Tools and Algorithms for Deciding Relations on Timed Automata

Mihir Mehta

Department of Computer Science and Engineering,
Indian Institute of Technology, Delhi.
cs1090197@cse.iitd.ac.in

Abstract. This describes the author's work in implementing the construction of zone-valuation graphs for timed automata and using these to verify certain relations on pairs of timed automata.

Table of Contents

List of Figures

List of Tables

1 Objectives

The work described in this thesis builds on the work of [3] in which Guha et al described an algorithm to generate *zone valuation graphs* for timed automata and an algorithm to use such zone-valuation graphs to determine *timed performance prebisimilarity* on pairs of timed automata. Our aim was to implement these algorithms in a generalised manner in order to verify various other time abstracted relations, such as time abstracted bisimulations [4] and time abstracted simulation equivalence. Towards this end, we studied the literature about timed untimed automata as well as various existing tools for verifying these equivalences (such as `minim`, described in [4]). Our implementation, in OCaml, implements several of these relations and leaves some scope for implementing others.

2 Labelled transition systems

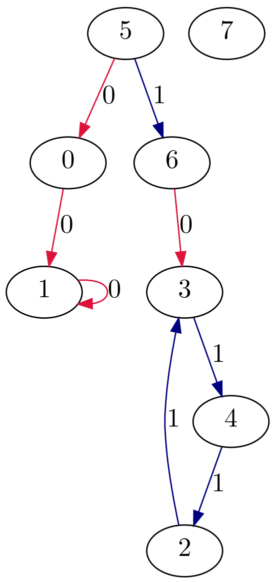


Fig. 1: An example of a labelled transition system. Here, the states are $\{0, 1, 2, \ldots 7\}$ and the actions are $\{0, 1\}$.

Definition 1. Labelled Transition System*: A labelled transition system (LTS) [5] is an automaton which is described by*

- S*, a set of* states
- Act*, a set of* actions
- $\rightarrow \subseteq S \times Act \times S$*, a* transition relation*.*
- *optionally,* $I \subseteq S$ *,a set of initial states.*

LTS are useful for describing the behaviour of untimed systems, and serve as the foundation for the development of more complex models such as CCS and timed automata. Thus, equivalences on LTS serve as the theoretical foundation for many timed and time abstracted equivalences on timed automata, and also have direct applications in determining some of these equivalences in cases where timed automata can be reduced to equivalent LTS.

3 Equivalences on labelled transition systems

3.1 Strong bisimilarity

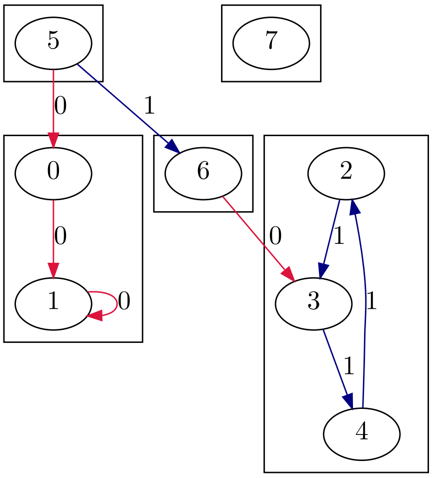


Fig. 2: Strong bisimilarity quotient of the LTS in Figure 1.

Definition 2. Strong bisimulation*: A binary relation R is a strong bisimulation if and only if, for all $(s_1, s_2) \in R$ and $a \in Act$.*
$\forall s_1'(s_1 \xrightarrow{a} s_1' \Rightarrow \exists s_2'.(s_2 \xrightarrow{a} s_2' \wedge (s_1', s_2') \in R))\wedge$
$\forall s_2'(s_2 \xrightarrow{a} s_2' \Rightarrow \exists s_1'.(s_1 \xrightarrow{a} s_1' \wedge (s_1', s_2') \in R))$

Definition 3. Strong bisimilarity*: It can be shown that the union of all strong bisimulations over the set of states is a strong bisimulation. This binary relation is called strong bisimilarity, denoted by $\sim$.*

4 Timed automata

Definition 4. Timed Automaton*: A timed automaton [6] over a finite set of clocks C and a finite set of actions Act is a 4-tuple (L, l_0, E, I).*

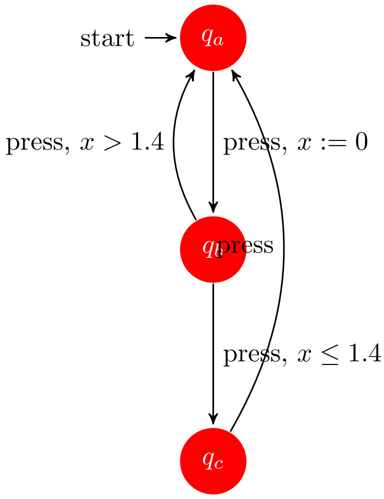


Fig. 3: Timed automaton representing a light bulb with two brightness settings, example taken from [1]

- *L is a finite set of locations.*
- *l_0 is the initial location.*
- *$E \subseteq L \times B(C) \times Act \times 2^C \times L$ is a finite set of edges.*
- *$I : L \to B(C)$ assigns invariants to each edge location.*
- *$B(C)$ is the set of clock constraints over C. An element of $B(C)$ can be an equality, a slack inequality, or a strict inequality on $v(x)$ for some $x \epsilon C$, or an AND combination of such constraints.*
- *The state of the automaton at any particular instant is given by the ordered pair (l, v) which gives the location and assigns a value to each clock.*
- *A transition is either a delay transition where the automaton stays at the same location while advancing each clock by the same time delay, or an action transition where the automaton performs a state change while resetting some of its clocks.*

It is evident that the state space in a timed automaton is, in general, uncountably infinite. This makes it impossible for any algorithm to terminate which explores the state space in a naive manner. However, some standard techniques to discretise the state space of timed automata exist, which we elaborate on, below.

- *State*: A state of a timed automaton is a pair (l, v) where l is a location in the automaton and v is a clock valuation satisfying $l.invar$
- *Symbolic state*: A symbolic state is a set of states in the timed automaton. The constituent states do not necessarily share a location.
- *Zone*: A zone is a symbolic state where the all the constituent states share a location and the set of valuations of these states forms a convex polyhedron on the valuation space.

5 Timed relations on timed automata

5.1 Time abstracted bisimilarity

Definition 5. Strong time abstracted bisimulation*: A binary relation R is a strong time abstracted bisimulation (STaB) if and only if, for all* $(s_1, s_2) \in R$, $a \in Act$, $d \in R_{\geq 0}$

$\forall s_1'(s_1 \xrightarrow{a} s_1' \Rightarrow \exists s_2'.(s_2 \xrightarrow{a} s_2' \wedge (s_1', s_2') \in R))\wedge$
$\forall s_2'(s_2 \xrightarrow{a} s_2' \Rightarrow \exists s_1'.(s_1 \xrightarrow{a} s_1' \wedge (s_1', s_2') \in R))\wedge$
$\forall s_1'(s_1 \xrightarrow{d} s_1' \Rightarrow \exists (s_2', d').(s_2 \xrightarrow{d'} s_2' \wedge (s_1', s_2') \in R))\wedge$
$\forall s_2'(s_2 \xrightarrow{d} s_2' \Rightarrow \exists (s_1', d').(s_1 \xrightarrow{d'} s_1' \wedge (s_1', s_2') \in R))$

Definition 6. Strong time abstracted bisimilarity*: It can be shown that the union of all strong time abstracted bisimulations over the set of (location, valuation) pairs is a strong time abstracted bisimulation. This binary relation is called strong time abstracted bisimilarity.*

Definition 7. Time abstracted delay bisimulation*: A binary relation R is a time abstracted delay bisimulation (TadB) if and only if, for all* $(s_1, s_2) \in R$, $a \in Act$, $d \in R_{\geq 0}$

$\forall s_1'(s_1 \xrightarrow{a} s_1' \Rightarrow \exists (s_2', d).(s_2 \xrightarrow{d}\xrightarrow{a} s_2' \wedge (s_1', s_2') \in R))\wedge$
$\forall s_2'(s_2 \xrightarrow{a} s_2' \Rightarrow \exists (s_1', d).(s_1 \xrightarrow{d}\xrightarrow{a} s_1' \wedge (s_1', s_2') \in R))\wedge$
$\forall s_1'(s_1 \xrightarrow{d} s_1' \Rightarrow \exists (s_2', d').(s_2 \xrightarrow{d'} s_2' \wedge (s_1', s_2') \in R))\wedge$
$\forall s_2'(s_2 \xrightarrow{d} s_2' \Rightarrow \exists (s_1', d').(s_1 \xrightarrow{d'} s_1' \wedge (s_1', s_2') \in R))$

Definition 8. Time abstracted delay bisimilarity*: It can be shown that the union of all time abstracted delay bisimulations over the set of (location, valuation) pairs is a time abstracted delay bisimulation. This binary relation is called time abstracted delay bisimilarity.*

Definition 9. Time abstracted observational bisimulation*: A binary relation R is a time abstracted observational bisimulation (TaoB) if and only if, for all* $(s_1, s_2) \in R$, $a \in Act$, $d \in R_{\geq 0}$

$\forall s_1'(s_1 \xrightarrow{a} s_1' \Rightarrow \exists (s_2', d, d').(s_2 \xrightarrow{d}\xrightarrow{a}\xrightarrow{d'} s_2' \wedge (s_1', s_2') \in R))\wedge$
$\forall s_2'(s_2 \xrightarrow{a} s_2' \Rightarrow \exists (s_1', d, d').(s_1 \xrightarrow{d}\xrightarrow{a}\xrightarrow{d'} s_1' \wedge (s_1', s_2') \in R))\wedge$
$\forall s_1'(s_1 \xrightarrow{d} s_1' \Rightarrow \exists (s_2', d').(s_2 \xrightarrow{d'} s_2' \wedge (s_1', s_2') \in R))\wedge$
$\forall s_2'(s_2 \xrightarrow{d} s_2' \Rightarrow \exists (s_1', d').(s_1 \xrightarrow{d'} s_1' \wedge (s_1', s_2') \in R))$

Definition 10. Time abstracted observational bisimilarity*: It can be shown that the union of all time abstracted observational bisimulations over the set of (location, valuation) pairs is a time abstracted observational bisimulation. This binary relation is called time abstracted observational bisimilarity.*

6 Algorithms

6.1 Creation of the zone valuation graph

Overview This algorithm is adapted from the algorithms in [3] and [7]. We changed it to make the correctness more evident. This algorithm consists of a *forward propagation* which ensures that all *reachable* zones are created, and a *backward propagation* which ensures that each zone is *stable* with respect to its successors.

For the forward propagation, we use a queue, akin to the queue of the classic *breadth-first search* algorithm for graphs, to traverse the timed automaton, starting from the inital location, to ensure that all reachable zones are created. In each queue element (with the exception of the first element containing the inital location, which has no predecessor), we store a location l_{succ}, its predecessor in the current path l_{pred}, and the transition t from l_{pred} to l_{succ}. It should be noted that this may result in some locations being visited multiple times, and in unreachable locations never being visited. Each time a location is visited, we create therein, new zones which are reachable from the zones of the predecessor. Thus, for each predecessor zone $(l_{pred}, \zeta_{pred_i})$, the derived successor zone is $(l_{succ}, \zeta_{succ_i})$, where

$$\zeta_{succ_i} = ((\zeta_{pred_i} \uparrow \cap t.guard)[t.resets := 0]) \uparrow$$

Thus, if we let $(l_{pred}, \zeta_{pred_i})$ range over the existing zones of l_{pred}, and if we let $(l_{succ}, \zeta_{succ_j})$ range over the existing zones of l_{succ}, then the new zones in l_{succ} will cover

$$\zeta_{succ_{new}} = (\bigcup_i \zeta_{succ_i}) - (\bigcup_j \zeta_{succ_j})$$

Since $\zeta_{succ_{new}}$ is not necessarily convex, we may need to split it into multiple convex polyhedra before creating the corresponding zones in the l_{succ}. Then, we split the zones of l_{succ} to ensure stability with respect to its invariant and outgoing edge constraints. If any new zones are thus created, we enqueue each of the location's successors, in order to ensure that all reachable zones are created. The forward propagation ends when the queue is empty.

In the backward propagation, we iterate through the transitions of the timed automaton, multiple times if necessary, splitting the zones of the source of each transition with respect to the zones of the transition's target, until we achieve stability of each zone of each location. We recall that for stability, whenever we have an edge in the zone valuation graph from a zone (l_{pred}, ζ_{pred}) to (l_{succ}, ζ_{succ}) corresponding to a transition t in the timed automaton, we require

$$\zeta_{pred} \subseteq t.guard \wedge \zeta_{pred}[t.resets := 0] \subseteq \zeta_{succ}$$

Thus, when this does not hold, we split (l_{pred}, ζ_{pred}) into

$$(l_{pred}, \zeta_{pred} \cap (t.guard \cap [t.resets := 0]\zeta_{succ}))$$

(which is convex and has an edge to (l_{succ}, ζ_{succ})) and

$$(l_{pred}, \zeta_{pred} - (t.guard \cap [t.resets := 0]\zeta_{succ}))$$

(which does not have an edge to (l_{succ}, ζ_{succ}) and may need to be split into convex zones.)
This generates the zone valuation graph.

Pseudocode

```
Initialise the queue Q with a single element (null, null, l_0);
Initialise the graph zone_graph with a single node (l_0, v_0 ↑) with an ϵ
self-loop;
while Q is not empty do
    Dequeue (l_parent, t, l_child) from Q;
    if l_parent ≠ null then
        foreach zone Z_parent of l_parent do
            Add new zones to the zones of l_child so that all zones reachable
            from Z_parent are represented;
            Abstract if necessary;
            Update edges from Z_parent to the new zones of l_child if new
            zones are created in l_child or l_parent is null then
                foreach outgoing transition t' of l_child do
                    Enqueue (l_child, t', t'.target) in Q;
                end
            end
        end
    end
    Set new_zone;
    while new_zone do
        Reset new_zone;
        foreach transition t in the timed automaton do
            Split the zones of t.source to be stable with respect to the
            zones of t.target;
            Update edges accordingly;
            if new zones are created in t.source then
                Set new_zone;
            end
        end
    end
end
Return zone_graph;
```

Proof of correctness

- `Termination`: The algorithm, in the worst case, will create as many zones as there are regions in the region graph, as the region graph abstraction ensures that the number of zones cannot exceed the number of regions. Since the number of regions is known to be bounded, termination of the algorithm is guaranteed.
- `Reachability`: Since the initial zone is the future of the zero valuation in the initial zone, it is reachable by definition. A new zone is only created when it is reachable from some zone which has already been created, thus each zone which is created is reachable by induction.
- `Stability`: Since the termination of the backward propagation step only occurs after an iteration in which all the edges of the timed automaton are traversed without causing any splitting of states, it follows that the zone graph is stable with respect to itself after this last iteration.

6.2 Abstraction

Overview From the description of the above algorithm, it is evident that the forward propagation may continue indefinitely if implemented in a naive fashion. For example, in the automaton in Figure 4a, the number of zones may expand in each iteration, as shown in Figure 4b, Figure 4c, Figure 4d.

However, this is inconsistent with what we know about region graphs [6] and their implication of finiteness for the state spaces of timed automata, thus, we have abstractions which serve to cap the number of zones in a zone graph by reducing zones to equivalent regions which contain them, thus ensuring termination of zone creation algorithms.

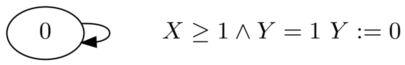
In this implementation we use a simplified version of the *maximum constants* abstraction described in [2].

Algebraically, our abstraction can be thus described: given a set of clocks $\{x_i | 1 \leq i \leq n\}$, a maximum constant k over all clocks, and a DBM $M = \langle m_{ij}, \smile_{ij} \rangle_{0 \leq i,j \leq n}$, we can replace M with $M' = \langle m'_{ij}, \smile'_{ij} \rangle_{0 \leq i,j \leq n}$ where

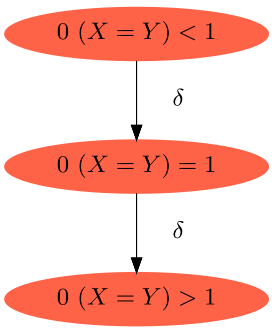
$$(m'_{ij}, \smile'_{ij}) = \begin{array}{ll} (\infty, <) & \text{if } m_{ij} > k \\ (-k, <) & \text{if } m_{ij} < -k \\ m_{ij}, \smile_{ij} & \text{if } -k \leq m_{ij} \leq k \end{array}$$

References

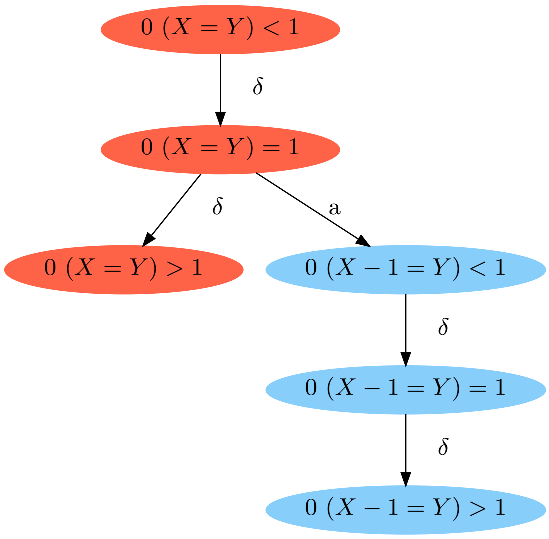
1. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive systems: modelling, specification and verification. Cambridge University Press (2007)
2. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: In TACAS, Springer (2003) 254–277
3. Guha, S., Narayan, C., Arun-Kumar, S.: On decidability of prebisimulation for timed automata. In Madhusudan, P., Seshia, S.A., eds.: CAV. Volume 7358 of Lecture Notes in Computer Science., Springer (2012) 444–461
4. Tripakis, S., Yovine, S.: Analysis of timed systems using time-abstracting bisimulations. Formal Methods in System Design **18**(1) (2001) 25–68



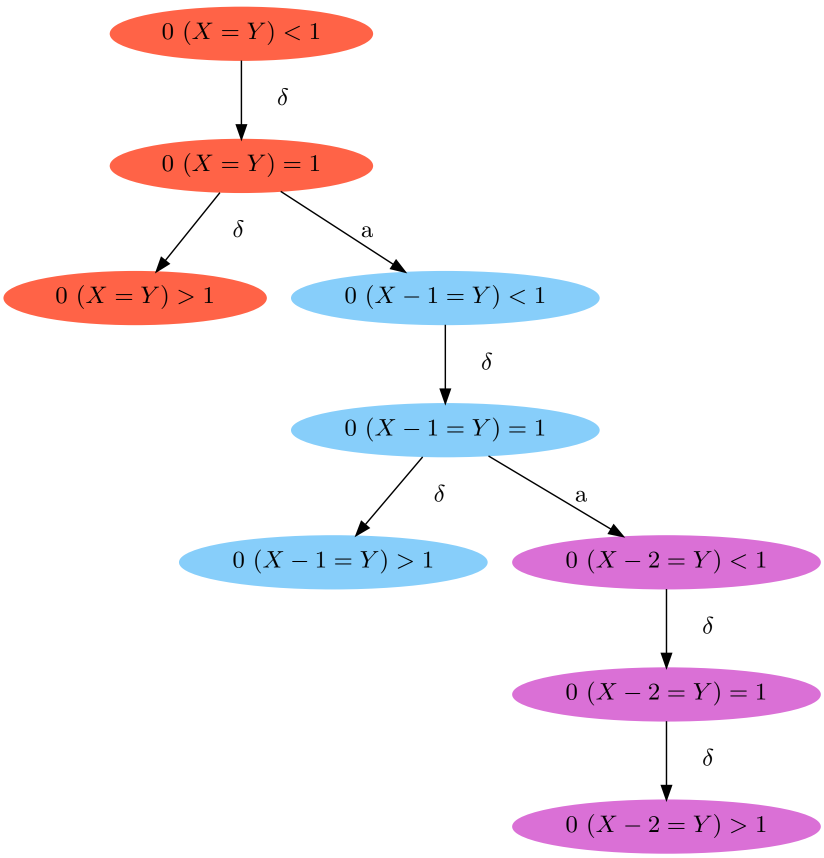
Timed automaton with potentially infinite state space.



Zones of Figure 4a after 1 iteration.



Zones of Figure 4a after 2 iterations.



Zones of Figure 4a after 3 iterations.

Fig. 4: Timed automaton with a potentially infinite set of zones, example taken from [2].

5. Keller, R.M.: Formal verification of parallel programs. Commun. ACM **19**(7) (July 1976) 371–384
6. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126** (1994) 183–235
7. Guha, S.: An algorithm to generate zone valuation graph. unpublished