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.
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, al-
lowing 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 au-
tomata that may communicate through channels (synchronously) or using shared vari-
ables and shared clocks. Each automata may bear local variables and clocks. Each au-
tomata 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
array

self. "Send release";

transition 2Safe_S1 (Sid_t $Sid)
[S_state[$Sid] == 1 & & glob_L == 1]
label "Sendtake"[
S_state[$Sid] = 2;
S_c_y[$Sid] = 0;
]
transition t3S1Unsafe (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 t1Free_S0 (Tid_t $Tid)
[T_state[$Tid] == 2]
label "Recvtake"[
T_state[$Tid] = 1;
]
transition t2T0_one (Tid_t $Tid)
[T_state[$Tid] == 1]
label "Recvtake"[
T_state[$Tid] = 3;
]
transition t4Tone_free (Tid_t $Tid)
[T_state[$Tid] == 0]
label "Recvtake"[
T_state[$Tid] = 2;
]
transition t5Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t1Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t1S0_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 "Recvtake"[
T_state[$Tid] = 2;
]
transition t5Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t1Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t4Tone_free (Tid_t $Tid)
[T_state[$Tid] == 0]
label "Recvtake"[
T_state[$Tid] = 2;
]
transition t5Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t1Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t4Tone_free (Tid_t $Tid)
[T_state[$Tid] == 0]
label "Recvtake"[
T_state[$Tid] = 2;
]
transition t5Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t1Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t4Tone_free (Tid_t $Tid)
[T_state[$Tid] == 0]
label "Recvtake"[
T_state[$Tid] = 2;
]
transition t5Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t1Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t4Tone_free (Tid_t $Tid)
[T_state[$Tid] == 0]
label "Recvtake"[
T_state[$Tid] = 2;
]
transition t5Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t1Ttwo_one (Tid_t $Tid)
[T_state[$Tid] == 3]
label "Recvtake"[
T_state[$Tid] = 0;
]
transition t4Tone_free (Tid_t $Tid)
[T_state[$Tid] == 0]
label "Recvtake"[
T_state[$Tid] = 2;
]
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. slow-
est, 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: 
  \[
  \text{if} \left\{ \begin{array}{l}
  x < k; \\
  \text{else}
  \end{array} \right. \{ \text{abort}; \} \}
  \]
  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 
  \[
  \text{if} \left\{ \begin{array}{l}
  x \leq K; \\
  \text{else}
  \end{array} \right. \{ x = x + 1; \}
  \]
  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 \( \tau \) 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 step time 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 \( id = (elapse, true, nop) \) to represent accumulation of states (identity transition). We then compute the set of all states reachable by waiting in the current location with statement 
  \[
  \text{fixpoint} \{ \text{call}(elapse); \}
  \]
  Finally we fire any discrete transition with \( \text{call}(dtrans) \) 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.
<table>
<thead>
<tr>
<th># proc</th>
<th>clock bound</th>
<th>Uppaal states</th>
<th>GAL essential states</th>
<th>GAL one states</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time mem</td>
<td></td>
<td>time mem</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>4.9</td>
<td>35K 2.6·10⁵</td>
<td>9.4</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>4.9</td>
<td>35K 2.6·10⁵</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>5</td>
<td>35K 2.6·10⁵</td>
<td>575</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>416</td>
<td>MO</td>
<td>?</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>250</td>
<td>MO</td>
<td>?</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>350</td>
<td>MO</td>
<td>?</td>
</tr>
<tr>
<td>5</td>
<td>7,14,21</td>
<td>0</td>
<td>1K 692</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>13,26,55</td>
<td>0</td>
<td>1K 692</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>26,52,808</td>
<td>0</td>
<td>1K 692</td>
<td>TO 3.6G</td>
</tr>
<tr>
<td>15</td>
<td>7,14,21</td>
<td>133</td>
<td>1.5G 1.2·10⁹</td>
<td>108</td>
</tr>
<tr>
<td>20</td>
<td>7,14,21</td>
<td>364</td>
<td>MO</td>
<td>457</td>
</tr>
<tr>
<td>25</td>
<td>7,14,21</td>
<td>540</td>
<td>MO</td>
<td>TO 2.3G</td>
</tr>
<tr>
<td>15</td>
<td>2,752</td>
<td>167</td>
<td>10K 1132</td>
<td>334</td>
</tr>
<tr>
<td>15</td>
<td>5,755</td>
<td>167</td>
<td>10K 1132</td>
<td>351</td>
</tr>
<tr>
<td>15</td>
<td>10,760</td>
<td>168</td>
<td>10K 1132</td>
<td>383</td>
</tr>
<tr>
<td>10</td>
<td>20,520</td>
<td>0</td>
<td>6K 507</td>
<td>39</td>
</tr>
<tr>
<td>15</td>
<td>20,770</td>
<td>167</td>
<td>10K 1132</td>
<td>345</td>
</tr>
<tr>
<td>20</td>
<td>20,1020</td>
<td>TO 17K</td>
<td>?</td>
<td>TO 1.5G</td>
</tr>
</tbody>
</table>

Fischer mutual exclusion

<table>
<thead>
<tr>
<th></th>
<th>time mem</th>
<th></th>
<th>time mem</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
<td>212M 5.3·10⁹</td>
<td>218</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>851M 2.9·10¹²</td>
<td>411</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>575</td>
<td>1.8G 3.8·10¹³</td>
<td>TO 3.8G</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>277M 3.4·10¹²</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>700M 3.6·10¹⁶</td>
<td>411</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>269</td>
<td>1.3G 3.3·10²⁴</td>
<td>488 MO</td>
</tr>
</tbody>
</table>

CSMA Collision Detection

<table>
<thead>
<tr>
<th></th>
<th>time mem</th>
<th></th>
<th>time mem</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2,752</td>
<td>153</td>
<td>1.5G 1.2·10⁹</td>
<td>108</td>
</tr>
<tr>
<td>5</td>
<td>26,52,808</td>
<td>0</td>
<td>1K 692</td>
<td>TO 3.6G</td>
</tr>
<tr>
<td>15</td>
<td>7,14,21</td>
<td>133</td>
<td>1.5G 1.2·10⁹</td>
<td>108</td>
</tr>
<tr>
<td>20</td>
<td>7,14,21</td>
<td>364</td>
<td>MO</td>
<td>457</td>
</tr>
<tr>
<td>25</td>
<td>7,14,21</td>
<td>540</td>
<td>MO</td>
<td>TO 2.3G</td>
</tr>
</tbody>
</table>

HDDI Token Ring

<table>
<thead>
<tr>
<th></th>
<th>time mem</th>
<th></th>
<th>time mem</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>2,752</td>
<td>167</td>
<td>10K 1132</td>
<td>334</td>
</tr>
<tr>
<td>15</td>
<td>5,755</td>
<td>167</td>
<td>10K 1132</td>
<td>351</td>
</tr>
<tr>
<td>15</td>
<td>10,760</td>
<td>168</td>
<td>10K 1132</td>
<td>383</td>
</tr>
<tr>
<td>10</td>
<td>20,520</td>
<td>0</td>
<td>6K 507</td>
<td>39</td>
</tr>
<tr>
<td>15</td>
<td>20,770</td>
<td>167</td>
<td>10K 1132</td>
<td>345</td>
</tr>
<tr>
<td>20</td>
<td>20,1020</td>
<td>TO 17K</td>
<td>?</td>
<td>TO 1.5G</td>
</tr>
</tbody>
</table>

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