Master research Internship

Bibliographic report

Abstraction Refinement for Weighted Timed Automata

Domain: Formal Languages and Automata Theory

Author:
Victor Roussanaly

Supervisor:
Nicolas Markey
Ocan Sankur
SUMO

Abstract: Timed automata extend finite automata with a finite number of variables called clocks, which are used to explicitly model durations and time constraints. They have become a standard model in the formal verification of real-time systems; model-checking tools such as Uppaal have been developed and used to verify several non-trivial systems. In this internship, we suggest studying a counter-example-guided abstraction refinement scheme to solve the quantitative analysis problem for timed automata both more efficiently, and in a broader setting. The idea is simple: one starts by analyzing a coarse abstraction of the system (obtained for instance by collapsing states). If the abstract model satisfies the property, then this implies the same for the original system. If the property is not satisfied in the abstraction, then a refinement is needed.

Contents

1 Introduction

In real time modeling and verification, several formalisms are used such as Petri nets and real time logics. Another popular model is *timed automata*. First introduced in [1], a timed automaton is a finite automaton to which one adds clocks that are modeled by real-valued variables, and the behavior of the automaton is restricted by constraints on these clocks. Nowadays, the model that is studied a lot, as in [2], and especially in this internship, is *timed safety automata*. These automata are used in the model checker UPPAAL, and since they are the only ones we are studying here, we will refer to them simply as timed automata.

Usually, a model induces a transition system that will be checked for specific properties by the model checker. This transition system needs a finite number of states, which is not trivial in timed automaton due to the clocks that can have an infinite uncountable number of values. In order to have a finite transition system, we need to use abstractions, that will aggregate some states together. This will allow us to have a transition system that we can check, but since we have fewer states, the information on the system is less precise. Most abstractions are based on zones and regions in the space of values of clocks, as detailed in [2] and [3]. Reducing the number of states in the transition system is important to allow for faster model checking and we believe that this is achievable with more aggressive abstractions.

A popular method is the *counterexample guided abstraction refinement*. This method, detailed in [4], is a process that gives a first abstraction that is effective but not very informative, then check for the desired property. If the property is not satisfied in a specific run of the abstracted system, then we check if the run is present in the original system. If the run is not possible, we call the run *spurious*, meaning that it does not imply that the original system does not satisfies the property, and we refine the abstraction to eliminate this run. This method is used in [5] for checking the validity of ACTL* formulas on Kripke structures. It is also used in [6] to check for winning strategies in two-players games. The goal of the internship is to apply this paradigm to timed automata.

First we will explain the abstraction paradigm and give an example on ACTL*, then we will introduce timed automata. Finally we will show the abstractions that already exists on these automata and explain what abstractions we want to add in this internship.

2 Abstraction

Abstraction is a process used a lot in model checking. Usually we have a model and we want to check whether or not this model respect some properties. Since models can have an important number of states, the computation time can be too long, and abstraction is used to reduce the size of the model without losing the properties we want to check. In [4], a method called *counterexample guided abstraction refinement* is detailed as a three steps process:

- First we abstract the model. We reduce the number of states, giving a model that has a smaller size to reduce the computing time. Some states, for example s_1 and s_2,

will be aggregated into a state s. When a transition is possible from s_1 and not from s_2, it will still be possible for the abstracted state s, meaning that some runs in the abstraction are not possible in the original model.

- We run the abstracted model to check for the property we are studying.
- If there is a bad behavior, meaning a run that doesn't satisfy the property, we check whether or not it is present before the abstraction. If it is not the case, it means that we lost important properties in our abstraction and we need to refine it and start over.

This paradigm is used in [5] and [6]. We will now show how it is applied in other domains in order to explain how we will use it in this internship.

2.1 Example on ACTL* and Kripke structures

Kripke structure. In [5], it is explained how to use abstraction for model checking on ACTL* formula. ACTL* is a subset of CTL* that gives temporal logic on paths, but without the use of the existential operator. Formally it is applied on Kripke structures which are defined as a tuple $M = (S, R, I, L)$ over a set of atomic propositions A where:

- S is a set of states.
- $R \subseteq S^2$ is the set of transitions.
- $I \in S$ is the set of initial states.
- $L : S \to 2^A$ is a labeling function giving to each state the atomic proposition it satisfies.

A path is an infinite sequence of states $\pi = (s_0, s_1...)$ such that $\forall 0 \leq i, (s_i, s_{i+1}) \in R$. We denote for all $i \in \mathbb{N}$ by $\pi_{\geq i}$ the suffix of π starting at s_i.

Semantic of ACTL*. CTL* contains two types of formulas : state formulas and path formulas. We denote by ϕ the state formulas, and by Φ the path formulas. A state formula is $\phi = a|\phi_1 \wedge \phi_2|\neg a|\phi_1 \vee \phi_2|\forall\Phi|\exists\Phi$ and a path formula is defined as $\Phi = \phi|\circ\Phi'|\Diamond\Phi'|\Box\Phi'|\Phi_1 \mathbf{U} \Phi_2$. In a Kripke structure M, for a state s or a path $\pi = (s_0, s_1, ...)$, we say

- $s \models a$ iff $a \in L(s)$
- $s \models \neg a$ iff $a \notin L(s)$
- $s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$
- $s \models \phi_1 \vee \phi_2$ iff $s \models \phi_1$ or $s \models \phi_2$
- $s \models \forall\Phi$ iff for all path $\pi = (s, ...)$, $\pi \models \Phi$
- $s \models \exists\Phi$ iff there is a path $\pi = (s, ...)$ such that $\pi \models \Phi$

- $\pi \models \phi$ iff $s_0 \models \phi$.
- $\pi \models \circ\Phi$ iff $\pi_{\geq 1} \models \Phi$.
- $\pi \models \Box\Phi$ iff $\forall i \in \mathbb{N}$, $\pi_{\geq i} \models \Phi$.
- $\pi \models \Diamond\Phi$ iff $\exists i \in \mathbb{N}$, $\pi_{\geq i} \models \Phi$.
- $\pi \models \Phi_1 \mathbf{U} \Phi_2$ iff $\exists k \in \mathbb{N}$, $\pi_{\geq k} \models \Phi_2$ and $\forall i < k$, $\pi_{\geq i} \models \Phi_1$.

Simulation and abstraction. If for all initial states $s \in I$, $s \models \phi$ then M respect the property ϕ, denoted by $M \models \phi$. ACTL* is the fragment of CTL* such that the operator $\exists$ is not used. We say that M' simulates M, denoted by $M \preceq M'$ iff $A \subseteq A'$ and there is a relation $H \in S \times S'$ such that :

- for all $s \in I$ there is a $s' \in I'$ such that $(s, s') \in H$
- for all $(s, s') \in H$, $L(s) \cap A' = L'(s')$
- for all $(s, s') \in H$, if $(s, s_1) \in R$, then there is $s'_1 \in S$ such that $(s', s'_1) \in R$ and $(s_1, s'_1) \in H$.

If $M \preceq M'$, ϕ an ACTL* formula and $M' \models \phi$, then $M \models \phi$. We will then, for a given Kripke structure M, build an abstraction M' such that $M \preceq M'$. This gives that if M' satisfies an ACTL* property then M satisfies this property too. However, it can happen that there is a property ϕ that is not satisfied by M' but that is satisfied for M. In that case, we need to refine the abstraction to eliminate the run of M' that invalidates ϕ. In [5], an algorithm for finding this counterexample and refining the abstraction is given.

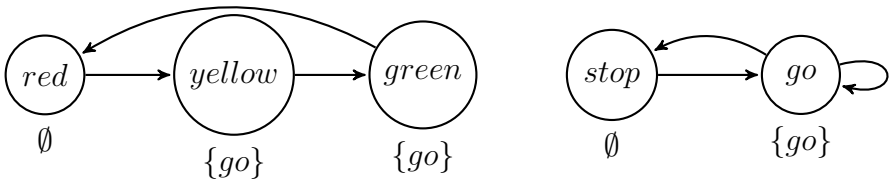

Figure 1: An example of a Kripke structure for a traffic light M_{light} (on the left) and its abstraction M_{abs} (on the right).

In Figure 1, we show two examples of Kripke structures. We have $H = \{(red, stop); (yellow, go); (green, $ as a simulation relation and M_{abs} simulates M_{light}. If we look at $\phi = \forall\Diamond(\neg go)$, we have $M_{light} \models \phi$. But we do not have $M_{abs} \models \phi$ because the path $stop, go, go, go, go, ...$ is possible. So we have an example of an abstraction that does not satisfy a property but the counterexample run is not possible on the original structure and we need a refinement. Whether or not we need a refinement depends on the formula we are trying to verify. If we look at the formula $\forall\Diamond go$, then both structures satisfy the property and the refinement is not needed. The refinement part of their algorithm tries to give the coarsest abstraction, which is a NP-hard problem.

2.2 Other examples

Two-player games. The method developed in [6] is similar. It is used to find a winning strategy in a two-player game which can be seen, in short, as a Kripke structure where the states are partitioned in two sets, one for player 1 and the other for player 2. Formally a two-player game is a tuple (V_1, V_2, R, L) where:

- V_1 is the set of states belonging to player 1, V_2 is the set states belonging to player 2, and we write $V = V_1 \cup V_2$.
- $R \subseteq V \times \Lambda \times V$ is the set of transitions labeled by elements of Λ
- $L : V \to 2^{AP}$ a labeling function.

The way it works is that when in a state of V_1 (respectively V_2), the player 1 (resp. the player 2) chooses the next move. The sequence of moves builds a run and usually we want to check safety properties of the form $\neg\Box err$ where $err \in AP$. More precisely we want to know if the player 1 has a strategy to satisfy this property, meaning if the player 1 has a decision tree that allows him, whatever the strategy of player 2 is, to satisfy the property. Since two-player games are similar to Kripke structures, we can build an abstracted game the same way we built an abstracted structure, but in this case the counterexamples are not just runs but a decision tree representing the counter-strategy for player 2. If player 1 has winning strategy in the abstracted game, then it has a winning strategy in the original game. Otherwise player 2 has a counter-strategy in the abstracted game and we have to check whether or not it also works in the original game. If it's not the case, then it is spurious and the abstracted game needs to be refined.

Lazy abstraction. Another example of counterexample guided abstraction refinement can be found in [4]. This paper describes an algorithm in order to have automatic verification abstraction of C programs. This paper adds to the method the fact that the entire system may not need to be refined if there is a spurious run. Their example is based on the fact that C programs can be modelled by control flow graphs. After abstraction we search for spurious runs. If there are spurious runs, we only look at the abstracted states they are going through, and those are the states that we are refining. This is a bit better than the other abstractions we have shown because some parts of the control flow graph may stay coarser if information is not needed on those parts.

3 Timed Automata

Now that we have given an example of abstraction, we will introduce timed automata. A timed automaton is basically a finite automaton to which we add clocks. Clocks are variables that increase synchronously, on which we can have conditions and that can be reset.

Formal definition. Let C a set of clocks. An *atomic clock constraint* is a condition on the clocks $x \circ n$ or $x - y \circ n$ where x, y are clocks in C, $n \in \mathbb{N}$ and $\circ \in \{<, >, \leq, \geq, =\}$. A *clock constraint* is a conjunctive formula of atomic clock constraints on the clock We denote by $B(C)$ the set of constraints on the clocks of C, . Formally a timed automaton, as defined in [2] is a tuple (N, l_0, E, I) over a set of clocks and an alphabet (C, Σ) with

- N a finite set of locations.
- $l_0 \in N$ the initial location.
- $E \subseteq N \times \mathcal{B}(C) \times \Sigma \times 2^C \times N$ the transitions. They are labeled by a letter, boolean conditions over clocks and a set of clocks to reset.
- $I : N \to \mathcal{B}(C)$ the invariants. It attributes to each states boolean conditions over clocks.

Usually a timed automaton would be represented as seen in Figure 2. The transitions are labeled by a clock constraint, a letter of the alphabet, and the clocks that are reset. The initial state is pointed by an arrow (l_0 in this case), and the invariants are indicated under the states.

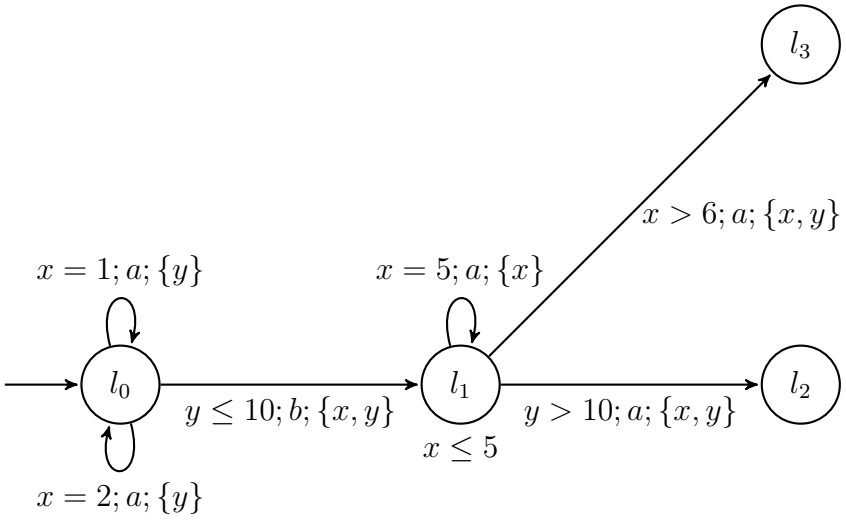

Figure 2: An example of a timed automaton on $(\{x; y\}, \{a; b\})$.

Semantic and language. We can give an operational semantic for timed automata by using a transition system. States are (l, u) with u a valuation of the clocks and $l \in N$. The transitions are :

- $(l, u) \xrightarrow{d} (l, u + d)$ with $d \in \mathbb{R}^*$ if $(u + d) \in I(l)$. By $u + d$ we mean the valuation of the clocks u where all the clocks have been incremented by d. These transitions represent waiting a delay d in the location l.

- $(l,u) \xrightarrow{a} (l',u')$ if $l \overset{g,a,r}{\to} l'$, $u \in g$, $u' = u[r \to 0]$ and $u' \in I(l')$. It corresponds to taking a transition in the timed automaton.

For example in our timed automaton in Figure 2, we can have a run
$(l_0,(0,0)) \xrightarrow{1} (l_0,(1,1)) \xrightarrow{a} (l_0,(1,0)) \xrightarrow{b} (l_1,(0,0)) \xrightarrow{5} (l_1,(5,5)) \xrightarrow{a} (l_1,(0,5)) \xrightarrow{5} (l_1,(5,10)) \xrightarrow{a}$
$(l_1,(0,10)) \xrightarrow{5} (l_1,(5,15)) \xrightarrow{b} (l_2,(0,0))$.

Note that in a run of a timed automaton, we can pair the transitions in (d,a) where d is a waiting action and a is the action of taking a transition in the timed automaton, with sometimes $d = 0$.

A timed action is a pair (t,a) where $t \in \mathbb{R}^+$ and $a \in \Sigma$, meaning that the action a is taken at time t. A timed trace is a sequence of timed actions $\xi = (t_1,a_1),(t_2,a_2),...$ where $t_i \leq t_{i+1}$. A run over ξ, where (l_0,u_0) is the initial state, is the run $(l_0,u_0) \xrightarrow{d_1}\xrightarrow{a_1} (l_1,u_1) \xrightarrow{d_2} \overset{a_2}{\cdot}..$ where $d_i = t_i - t_{i-1}$ with $t_0 = 0$. For a timed automaton A, the set of timed traces for which there exists a valid run in A defines the timed language accepted by A denoted by $L(A)$.

Decidability results. Many problems are undecidable for timed automata, such as the *language inclusion* problem that is to verify for two timed automaton A_1 and A_2 whether or not $L(A_1) \subseteq L(A_2)$. But the problem that is interesting in model checking is *the reachability problem*, asking whether or not it is possible to reach a final state l_f with a constraint $g_f \in B(C)$. This is equivalent to check whether or not $L(A) = \emptyset$. This problem is decidable [1] and allow us to check for safety property, but also for invariant properties and bounded liveness properties. It is solved in the model-checker UPPAAL [7] [8], first released in 1995. It implements most of the method we will describe in the next part. Since the main problem that interests us is the reachability problem, we will not be interested in the alphabet and the language.

In the long term, we want to look at weighted timed automata, in which case the alphabet is replaced by a cost function that associates to each transition a cost, and the goal becomes minimizing the cost of the run. The states can also have a cost associated to them, and waiting in a state will increase the cost of the run. We will not introduce and detail the weighted automata in this report, since the main goal of the internship is to add abstraction to timed automata, the weighted part being something we will add later on, but a formalism for weighted timed automata can be found in [9]. Some reachability results can be found in [10], using methods of abstraction similar to the ones we are going to detail in the next part.

The complexity of the reachability problem depends on the size of our transition system and as you can see, the state-space of our transition system is infinite. In order to have a finite space we need a first abstraction, to consider only the relevant intervals for the values of our clocks. We might also want more abstraction the reduce the number of states and the computing time of the model-checker.

4 Abstraction for timed automata

4.1 Regions

In order for us to define a region we need a *clock ceiling function*, denoted by k, that associates to each clock x a maximum value $k(x)$. A region is an equivalence class on $\sim$ where $\sim$ is an equivalence relation on clock evaluation defined by $u \sim v$ iff:

- for all clock x, $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or both $u(x) > k(x)$ and $v(x) > k(x)$.
- for all clock x, if $u(x) \leq k(x)$ then $frac(u(x)) = 0$ iff $frac(v(x)) = 0$.
- for all clocks x and y, if $u(x) \leq k(x)$ and $u(y) \leq k(y)$ then $frac(u(x)) \leq frac(u(y))$ iff $frac(v(x)) \leq frac(v(y))$.

From there we can write $[u]_\sim$ the equivalence classe of u, and we can build a transition system with a transition $\Rightarrow$ defined as :

- $(l, [u]_\sim) \Rightarrow (l, [v]_\sim)$ if $(l, u) \xrightarrow{d} (l, v)$ for d a positive number.
- $(l, [u]_\sim) \Rightarrow (l, [v]_\sim)$ if $(l, u) \xrightarrow{a} (l, v)$ for $a \in \Sigma$.

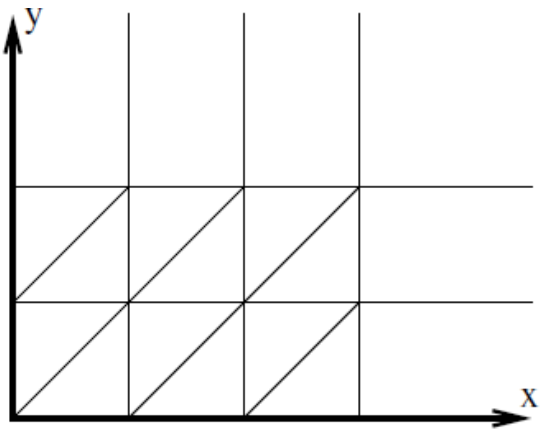

Figure 3: Example of regions for 2 clocks with $k(x) = 2$ and $k(y) = 3$.

In Figure 3 we show a representation of these regions. Every surface, but also every segment are regions in this case, which means that we have 60 regions for only 2 clocks. Since the number of region is finite, then the transition system $\Rightarrow$ is finite. We can also note that this is an abstraction of the transition system we defined earlier. If k is chosen such that $k(x)$ is the higher value that x is compared to in the timed automaton, we have a theorem:

Theorem 1. *For a timed automaton A with an initial state l_0, u_0, for $l_f \in N$, and u_f a clock valuation, $(l_0, u_0) \rightarrow^* (l_f, u_f)$ iff $(l_0, [u_0]_\sim) \rightarrow^* (l_f, [u_f]_\sim)$.*

This means that the reachability problem for a timed automaton is equivalent to the reachability problem for the region abstraction. Let us denote by $R(A,k)$ the number of regions for the timed automata A with a ceiling function k. In [1] it is explained that $R(A,k)$ can be bounded by $|C|! \cdot 2^{|C|} \cdot \prod_{x \in C}(2k(x)+2)$. Then the transition system is computed in $O((|N|+|E|) \cdot R(A,k))$ time and the reachability problem can be solved in $O((|N|+|E|) \cdot R(A,k))$ time. For a given timed automaton A with an initial state (l_0, u_0), $l_f \in N$ and r_f a region, we detail Algorithm 1 that gives us whether or not (l_f, u_f) with $u_f \in r_\sim$ is reachable.We can also note the important results that the reachability problem for timed automata is PSPACE-complete [1]. As we said, the number of regions can be quite

Algorithm 1 Algorithm for reachability problem using regions

compute $\Rightarrow$ from A and k
$WAIT = \{(l_0, [u_0]_\sim)\}$
$PASSED = \emptyset$
while $WAIT \neq \emptyset$ **do**
  take (l,r) from $WAIT$
  if $l = l_f \wedge r = r_f$ **then**
    Stop and return YES
  end if
  if $(l,r) \notin PASSED$ **then**
    add (l,r) to $PASSED$
    for all (l',r') such that $(l,r) \Rightarrow (l',r')$ **do**
      add (l',r') to $WAIT$
    end for
  end if
end while
return NO

important which is a problem for the time complexity of the algorithm. To speed up the algorithm, we might try to aggregate some of the regions together. To this aim, we will use *zones*.

4.2 Zones

A zone is defined by a constraint on the clocks. It can be viewed as the set of clocks valuation that satisfies the constraint. Because a set of atomic constraints is something that is not easy to manipulate when we implement zones, they are often represented as matrices called Difference Bound Matrices (DBM) [11].

If we denote $C_0 = C \cup \{\mathbf{0}\}$ where $\mathbf{0}$ is a reference clock always set to the value 0, we can express atomic constraints as $x - y \preceq n$ where $x, y \in C_0, n \in \mathbb{Z}, \preceq \in \{<, \leq\}$. A zone D is then defined by less than $|C_0|^2$ atomic constraints which can be represented in a matrix. If the clocks are noted x_i, $0 < i \leq |C|$ with $x_0 = \mathbf{0}$ then we have $D_{i,j} = (k, \preceq) \in (\mathbb{Z} \cup \{\inf\}) \times \{<, \leq\}$ meaning that we have the atomic constraint $x_i - x_j \preceq k$.

It is worth noting that several matrix represents the same zones. For example the constraint $(x-y \leq 0) \wedge (y-x \leq 0) \wedge (x \leq 1)$ and $(x-y \leq 0) \wedge (y-x \leq 0) \wedge (y \leq 1)$ are the same in term of possible clocks valuation. That is why we need a canonical representation of the zones and a zone is canonically represented by the unique constraint such that adding another atomic constraint will change the possible set of clocks valuation. In this case, the canonical representation is $(x-y \leq 0) \wedge (y-x \leq 0) \wedge (x-\mathbf{0} \leq 1) \wedge (\mathbf{0}-x \leq 0) \wedge (\mathbf{0}-y \leq 0) \wedge (y-\mathbf{0} \leq 1)$. This can be computed by considering the set of atomic constraints as a graph where the clocks are the edges and the atomic constraints are the edges. The canonical form of the zone is then computed by using a shortest path algorithm, usually Floyd-Warshall algorithm [12]. In the rest of the report a zone D is a constraint, but also the set of clocks valuation that satisfies this constraint.

Let D a zone and r a set of clocks, we define $D^{\uparrow} = \{u + d \mid u \in D, d \in \mathbb{R}^+\}$ and $D^r = \{u[r \to 0] \mid u \in D\}$. We can now define a transition system on the symbolic states composed of a state and a zone with the transition $\hookrightarrow$ defined as:

- $(l, D) \hookrightarrow (l, D^{\uparrow} \wedge I(l))$.
- $(l, D) \hookrightarrow (l', (D \wedge g)_r \wedge I(l'))$ if $l \overset{g,a,r}{\to} l'$.

There is a theorem in [5] that ensures that the zone transition system is a valid abstraction for reachability analysis.

Theorem 2. *For a timed automaton A with initial state (l_0, u_0), for $l_f \in N$ and D_f a zone*

- $(l_0, \{u_0\}) \hookrightarrow^* (l_f, D_f)$ *implies* $(l_0, u_0) \to^* (l_f, u_f)$ *for all* $u_f \in D_f$.
- $(l_0, u_0) \to^* (l_f, u_f)$ *implies* $(l_0, \{u_0\}) \hookrightarrow^* (l_f, D_f)$ *for some* D_f *such that* $u_f \in D_f$.

The problem here is that the zone-based transition system is possibly infinite, as shown in Figure 4, which shows the zone infinite transition system for the timed automaton shown in Figure 2. It is worth noting that some information has been lost in this first abstraction. For example, while the regions abstraction would have differentiated the case $l_0, x \leq 1$ and $l_0, x \geq 1$, here these cases are represented in a single state and the fact that we need $x \leq 1$ to take the transition labeled by a does not appear in this transition system.

4.3 Reducing the number of zones

In order to limit ourselves to a finite system, we apply the same method we used for regions. Let k a clock ceiling. We denote by $[D]_k$ the zone obtained from D as follows:

- We remove every atomic constraints $x - y < m$, and $x - y \leq m$ where $m > k(x)$.
- We replace all the constraints $x-y > m$ and $x-y \geq m$ where $m > k(x)$ by $x-y > k(x)$.

We define a new transition system with a transition $\hookrightarrow_k$ defined by $(l, D) \hookrightarrow_k (l', [D']_k)$ iff $(l, D) \hookrightarrow (l', D')$. Then we have

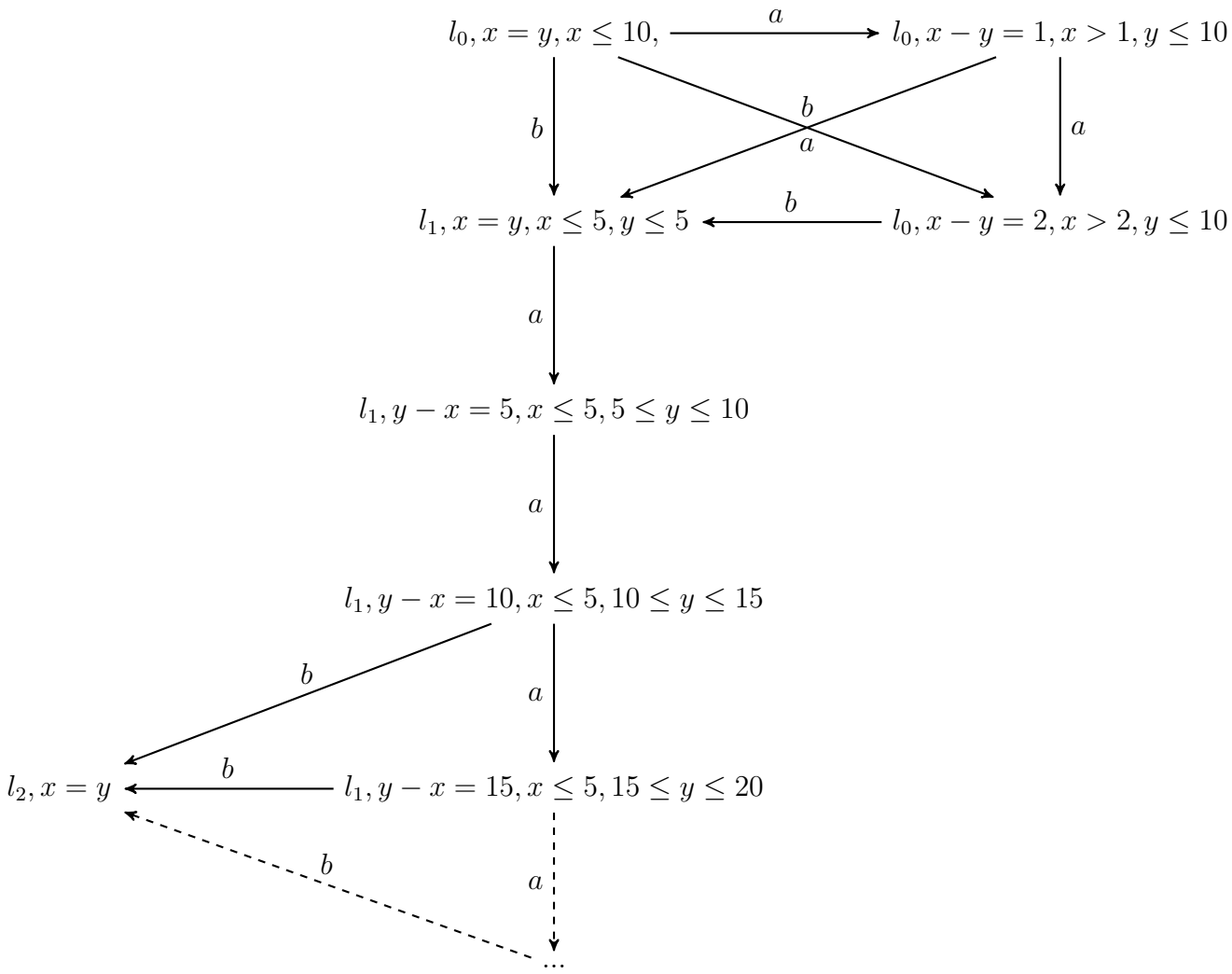

Figure 4: An infinite transition system for the zones of our example automaton.

Theorem 3. *For a timed automaton A with initial state (l_0, u_0) and no constraints of the type $x - y \circ n$, for $l_f \in N$ and D_f a zone*

- $(l_0, \{u_0\}) \hookrightarrow_k^* (l_f, D_f)$ *implies* $(l_0, u_0) \rightarrow^* (l_f, u_f)$ *for all* $u_f \in D_f$ *such that* $\forall x \in C, u_f(x) \geq k(x)$.
- $(l_0, u_0) \rightarrow^* (l_f, u_f)$ *with* $\forall x \in C, u_f(x) \geq k(x)$ *implies* $(l_0, \{u_0\}) \hookrightarrow_k^* (l_f, D_f)$ *for some* D_f *such that* $u_f \in D_f$.
- *The transition system defined by* $\hookrightarrow_k$ *is finite.*

In [3], a variant of this abstraction is used, where instead of one ceiling function, we use two functions L and U. We denote by $[D]_{L,U}$ the zone D that we modified such that :

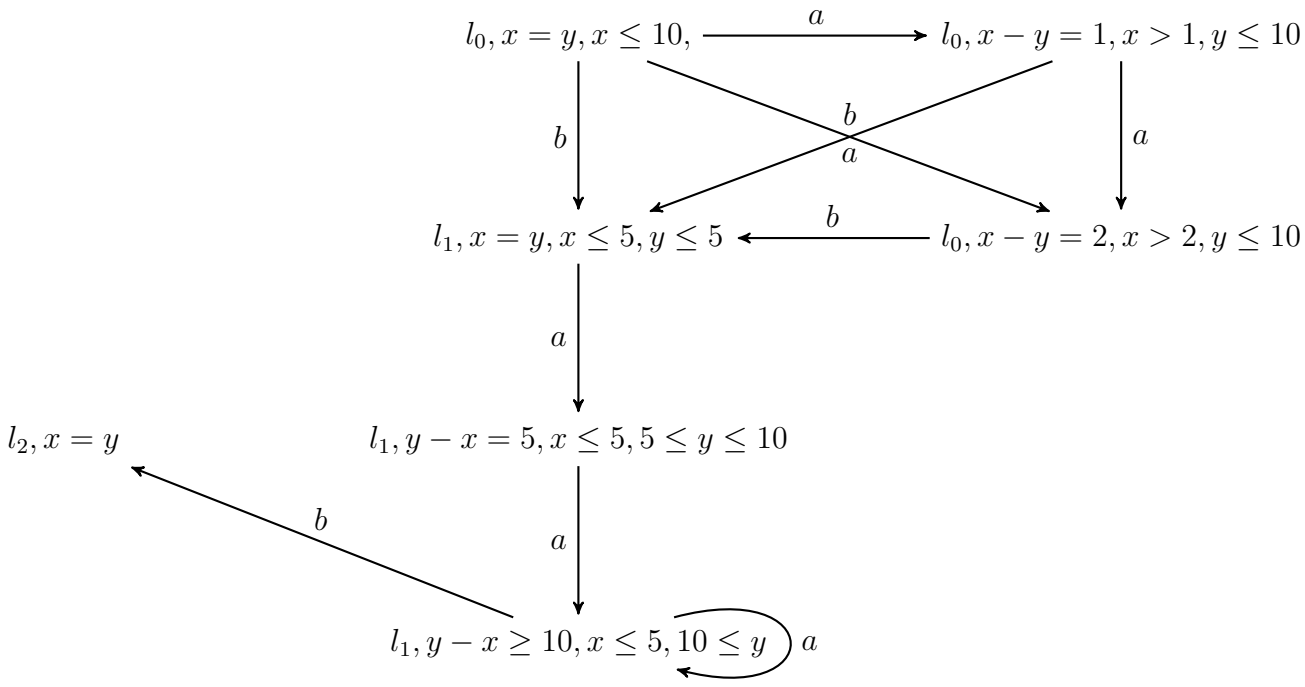

Figure 5: An finite transition system for the zones of our example automaton.

- We remove all constraints $x - y < m$, and $x - y \leq m$ where $m > L(x)$.
- We replace all the constraints $x - y > m$ and $x - y \geq m$ where $m > U(x)$ by $x - y > U(x)$.

This also gives us another abstracted transition system defined by $\hookrightarrow_{L,U}$ and Theorem 3 holds for $\hookrightarrow_{L,U}$ if $L(x)$ is defined as the lowest value k such that $x > k$ or $x \geq k$ appears in a guard or invariant of the timed automata, and $U(x)$ is defined as the greatest value k such that $x < k$ or $x \leq k$ appears in a guard or invariant of the timed automata. This abstraction is a bit coarser but need a deeper analysis of the automaton to find L and U.

In both cases we only gave the method in the case where there is no constraints of the type $x - y \circ n$ in the automaton. In the case there is that kind of constraints, there is a variant of the abstraction that can be found in [3], but we will not use this case in the internship, so we will not detail it here. Now that we have a finite transition system, we can modify Algorithm 1 to solve the reachability problem. The algorithm is very similar, but the abstraction we did aggregated the regions, which can provide fewer reachable states in the transition system. The number of zones can be greater than the number of regions, but if we look at zones inclusion, we can speed up the reachability analysis alogorithm. In Algorithm 2 we denote by A the automaton, (l_0, D_0) the initial state, l_f the state we are trying to reach and D_f the conditions on clocks we want to have when we reach l_f.

Even though this algorithm is pretty efficient, and the details of the implementation can be found in [2], the goal of the internship is to use the method described in Section 2 to have

Algorithm 2 Algorithm for reachability problem using zones

compute $\hookrightarrow$ from A and k (or L, U)
$WAIT = \{(l_0, D_0)\}$
$PASSED = \emptyset$
while $WAIT \neq \emptyset$ **do**
  take (l, D) from $WAIT$
  if $l = l_f \wedge D \subseteq D_f$ **then**
    Stop and return YES
  end if
  if $D \not\subseteq D'$ for all $(l, D') \in PASSED$ **then**
    add (l, D) to $PASSED$
    for all (l', D') such that $(l, D) \hookrightarrow (l', D')$ **do**
      add (l', D') to $WAIT$
    end for
  end if
end while
return NO

a coarser abstraction.

5 Goal of the internship

We have explained the counterexample guided abstraction refinement, and detailed some abstractions that already exists on timed automata. Even though these abstractions allow for model checking, we want to use the counterexample guided abstraction refinement to reduce the number of states in our abstraction, and allow for faster model checking. The main idea, that stems from [13], is to have less granularity when describing the zones by increasing the size of the intervals. For example, we can only consider the integers that are multiples of 5. In our examples that would fuse the zones $l_0, x - y = 1, x > 1, y \leq 10$ and $l_0, x - y = 2, x > 2, y \leq 10$ into $l_0, 0 < x - y < 5, x > 0, y \leq 10$ and this would not change the reachability properties. On the other hand, properties that are not reachability-based would be changed.

We can note that this method needs refinement because abstractions can be too coarse. For example if we only consider the integers that are multiple of 10, we would have the two zones we are interested in that would be aggregated, and l_2 would still be reachable. But on the other hand, the state l_3 would also be reachable, which is not the case in our original automaton. In this case we want to have an algorithm that catch the fact that this state is reachable in the abstraction but not in the original automata, and then refines the abstraction to have no spurious runs. Something we are also interested in is the fact that maybe there might be a part of the automaton that benefits from the coarser abstraction, and keeps the refinement only on the part of the automaton that needs it. For example

on the example automaton, only the clock x in the state l_1 is a problem in the abstraction being too coarse. In this way we want to apply the refinement method that can be found in [4], that is called *lazy abstraction*. In this paper, the method detailed only refines the part of the abstraction that produces spurious runs, instead of having to refine the entire abstraction. The goal of the internship is to creates a lazy abstraction for timed automata, and implements it. In [14], a variant of timed automata, called *updatable timed automata*, is introduced, and although this is not the automata we are studying, we hope to use some of their methods and abstraction to reach our goal. Another goal will be to extend it to weighted timed automata, which carry more information that we will have to abstract.

References

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, pages 87–124, 2003.

[3] Gerd Behrmann, Patricia Bouyer, Kim G. Larsen, and Radek Pelánek. *Lower and Upper Bounds in Zone Based Abstractions of Timed Automata*, pages 312–326. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[4] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM.

[5] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[6] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Counterexample-guided control. In Jos C. M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, volume 2719 of *Lecture Notes in Computer Science*, pages 886–902. Springer-Verlag, June 2003.

[7] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[8] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. *Uppaal in 1995*, pages 431–434. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[9] Patricia Bouyer. Weighted timed automata: Model-checking and games. *Electron. Notes Theor. Comput. Sci.*, 158:3–17, May 2006.

[10] Patricia Bouyer, Maximilien Colange, and Nicolas Markey. Symbolic optimal reachability in weighted timed automata. *CoRR*, abs/1602.00481, 2016.

[11] David L. Dill. *Timing assumptions and verification of finite-state concurrent systems*, pages 197–212. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990.

[12] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[13] Rajeev Alur, Thao Dang, and Franjo Ivani. Counterexample-guided predicate abstraction of hybrid systems. *Theoretical Computer Science*, 354(2):250 – 271, 2006. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)Tools and Algorithms for the Construction and Analysis of Systems 2003.

[14] Patricia Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design*, 24(3):281–320, 2004.