
  label "updClkSy_$Sid" { }
// S.S1 has a max tracking value.
transition updClkSy_S1 (Sid_t $Sid)
  [S_state[$Sid] == 2]
  label "updClkSy_$Sid" {
 if (! S_c_y[$Sid] >= S_delay[$Sid]) {
  S_c_y[$Sid] = S_c_y[$Sid] + 1;
 }
}
// S.unsafe is inactive.
transition updClkSy_unsafe (Sid_t $Sid)
  [S_state[$Sid] == 3]
  label "updClkSy_$Sid" { }
// Make sure T is not in urgent location
transition passUrgentT (Tid_t $Tid)
  [T_state[$Tid] == 0
     || T_state[$Tid] == 2
     || T_state[$Tid] == 3]
  label "passUrgentT_$Tid" {}
// discrete transitions
transition t1SS0_safe (Sid_t $Sid)
  [S_state[$Sid] == 0
   && S_c_y[$Sid] >= S_delay[$Sid]]
  label "Sendrelease" {
 S_state[$Sid] = 1;
 S_c_y[$Sid] = 0;
}
transition t2Ssafe_S1 (Sid_t $Sid)
  [S_state[$Sid] == 1 && glob_L == 1]
  label "Sendtake" {
 S_state[$Sid] = 2;
 S_c_y[$Sid] = 0 ;
}
transition t3SS1_unsafe (Sid_t $Sid)
  [S_state[$Sid] == 2
   && S_c_y[$Sid] >= S_delay[$Sid]]
  label "Sendrelease" {
 S_state[$Sid] = 3 ;
 S_c_y[$Sid] = 0 ;
}
transition t4Sunsafe_S0 (Sid_t $Sid)
  [S_state[$Sid] == 3 && glob_L == 0]
  label "Sendtake" {
 S_state[$Sid] = 0 ;
 S_c_y[$Sid] = 0 ;
}
transition t1Tfree_S0 (Tid_t $Tid)
  [T_state[$Tid] == 2]
  label "Recvtake" {
 T_state[$Tid] = 1 ;
}
transition t2TS0_one (Tid_t $Tid)
  [T_state[$Tid] == 1]
  label "dtrans" {
 T_state[$Tid] = 0 ;
}
transition t3TS0_two (Tid_t $Tid)
  [T_state[$Tid] == 1]
  label "Recvtake" {
 T_state[$Tid] = 3 ;
}
transition t4Tone_free (Tid_t $Tid)
  [T_state[$Tid] == 0]
  label "Recvrelease" {
 T_state[$Tid] = 2 ;
 glob_L = 1 - glob_L ;
}
transition t5Ttwo_one (Tid_t $Tid)
  [T_state[$Tid] == 3]
  label "Recvrelease" {
 T_state[$Tid] = 0 ;
}
// to accumulate states in the fixpoint
transition id [true]
  label "elapseOne" { }

// Go to successor essential states,
// in one atomic step.
transition succ [true]
  // no label = label \tau of formal def.
{
 // Accumulate states reachable by arbitrary delay
 fixpoint {
  self."elapseOne" ;
 }
 // Fire one of the discrete transitions
 self."dtrans" ;
}
} //end of GAL
```

Fig. 2. GAL resulting from the transformation of the Uppaal bridge model, with essential states semantics.

instantiating process type definitions, allowing to share their description. Process type descriptions can carry parameters that are given a value at instantiation time.

Most of these features are visible on the example of figure 1, taken from the Uppaal distribution. This is a classical problem where the goal is to minimize the time necessary to get everybody across the bridge using a single Torch, when the bridge can only bear two persons. The translation to GAL applies the following rules (please refer to the example listings of Fig.1 and Fig.2):

Shared variables. Each global clock or variable is translated to an integer variable (e.g. *L*). Symbolic names for constants are translated to GAL constant parameters (e.g. *slowest, slow ...*). Channel *c* is translated to a GAL transition with body $call(send_c); call(receive_c)$ and label *dtrans* (for "discrete transition").

Process type. For each process type declaration T we compute the number of instances n and assign an index to each of them (e.g. Viking1 gets index 0 amongst S instances). We define a GAL range id_T as the interval 0 to $n-1$. We build for each local variable or clock of the process type an array of n variables to hold the values of each instance. We also create an array of n variables to store the current location of each automata. Parameters of the process type are given an initial value, matching that of the Uppaal specification. These additional variables will be written out as constants before model-checking, but they help traceability and readability.

Discrete Transition. For each transition t of a process type declaration T, we create a transition in the GAL with a parameter *id* in range id_T. All accesses to a local variable or clock in x are translated to $x[id]$. The GAL guard of the transition is directly translated from the enabling conditions of the TA transition, with an added test for the current location of the automata. If the transition sends $c!$ or receives $c?$ from a channel c, the label of the resulting transition is $send_c$ or $receive_c$ as appropriate. Otherwise we label the transition *dtrans*. The actions (resets and effects) are translated directly to the body of the GAL transition. The location is also updated if necessary (i.e. source and target locations of the TA transition differ). If the target location has a clock invariant of the form $x \leq k$ and x is not reset we add a statement $if(!x \leq k)abort;$ at the end of the transition body.

Time delay. We add a transition labeled *elapse* to represent a time delay of one unit. For each clock c, we introduce a label $elapse_c$ that will label all possible ways of updating c at a given time step. The body of the elapse transition is a sequence of calls to each of the $elapse_c$ labels for each of the clocks of the specification, so that time elapses at the same rate for all clocks. If it is a clock local to type T, the label carries an *id* parameter in range id_T and each of these labels is called in elapse (we use a for loop, see Fig.2). For each automata instance that can see the clock (i.e. only one for local clocks as in the example), we create a transition with a guard that tests its current location and with three possible body definitions:

- The location can be *inactive* w.r.t to clock x. This means the clock will be reset before it can ever be read according to the locally reachable transitions of the TA. In such a case no action needs to be taken, the transition has empty body. However, an additional statement to reset x is added at the end of every transition leading to this local state (e.g. line 84 of Fig.2).

- The location is *time constrained* w.r.t to clock x, i.e. the location has a clock invariant of the form $x \leq k$. The GAL transition body is : $if(!x < k)\{abort;\}else\{x = x+1;\}$. If $k = 0$ (urgent location) it thus reduces to an immediate abort since clocks are positive integers.
- The location is *time monitored* up to constant K w.r.t. clock x. K is the highest constant that local clock x is compared against on TA transitions and locations reachable from the current location without resetting x. To simplify tests, we let K be the highest constant global clock x is compared against in any transition of any automata. In such a case, the body of the GAL transition is $if(x \leq K)\{x = x+1;\}$, incrementing and tracking the clock value up to $K+1$, thus exploring all firing or disabling opportunities while maintaining a finite support for the state space. Clock values might otherwise diverge in some locations constrained only by an earliest firing time and no upper bound on delay.

Urgent locations. For automata with urgent locations but no local clocks, we add to elapse a call to a transition that tests that the current location is *not* urgent.

Our prototype translation does not yet support the full language of Uppaal, including function calls and commit semantics, though these could be implemented as well.

These translation rules produce a parametric GAL specification, with two labels $dtrans$ and $elapse$ representing respectively the effect of all discrete transitions and of time elapsing by one unit (if possible).

A first approach is to let both $dtrans$ and $elapse$ be equal to the local label τ of GAL definition. This means that successors of a state are reached by a discrete transition or by waiting for one time unit in the current location. This encoding allows to observe all time steps, hence a shortest path trace returned by *ITS-tools* for the reachability property “everyone reaches the other side” will contain a minimal number of $elapse$ occurrences i.e. a solution to the optimization problem.

Essential States. However, we could consider a smaller abstraction of the state graph, that preserves CTL properties provided atomic properties of the formula do not refer to clocks. This abstraction called *essential states* was first defined [16] for time Petri nets in a discrete time setting (and is implemented notably by the tool Tina [7] see option $-SD$). It retains in the state graph only states that are reached through a discrete transition. While it abstracts sequences of time steps, it preserves location reachability and causality since all timings from a location are explored. Because discrete transitions frequently reset clocks, many (abstracted) states adjacent by a time step often lead to a single successor essential states, helping reduce the state graph size.

To implement this idea, we add a transition that computes this abstract successor relation directly in GAL. We first add a transition $id = \langle elapse, true, nop \rangle$ to represent accumulation of states (identity transition). We then compute the set of all states reachable by waiting in the current location with statement $fixpoint\{call(elapse);\}$. Finally we fire any discrete transition with $call(dtrans)$ to obtain a successor essential state.

This example shows an advanced usage of the features of GAL, since it exploits the particular semantics of the fixpoint, and was chosen for this purpose. Its computation will automatically benefit from saturation at the ITS-tools level.

From the implementation point of view, we wrote a small Xtext [4] grammar recognizing the Uppaal XTA format, generating a parser/serializer, a rich editor and an EMF metamodel of XTA (150 lines, generating several klocs). By leveraging EMF tools [1] a rich API for manipulation of model instances can then be generated. The transformation is written in plain Java (900 lines in total), using the EMF API to both explore the source XTA model and produce target GAL models. This implementation path is recommended for our target users who develop a DSL. If the EMF metamodel of their DSL is already available (which is often the case) adoption cost is very low. Dialects of communicating state machines are a particularly good target for this.

4 Experiments

We experimented the model-checking of Timed Automata through a GAL translation on three scaleable benchmark examples provided with the Uppaal distribution (see Fig 3). We experimented scaling both number of process and clock bounds. When clock bounds are scaled, Uppaal DBM encoding of zones is basically not impacted, but larger zones means exponentially more discrete states in our setting. Hence Uppaal is much more resilient to large clock values than either of our models as could unfortunately be expected in a discrete setting. However, when more parallel processes are added the symbolic approach scales decently, quickly outperforming Uppaal.

Though the essential states semantics approach always constructs less states than the one time unit step approach, it does not always outperform it in time and memory. The additional cost of the fixpoint at each step of the transition relation outweighs the gains from the reduction in number of states for two of the models.

Overall the experiments agree with those of KronosBDD [10], Rabbit [9] or more recently in PAT [15] that use similar symbolic encodings of states and the same discrete assumptions we have.Though symbolic encodings are resistant to an increased process count they scale poorly with clock values. The basic problem is that constraints between variables encoding clocks x and y of the form $x < y+k$, which occur naturally in the state space are very poorly encoded by a decision diagram, since each value of x leads to a different set of values for y, severely limiting sharing of subtrees. A number of dedicated symbolic data structures (CDD [6], CRD [18]) have been proposed to tackle this problem, but these are out of scope.

Note that these tools handling discrete time symbolically were all written at a decision diagram operation level, involving much more effort and expertise than our experimental translation described above. With this translation to GAL, ITS-tools becomes a viable alternative to analyze timed automata with a lot of concurrency.

5 Conclusion

Depending on the nature of the specification, particularly if it involves a high number of concurrently active clocks and/or a high number of automata, but rather small time bounds (or if the time bounds can be appropriately scaled down), ITS-tools are a good candidate for performing formal analysis.

A contrario, if the system has little true concurrency and/or uses large time bounds, the explicit state approach using DBM to represent zones scales much better.

# proc	clock	Uppaal			GAL essential			GAL one		
	bound	time	mem	states	time	mem	states	time	mem	states
Fischer mutual exclusion										
10	4	4.9	35K	$2.6 \cdot 10^5$	9.4	212M	$5.3 \cdot 10^9$	26	701M	$7 \cdot 10^9$
10	8	4.9	35K	$2.6 \cdot 10^5$	40	851M	$2.9 \cdot 10^{11}$	218	1.5G	$4.9 \cdot 10^{11}$
10	16	5	35K	$2.6 \cdot 10^5$	575	1.8G	$3.8 \cdot 10^{13}$	TO	3.8G	?
15	2	416	MO	?	9.5	277M	$3.4 \cdot 10^{12}$	43	1.1G	$3.5 \cdot 10^{12}$
20	2	250	MO	?	26	700M	$3.6 \cdot 10^{16}$	411	1.4G	$3.7 \cdot 10^{16}$
30	2	350	MO	?	269	1.3G	$3.3 \cdot 10^{24}$	488	MO	?
CSMA Collision Detection										
5	7,14,21	0	1K	692	2	56M	$1.8 \cdot 10^6$	4	100M	$5.3 \cdot 10^6$
5	13,26,55	0	1K	692	18	354M	$2.3 \cdot 10^7$	26	493M	$1.3 \cdot 10^8$
5	26,52,808	0	1K	692	TO	3.6G	?	TO	1.4G	?
15	7,14,21	133	1.5G	$1.2 \cdot 10^6$	108	1.3G	$6.7 \cdot 10^{18}$	37	888M	$1.1 \cdot 10^{19}$
20	7,14,21	364	MO	?	457	1.7G	$1 \cdot 10^{25}$	88	1.7G	$1.6 \cdot 10^{25}$
25	7,14,21	540	MO	?	TO	2.3G	?	167	3.3G	$2.1 \cdot 10^{31}$
HDDI Token Ring										
15	2,752	167	10K	1132	334	1.3G	90	85	1.2G	$2.26 \cdot 10^4$
15	5,755	167	10K	1132	351	1.4G	90	87	1.2G	$2.27 \cdot 10^4$
15	10,760	168	10K	1132	383	1.5G	90	90	1.3G	$2.28 \cdot 10^4$
10	20,520	0	6K	507	39	880M	60	22	377M	$1 \cdot 10^4$
15	20,770	167	10K	1132	345	1.5G	90	4	100M	$2.3 \cdot 10^4$
20	20,1020	TO	17K	?	TO	1.5G	?	233	1.3G	$4.1 \cdot 10^4$

Fig. 3. Performance comparison of *ITS-tools* and Uppaal 4.1.16. GAL essential corresponds to the essential states abstraction and GAL one to the one unit time delay semantics. TO indicates a timeout (600 seconds), MO indicates a memory overflow (4GB). Run on a Linux 64 Intel Core7.

References

1. Eclipse Modeling Framework. `http://www.eclipse.org/modeling/emf/`.
2. NuSMV: a new symbolic model verifier. `http://nusmv.fbk.eu/`.
3. Uppaal home page. `http://www.uppaal.org`.
4. Xtext: a tool for building textual dsl. `https://www.eclipse.org/Xtext/`.
5. G. Behrmann, K. G. Larsen, O. Moller, A. David, P. Pettersson, and W. Yi. Uppaal-present and future. In *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, volume 3, pages 2881–2886. IEEE, 2001.
6. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Computer aided verification*, pages 341–353. Springer, 1999.
7. B. Berthomieu and F. Vernadat. Time petri nets analysis with tina. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 123–124. IEEE, 2006.
8. D. Beyer. Improvements in BDD-based reachability analysis of timed automata. In J. N. Oliveira and P. Zave, editors, *Proceedings of the Tenth International Symposium of Formal Methods Europe (FME 2001, Berlin, March 12-16): Formal Methods for Increasing Software Productivity*, LNCS 2021, pages 318–343. Springer-Verlag, Heidelberg, 2001.

9. D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A tool for BDD-based verification of real-time systems. In W. A. Hunt and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003, Boulder, CO, July 8-12)*, LNCS 2725, pages 122–125. Springer-Verlag, Heidelberg, 2003.
10. M. Bozga, O. Maler, and S. Tripakis. Efficient verification of timed automata using dense and discrete time semantics. In *Correct Hardware Design and Verification Methods*, pages 125–141. Springer, 1999.
11. R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96,*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, NJ, USA, July 1996. Springer.
12. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. Hwang. Symbolic model checking: 10^{20} States and beyond. *Information and computation*, 98(2):142–170, 1992.
13. M. Colange, S. Baarir, F. Kordon, and Y. Thierry-Mieg. Towards Distributed Software Model-Checking using Decision Diagrams. In *25th International Conference on Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 830–845. Springer Verlag, July 2013.
14. T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Automata, Languages and Programming*, pages 545–558. Springer, 1992.
15. T. K. Nguyen, J. Sun, Y. Liu, J. S. Dong, and Y. Liu. Improved bdd-based discrete analysis of timed systems. In *FM 2012: Formal Methods*, pages 326–340. Springer, 2012.
16. L. Popova-Zeugmann and D. Schlatter. Analyzing paths in time petri nets. *Fundamenta Informaticae*, 37(3):311–327, 1999.
17. F. Somenzi. The colorado university decision diagrams package. `http://vlsi.colorado.edu/~fabio/CUDD/`.
18. F. Wang. Efficient verification of timed automata with bdd-like data structures. *STTT*, 6(1):77–97, 2004.