Revisiting Bounded Reachability Analysis of Timed Automata Based on MILP

Iulian Ober

Revisiting Bounded Reachability Analysis of Timed Automata Based on MILP

Iulian Ober(✉)

University of Toulouse - IRIT,
118 Route de Narbonne, 31062 Toulouse, France
iulian.ober@irit.fr

Abstract. We study the reduction of bounded reachability analysis of timed automata (TA) to a Mixed Integer Linear Programming (MILP) problem. While bounded model checking of timed automata has been explored in the literature based on the satisfiability of Boolean constraint formulas over linear arithmetic constraints verified using SAT Modulo Theory (SMT) solvers, the approach presented in this paper opens up the alternative of using MILP solvers. We present some preliminary results comparing the two approaches and provide ideas on how linear optimization can be useful for analyzing the behavior of TA. The results are supported by a prototype implementation which relies either on a MILP solver (Gurobi) or an SMT solver (MathSAT). Certain techniques for reducing the search space and improving the performance are also discussed.

1 Introduction

Timed automata [1] allow the specification of time-dependent behavior and they have been used as underlying semantic model for real-world, industry-grade languages used in the design and analysis of real-time systems, such as SDL [7,14] and extensions of UML [11,15,17]. As S. Graf remarked in [14], "at the semantic level, it is interesting to have a minimal number of basic primitives allowing expression of all concepts" [related to time], and timed automata primitives fill this need both for functional design elements and for non-functional aspects.

Since the applications for these models are often safety-critical (e.g., real-time systems, communication protocols), their formal verification has received wide attention in the research literature. There are several mature tools for verifying or simulating various flavors of timed automata-based models, including [3,8,26,27]. Although timed automata give raise to infinite state spaces due to the dense domain of time, both reachability and model checking of various logics are decidable based on finite representations of the state space. The tools and analysis methods cited above rely on symbolic representations of state sets, such as the Difference Bound Matrices (DBMs, introduced in [12]), or more efficient ones such as CDDs, RED [18,26].

Bounded model checking (BMC) [5] on the other hand is a successful method for analyzing models that yield very large state spaces. It relies on encoding the

https://doi.org/10.1007/978-3-030-00244-2_18

next-state relationship as a logical formula and on instantiating this formula a bounded number of times to encode all possible runs of depth equal to the bound. Then, a valid run corresponds to an assignment of the variables that satisfies the formula. The verification of properties on runs is hence reduced to the Boolean satisfaction problem (SAT) for the logical formula encoding possible runs. BMC was initially introduced for discrete state-transition systems and formulas are expressed in plain propositional logic. BMC has also been studied for timed automata (see Sect. 5), generally based on formulations that use Boolean constraint formulas over linear arithmetic constraints, i.e., Boolean combinations of propositional variables and linear relations over real variables that can be fed to an SMT solver.

In this paper we study an alternative approach to bounded reachability analysis of timed automata, based on Mixed Integer Linear Programming (MILP). We propose several formulations that aim to increasingly improve performance through reducing the search space and we compare these formulations based on two benchmark examples. Since the formulation is also expressible as a Boolean constraint problem over linear arithmetic constraints, we are able to compare the performance of the MILP-based method with one based on SMT. In this first study we have limited the scope to the verification of simple reachability properties; the method can nevertheless be extended to bounded model-checking for more complex temporal properties.

The paper is structured as follows: Sect. 2 provides the definitions for the version of timed automata used in the paper and introduces MILP. Section 3 discusses different formulations of the bounded reachability as a MILP problem. Section 4 discusses and compares experimental results for the different variants and solvers. Sections 5 and 6 discuss the related work before concluding.

2 Preliminaries

2.1 Timed Automata

We rely on a standard definition of timed automata [1]. A timed automaton is a state-transition graph in which transitions may be guarded with conditions on *clock variables*, used to measure the progress of time. Clocks may be reset when a transition fires and they advance at the same rate.

Let X be a finite set of clock identifiers. A *valuation* is a function $\mathbf{v} : X \to \mathbb{R}$ assigning a real value to each clock. A *clock predicate* ζ over X is a logical conjunction of conditions of the form $x \bowtie \mathbf{c}$ where $x \in X$, $\mathbf{c} \in \mathbb{Z}$ (or $\mathbf{c} \in \mathbb{R}$ when the integrality hypothesis is not needed) and $\bowtie$ is one of $<, \leq, >, \geq$, or $=$. Our notation will not distinguish between the predicate and the set of valuations that satisfy it; thus, $\mathbf{v} \in \zeta$ denotes that the valuation $\mathbf{v}$ satisfies the predicate ζ. Let $Cond(X)$ be the set of clock predicates over X.

A *timed automaton* is a tuple $A = (L, l_{init}, X, Inv, Ch, T)$ where L is a finite set of identifiers (the *locations*), $l_{init} \in L$ is the initial location, X is a finite set of *clocks*, $Inv : L \to Cond(X)$ is a function associating an invariant to to each location, Ch is a set of identifiers (the synchronization channels), and T is a

set of tuples of the form $t = (src, dst, syn, grd, rst)$ (the *transitions*) such that: $src, dst \in L$, $syn \in \{\epsilon\} \cup \{?, !\} \times Ch$, $grd \in Cond(X)$, $rst \subseteq X$. The components of t designate the source/destination location, synchronization action (ϵ for no synchronization), the guard condition and respectively a set of clocks that are reset to zero. When several automata are involved, we will use the superscripts to refer to the components of a particular automaton B, e.g., L^B, X^B; for the components of transition tuples, we will use projection functions having the same name as the respective component in the definition above (e.g., $src(t), dst(t)$).

The semantics of a timed automaton is given by its transition system, i.e. a graph in which vertices are configurations and edges represent transitions. A configuration is a pair $(l, \mathbf{v})$ where l is a location and $\mathbf{v}$ is a clock valuation such that $\mathbf{v} \in Inv(l)$. There are two kinds of transitions: elapsing of a duration $\delta \in \mathbb{R}$, denoted $(l, \mathbf{v}) \xrightarrow{\delta} (l, \mathbf{v} + \delta)$ (where $\mathbf{v} + \delta$ is the valuation such that $(\mathbf{v} + \delta)(x) = \mathbf{v}(x) + \delta$) and discrete transitions, denoted $(l, \mathbf{v}) \xrightarrow{t} (l', \mathbf{v}')$ where $t \in T$. Time elapsing is conditioned by $\mathbf{v} + \delta \in Inv(l)$. The discrete transition t is conditioned by $l = src(t)$, $l' = dst(t)$, $\mathbf{v} \in grd(t)$ and $\mathbf{v}'(x) = 0$ for all $x \in rst(t)$ and $\mathbf{v}'(x) = \mathbf{v}(x)$ for all $x \in X \setminus rst(t)$. A path in the transition system is called a *run*. A run is in *canonical form* if it starts and ends with a time elapsing transition (possibly of duration zero) and the sequence of transitions composing it strictly alternates time elapsing transitions and discrete transitions. It is easy to see that any run can be transformed into an equivalent cannonical run by summing up the delay of successive time transitions and by inserting zero-delay transitions where needed.

Given a set of timed automata $A_1, ..., A_n$ with pairwise disjoint sets of locations and clocks, the system of timed automata $\mathcal{A} = A_1 \parallel \ldots \parallel A_n$ is defined by its transition system as follows. The configurations are pairs of the form $((l_1, ..., l_n), \mathbf{v}_1 \sqcup ... \sqcup \mathbf{v}_n)$, where $(l_1, ..., l_n) \in L^{A_1} \times ... \times L^{A_n}$ and $\sqcup$ is the union operator for functions with disjoint domains. Time elapsing transitions $((l_1, ..., l_n), \mathbf{v}_1 \sqcup ... \sqcup \mathbf{v}_n) \xrightarrow{\delta} ((l_1, ..., l_n), \mathbf{v}'_1 \sqcup ... \sqcup \mathbf{v}'_n)$ are possible iff $\forall k, (l_k, \mathbf{v}_k) \xrightarrow{\delta} (l_k, \mathbf{v}'_k)$. Discrete transitions without synchronization $((l_1, ..., l_n), \mathbf{v}_1 \sqcup ... \sqcup \mathbf{v}_n) \xrightarrow{\epsilon} ((l'_1, ..., l'_n), \mathbf{v}'_1 \sqcup ... \sqcup \mathbf{v}'_n)$ are possible iff $\exists k, (l_k, \mathbf{v}_k) \xrightarrow{\epsilon} (l'_k, \mathbf{v}'_k)$ and $\forall j \neq k, l_j = l'_j$ and $\mathbf{v}_j = \mathbf{v}'_j$. Discrete transitions with synchronization are possible only in pairs of an output (!) and an input (?): $((l_1, ..., l_n), \mathbf{v}_1 \sqcup ... \sqcup \mathbf{v}_n) \xrightarrow{\mathbf{c}} ((l'_1, ..., l'_n), \mathbf{v}'_1 \sqcup ... \sqcup \mathbf{v}'_n)$ iff $\exists \mathbf{c} \in Ch^{A_1} \cup ... \cup Ch^{A_n}, k, l$ such that $(l_k, \mathbf{v}_k) \xrightarrow{!\mathbf{c}} (l'_k, \mathbf{v}'_k), (l_l, \mathbf{v}_l) \xrightarrow{?\mathbf{c}} (l'_l, \mathbf{v}'_l)$ and $\forall j \notin \{k, l\}, l_j = l'_j$ and $\mathbf{v}_j = \mathbf{v}'_j$. This version of non-associative n-ary composition is commonly used in practice, for example in the UPPAAL tool [3].

The reachability problem for timed automata is known to be decidable [1]. The decision procedure relies on the integrality of constants used in clock predicates. Our bounded reachability method, as well as others proposed in the literature, can relax this hypothesis and work with real constants (e.g., represented as floating point numbers). On the other hand, since MILP problems only admit non-strict linear constraints (see next paragraph), we forbid strict comparisons

in clock predicates. One can replace strict comparisons used in the automata with non-strict ones by fixing a minimum gap.

2.2 MILP

A Linear Programming problem is a mathematical optimization problem in which constraints are linear inequalities and the objective function is also a linear. A Mixed-Integer Linear Programming (MILP) problem is an LP problem in which some of the variables are constrained to be integers [23]. Like SAT, MILP is NP-complete, but many solvers are capable of solving very large problems arising in practice and their performance has vastly improved during the past decades.

Binaries (i.e., integer variables with value 0/1) can be used to represent Booleans and MILP can encode arbitrary Boolean constraints through inequalities, sometimes more compactly than using the standard logical operators.

In addition to inequalities, some solvers may accept a number of additional constraint types, such as *indicator constraints* [16] which have the form $b \rightarrow C$ where b is a binary variable and C is a linear inequality that has to be satisfied by the solution only if b has the value 1. This is the only form of non-linear constraint that we will use in our formulation of the reachability problem.

3 Formulating Bounded Reachability in MILP

Let $\mathcal{A} = A_1 \parallel \ldots \parallel A_n$ be a system of timed automata. We discuss here the way in which reachable states and transitions of the system are encoded as variables and constraints of a MILP problem. Several options are available for the encoding, one of the goals of this section being to define the variants so that their performance can be compared in the experiments section.

Let us remind first that the formulation concerns the states of the system that can be reached through a sequence of transitions of bounded length. To simplify the definitions, we consider first that there is a total order between the states and between the transitions, although this constraint will be relaxed later on.

3.1 Encoding of State

The state of the automaton A_k at step i is characterized by the location in which it resides and the values of its clocks. To encode these, we use:

- a set of binary variables, one for each location of A_k:

$$VL_i^{A_k} = \{l_i | l \in L^{A_k}, 0 \leq i \leq \mathbf{B}\}$$

- a set of continuous variables, one for each clock of A_k, which will designate the last time (with respect to a time reference frame) when the clock was reset:

$$VX_i^{A_k} = \{reset_i^x | x \in X^{A_k}, 0 \leq i \leq \mathbf{B}\}$$

Since A_k can only be in one state at a time, the following constraint holds:

$$\sum_{l \in VL_i^{A_k}} l = 1 \tag{1}$$

To encode the initial state of each automaton, the following constraints have to hold:

$$\forall l \in L^{A_k} : \quad l_0 = 1 \quad \text{iff} \quad l = l_{init}^{A_k} \tag{2}$$

$$\forall x \in X^{A_k} : \quad reset_0^x = 0 \tag{3}$$

The global state of the system at step i also includes the time since the beginning of the run: now_i (with the constraint $now_0 = 0$). For the moment we consider the case where the transitions of the system are totally ordered in a sequence, hence we can use a global time reference frame. This will no longer be the case when the total order constraint is relaxed, later on.

The state of each automaton has to observe the invariant of its current location. Since each location invariant is a conjunction of atomic clock conditions, each of these can be treated as a separate MILP constraint. By notation abuse, we will write $c \in Inv(l)$ when c is an atomic clock condition part of the conjunction $Inv(l)$. At step i, an atomic condition $x \bowtie \alpha$ is equivalent to the linear expression $now_i - reset_i^x \bowtie \alpha$ and an atomic condition $x - y \bowtie \alpha$ is equivalent to the linear expression $reset_i^y - reset_i^x \bowtie \alpha$. Let LE_i^c denote the linear expression corresponding to condition c at step i. Then, the following constraints have to hold:

$$\forall l \in L^{A_k}, \quad \forall c \in Inv(l) : \quad l_i \rightarrow LE_i^c \tag{4}$$

The purpose of the model is to verify reachability of certain states. For the experiments, we specified the searched state as a conjunction of conditions on automata locations and clocks values at step $\mathbf{B}$, for which the encoding is straightforward.

3.2 Encoding of Transitions

To allow for an efficient formulation of the possible runs of the system, our MILP model allows, by construction, only for canonical runs (in which discrete steps and time elapsing steps strictly alternate). Thus, a step i is in our case formed of a time elapsing step (possibly of delay equal to zero) followed by a discrete step. Thus, when we refer to a sequence of length $\mathbf{B}$, this is actually a sequence of $2\mathbf{B} + 1$ steps: $\mathbf{B}$ pairs formed of a time step and a discrete step, plus a final time step in order to allow for time to go on after the last discrete step. Steps are numbered from 0 to $\mathbf{B}$.

The time elapsing steps are not explicitly encoded, which further simplifies the model. Instead, we simply add the condition that time has to progress in the right direction:

$$\forall i : \quad now_i \leq now_{i+1} \tag{5}$$

With this in mind, now_i designates the current time before the pair (time delay, discrete transition) of rank i. Thus the discrete transition i takes place at time now_{i+1}.

A first consequence is that the constraint (4) given above models the satisfaction of location invariants *before* the time step i but not *after*. In order to ensure the satisfaction of the invariant *after* the step i (and hence, between the two, since invariants are convex), we need an additional constraint. Let LEA_i^c denote the linear expression corresponding to condition c *after* step i. It is easy to see that LEA_i^c can be built similarly to LE_i^c, based on now_{i+1} (time *after* the delay step i) and on the values of $reset_i^x$ (reset dates *before* the discrete step i).

$$\forall l \in L^{A_k}, \quad \forall c \in Inv(l) : \quad l_i \rightarrow LEA_i^c \tag{6}$$

For each discrete transition we will use an auxiliary binary variable that models the fact that the transition is triggered at step i. While this is not usually done in other formulations used for BMC, we find that this makes it easier to express the constraints and to reconstruct the sequence of transitions when the solver finds a feasible solution. Thus:

$$VT_i^{A_k} = \{t_i | t \in T^{A_k}, 0 \leq i < \mathbf{B}\}$$

Except for synchronization which is discussed in the next section, the other necessary conditions for a discrete transition are given below. To simplify the formulas, the components of a transition t (i.e., $src(t)$, $dst(t)$,...) will also be denoted by $src(vt), dst(vt), ...$, for any $vt \in VT_i^{A_k}$ that corresponds to t.

$$\forall t \in VT_i^{A_k} : \quad t \rightarrow src(t_i) \wedge dst(t_{i+1}) \tag{7}$$

$$\forall t \in VT_i^{A_k}, \quad \forall c \in grd(t) : \quad t \rightarrow LEA_i^c \tag{8}$$

$$\forall t \in VT_i^{A_k}, \quad \forall x \in VX_i^{A_k} \text{ s.t. } x \in rst(t) : \quad t \rightarrow (reset_{i+1}^x = now_{i+1}) \tag{9}$$

$$\forall t \in VT_i^{A_k}, \quad \forall x \in VX_i^{A_k} \text{ s.t. } x \notin rst(t) : \quad t \rightarrow (reset_{i+1}^x = reset_i^x) \tag{10}$$

Instead of a discrete transition, an automaton A_k may perform a special "skip" transition at step any i, without changing either the state or the values of *reset* variables. In the following section we will discuss some additional conditions that ensure that *skip* steps of individual automata are only used under certain conditions, so that the global system runs continue to have the canonical form. To represent the *skip* transitions, a binary variable $skip_i^{A_k}$ is introduced for each i and A_k, along with these constraints:

$$\forall i, \forall x \in X^{A_k} : \quad skip_i^{A_k} \rightarrow (reset_{i+1}^x = reset_i) \tag{11}$$

$$\forall i, \forall l \in L^{A_k} : \quad skip_i^{A_k} \rightarrow (l_{i+1} = l_i) \tag{12}$$

Skip transitions are also useful for encoding the fact that a bounded sequence of length $\mathbf{B}$ may be followed by one final time step: we extend the length of the sequence by one and we require that the last discrete step (numbered $\mathbf{B}$) be a *skip*.

3.3 Relaxing the Order of Transitions and Handling Synchronization

From this point on, several variants of the model will be considered. They all share the variables and constraints described previously and differ essentially in the way in which transitions of individual automata are ordered within the global run and in how synchronization between automata is handled.

A first variant (denoted **SS** for *sequential steps*) is to consider that transitions are ordered sequentially. At each step, only one automaton may fire a discrete transition. In order to account for synchronization, the constraints ensure that an *input* on some channel can only be executed by an automaton immediately after an *output* on the same channel was executed by a different automaton (i.e., in the next step and so that now does not change between the two). To preserve the canonical form of runs, a constraint ensures that, once a *skip* transition appears, all subsequent transitions are *skips*. Let *inputs*/*outputs* designate the set of all transitions that specify an input (resp. output) synchronization and $conjugated(t)$ be a function that gives the set of all transitions t' which specify an output synchronization with the same channel name as t. We do not formally define these, but it is relatively easy to see how they are syntactically derived from the definition of a system. The formulation is as follows:

$$\forall i : \sum_{k} (skip_i^{A_k} + \sum_{t \in VT_i^{A_k}} t) = 1 \tag{13}$$

$$\forall i, \forall t \in outputs : t_i \rightarrow \sum_{t' \in conjugated(t)} t'_{i+1} = 1 \tag{14}$$

$$\forall i > 0, \forall t \in inputs : t_i \rightarrow \sum_{t' \in conjugated(t)} t'_{i-1} = 1 \tag{15}$$

$$\forall i . 0 < i < \mathbf{B}, \forall t \in inputs : t_i \rightarrow (now_i = now_{i+1}) \tag{16}$$

By experimenting with this formulation, one rapidly concludes it is inefficient, mainly for two reasons. Firstly, since only one automaton is allowed to step at a time, one has to choose a relatively large bound **B**, which in itself penalizes performance. Secondly, if the model is used for establishing the unreachability of some configuration (as it is the case when one tries to verify a safety property), a positive result is achieved when the model is *infeasible* (the term used by MILP solvers, meaning *unsatisfiable*). However, the difficulty of proving infeasibility is generally correlated with the size of the Infeasible Irreducible System (IIS, equivalent of the UNSAT-core in SAT/SMT). Experiments show that with the **SS** formulation, the IIS is generally the entire model (i.e., no constraint can be removed without breaking infeasibility) – and therefore establishing infeasibility is hard.

This finding led us to seek more efficient formulations. A first variant (**MS1** for multi-step with unique time basis) is to allow for multiple automata to trigger discrete transitions within the same step. This also allows a simpler handling of

input/output synchronization, which can now be performed within the same step. The formulation is as follows:

$$\forall i, \forall k : skip_i^{A_k} + \sum_{t \in VT_i^{A_k}} t = 1 \tag{17}$$

$$\forall i, \forall t \in outputs : t_i \rightarrow \sum_{t' \in conjugated(t)} t'_i = 1 \tag{18}$$

$$\forall i, \forall t \in inputs : t_i \rightarrow \sum_{t' \in conjugated(t)} t'_i = 1 \tag{19}$$

This formulation is more efficient as it allows to use a lower value for the bound **B**, since several automata can trigger during a step. However, the use of a unique time basis for all automata (the now_i variables) introduces dependencies between their behaviors. As a consequence, even when a safety property could in principle be proved locally on one or a small subset of the system's automata, the actual IIS is still usually the entire model, and therefore infeasibility remains hard to prove.

A solution to this problem can be to de-correlate time progress in the different automata forming a system. As long as an automaton progresses without synchronizing with others, it can use its own value of *now* which can be different from the others', in a way similar to what was proposed in [20]. Only when two automata synchronize, they must agree on their respective value of *now*. To encode this we replace each now_i variable by a set of variables $now_i^{A_k}$, and the constraints (4), (5), (6), (8) and (9) are rewritten to refer to the local *now* of the concerned automaton. Of course, this implies that an automaton can only read/reset its own clocks.

In this model, there are several ways to achieve input/output synchronization. A first variant (denoted **MSm** for multi-step with multiple time bases) will rely on the same constraints as **MS1**, i.e., (17), (18) and (19), while adding two more:

$$\forall i, j, k, \forall t \in VT_i^{A_j}, \forall t' \in VT_i^{A_k} \text{ s.t. } t' \in conjugated(t) :$$
$$t \wedge t' \rightarrow (now_{i+1}^{A_j} = now_{i+1}^{A_k}) \tag{20}$$

meaning that local *nows* agree in case of synchronization, and

$$\forall j, k : now_{\mathbf{B}}^{A_j} = now_{\mathbf{B}}^{A_k}) \tag{21}$$

meaning that local *nows* agree at the end of the sequence.

To ensure that we obtain a canonical run with **MSm**, we can add a constraint enforcing that, if all automata perform a *skip* at step i, they will continue doing the same for all steps $j > i$. However, even with this constraint, an individual automaton may still perform a *skip* at step i and some discrete transition at a later step. As this seems to be a source of combinatorial explosion, we have sought to remove it, by no longer relying on the fact that inputs/outputs have to

take place in the same step (constraints (18) and (19)). This opens up interesting possibilities:

- The steps of the different automata forming the system are completely decorrelated. The run is no longer a unique sequence of (multi-)steps but a set of sequences, one for each automaton.
- The sequences can be of different length. One can imagine fixing the depth bound **B** differently for each automaton (e.g., depending on its own complexity).
- Each individual sequence can be constrained to be in canonical form, i.e., no more spurious *skip* transitions (except at the end of each run).

However, this also raises new challenges, as the global coherence of the model still has to be ensured. A solution is to use a matrix of auxiliary binary variables to represent the fact that step i of an automaton A_n synchronizes with step j of A_m. Constraints were added to ensure that synchronizations take place at the same time (similar to condition (20)), and that message overtaking does not occur. Details are omitted here, they can be found in the code of the prototype. Henceforth, this variant of the formulation will be denoted **ISs** (independent-steps with synchronization).

3.4 MILP Objective

The difference between an SMT-based bounded model checker/reachability analyzer and one based on MILP is that the latter may integrate an optimization objective. The objective has the form of a linear expression on model variables (depending on the solver, other forms of expressions, such as quadratic forms, may also be used). The objective proves to be useful for selecting a system run out of the set of feasible ones based on minimizing/maximizing various criteria. For example, for model debugging it is often convenient to obtain the shortest run that leads to the searched state, i.e. the run that contains the smallest number of (non-*skip*) discrete transitions. This can be obtained by minimizing the objective:

$$obj = \sum_{\substack{i,k \\ t \in VT_i^{A_k}}} t$$

Other examples of uses for the objective function include searching for runs that optimize the time of residence in certain locations. It is also easy to extend the model to handle weighted timed automata [6], which add costs on states/locations, so as to search for runs that optimize the total cost.

4 Experimental Results

The method described in the previous section was implemented in a prototype[1] written in Python and using Gurobi [16] as back-end MILP solver. In order to

[1] https://www.irit.fr/~Iulian.Ober/brat.

allow comparisons, the prototype can also encode the reachability problem as an SMT problem over linear arithmetic constraints, and use MathSAT [21] as backend (via the *pysmt* API). Both formulations use exactly the same constraints, therefore providing an interesting basis for comparison. The automata are specified programmatically directly in Python; however, the format is relatively close to the textual format of UPPAAL, to the point that we could adapt some benchmark generation scripts[2] to generate models for our experiments. Experiments were performed on a Linux machine with 8 Intel Core 2.4 GHz CPUs and 16 GB of memory. Note that the version based on Gurobi exploits the platform parallelism, whereas the one based on MathSAT only uses one of the processors.

4.1 Examples Used

Several examples have been built in order to exercise the prototype. We will concentrate in the following on two of them: the now-classical Train-Gate-Controller (TGC) example [1] and the CSMA/CD (Carrier Sense, Multiple-Access with Collision Detection) protocol, based on the model included in the UPPAAL benchmarks [22]. The CSMA/CD protocol allows to assign a broadcast network channel to one of several competing transmitters. A detailed description is given in [27]; let us note that the model is parametric in the number of transmitters.

4.2 Results for Feasible Models (Reachable States)

In the first experiment reported here, we search for a state for which we know that it may be reached at a certain depth. In the CSMA/CD example, for a model with N transmitters, an interesting candidate is the state $bus\_collision N$ of the automaton corresponding to the bus, since we know that it may be reached at a minimum depth of $N+1$. For each value of N two tests are performed, one with a depth bound $\mathbf{B} = N+1$ and another one with a larger bound. For each combination of N and depth, the different variants of formulation presented in Sect. 3.3 have been tried, both using the MILP encoding (Gurobi) and the SMT encoding (MathSAT). The quantitative results are listed in the Fig. 1; the green background designates the solver which produced faster results for a particular configuration. In all experiments the time limit was set to 1000 s.

On this experiment the speed of the two solvers is generally comparable, with a slight advantage for the MILP solver for lower values of $\mathbf{B}$ and for the SMT solver for larger ones. It is worth noting however that the MILP encoding provides results that are qualitatively more interesting: we have set the objective of finding traces with a minimum number of (non-skip) discrete transitions. In the case where the bound is strictly larger than $N+1$, the runs provided by the SMT solver contain many more transitions than necessary for reaching the goal state, while the runs provided by the MILP solver have exactly $N+1$ transitions. Thus, when reachability analysis is used for model understanding and debugging purposes, the MILP solution provides more interesting results.

[2] https://www.it.uu.se/research/group/darts/uppaal/benchmarks.

N / B	MS1		MSm		ISs	
	SMT	MILP	SMT	MILP	SMT	MILP
10 / 11	0.41	0.26	0.7	0.29	0.82	0.24
10 / 22	3.61	0.92	4.15	5.27	4.22	2.02
15 / 16	1.41	1.76	3.44	1.85	2.14	0.85
15 / 32	18.13	19.45	40.61	14.79	33.98	22.82
20 / 21	3.96	4.74	12.26	2.28	3.91	2.09
20 / 42	34.85	12.29	132.14	55.19	47.02	67.66
30 / 31	18.76	15.26	54.84	9.14	19.89	17.24
30 / 50	135.17	87.64	432.29	929.11	374.55	524.9
40 / 41	69.93	89.22	131.15	27.16	58.28	44.06
40 / 60	379.66	166.32	918.53	--	416.59	917.95

Fig. 1. Experiment 1 (CSMA/CD) – times in s.

As the numbers in Fig. 1 indicate, the CSMA/CD example does not benefit from the partially ordered runs afforded by the **MSm** and **ISs** variants. This is caused by the centralized nature of the example, as all the transitions of the transmitting stations synchronize with a transition of the bus, whose behavior is essentially sequential.

We proceed with a second experiment which exhibits an increased degree of parallelism. Based on the TGC example [1], we build a system composed of **N** Train-Gate-Controller triplets. In order to demonstrate the interest of having multiple time bases (the case of the **MSm** and **ISs** variants), the waiting delay before the Controller sends the signal to raise the Gate is set to a different value in each triplet. The reachable configuration that will be searched is one in which every Gate is in state *raising*, after a train has passed.

N	MS1		MSm		ISs	
	SMT	MILP	SMT	MILP	SMT	MILP
2	0.08	60.51	0.05	0.07	0.09	0.03
3	0.22	--	0.11	0.11	0.17	0.05
5	2.48	--	0.33	0.18	0.51	0.09
10	102.01	--	2.44	0.41	2.57	0.22
20	--	--	21.37	0.63	15.22	0.61
50	--	--	154.59	1.83	157.71	2.67
100	--	--	682.53	20.65	668.45	14.4

Fig. 2. Experiment 2 (TGC) – times in s.

The search times for different values of **N** are given in Fig. 2. Note that for the **MS1** variant, **N**+5 steps are necessary to reach the search state, whereas for the variants that use a separate time basis for each automaton (**MSm** and **ISs**) the same state can be reached in a constant number of steps (5). This explains the wide difference in performance between the three variants. It is also to be noted that the relative performance of the solvers is widely different depending

on the variant: the SMT solver is orders of magnitude faster on **MS1** while the MILP solver is up to 50 times faster on **MSm** and **ISs**.

4.3 Results for Infeasible Models (Unreachable States)

When reachability analysis is used for verifying a safety property (i.e., that some "bad state" is never reached), the MILP model (respectively the SMT problem) will be infeasible (unsatisfiable) when the property is verified. Experiments show that the performance of the solvers is not uniform whether the purpose is finding scenarios in a feasible model or proving that the model is infeasible. This section is dedicated to experiments for the latter case.

For the TGC example, a safety property is that the Gate cannot be in a state other than *closed* when the Train passes the Gate. We try to prove this property holds up to a "reasonable" bound for depth. The choice of the bound is somewhat arbitrary, but is informed by the results of experiment 2, which show that a full cycle of gate lowering – train passing – gate raising can be achieved in **N**+5 steps for **MS1** and in 5 steps for **MSm** and **ISs**. The bound is thus chosen to be 2***N** for **MS1** and respectively 10 for **MSm** and **ISs**.

The computation times for deciding infeasibility are given in Fig. 3. Notice that the SMT solver performs significantly better on this task than the MILP solver. The **MSm** and **ISs** formulations also perform much better than **MS1** for large models, **ISs** being the only formulation for which the MILP solver can handle larger systems in a reasonable time.

	MS1		MSm		ISs	
N	SMT	MILP	SMT	MILP	SMT	MILP
2	0.01	1.55	0.04	31.3	0.25	0.36
3	0.03	13.74	0.06	80.16	0.34	0.56
5	0.07	--	0.09	993.15	0.56	1.17
10	0.41	--	0.21	--	1.13	2.44
20	3.27	--	0.45	--	2.4	7.41
50	68.92	--	1.54	--	6.46	26.03
99	--	--	5.81	--	16.09	70.66

Fig. 3. Experiment 3 (TGC with unreachable end state) – times in s.

5 Related Work

Applying bounded model checking [5] to timed automata has been the subject of many studies in the past, beginning with [2,24,25]. The problem is reduced to satisfiability of formulas in a decidable first order logic (e.g., propositional logic with linear arithmetic constraints or difference logic). Most recent works rely on SMT solvers, which have made significant progress in the past years and are able to handle large specifications. To our knowledge, Mixed Integer Linear

Programming has not yet been explored for formulating bounded model-checking problems, except in the realm of linear hybrid automata [13]. The authors of [13] concentrate on the integration of a DPLL-based SAT solver with a linear programming routine in order to benefit from the capacity of the LP routine to solve large conjunctive systems of linear inequalities over the reals. Although the method proposed by [13] could be adapted to fit our needs, we have chosen to rely on an off-the-shelf MILP solver and we concentrated on making the formulation as efficient as possible and on comparing the MILP solution with one based on SMT.

The idea of reducing the length of runs (and hence the size of the search space) by allowing several automata to make discrete transitions in the same (multi-)step has been explored in [20]. It follows up on work on partial order reductions for timed automata [4,19]. We take the multi-step idea two steps forward, first by allowing the clocks of different automata to be de-synchronized in the same multi-step, and then by allowing synchronizing transitions to take place in different steps, which allows to separate the representations of the runs of different TAs and use different bounds on the run length for each automaton. A similar approach was presented in [9] in the context of linear hybrid automata.

6 Conclusions

The results presented in this paper show that there is a place for MILP-based bounded reachability analysis in the spectrum of analysis methods used for timed systems. While the SMT-based method outperforms it when there are no satisfying runs, which makes SMT a better candidate for approaching model-checking problems, the MILP-based method proves to be relatively fast for finding satisfying runs when they exist. Moreover, the method allows to search for runs that optimize certain criteria. Since different criteria may be encoded in the optimization objective, such as run length or time of residence in certain states, our approach provides a convenient method for exploring behavior, model understanding and debugging.

The paper also discusses certain techniques for reducing the size of the search space based on allowing as much as possible independent progress of the different automata forming the system. Several different formulations of the reachability problem are presented and we provide experimental data allowing to compare their relative performance. One formulation (**ISs**) is particularly interesting, both from the point of view of raw performance, and because it separates the representations of the runs of different automata, which allows to set different bounds on their respective length. We think that this should allow to handle more efficiently large systems that mix components of varying complexity.

The prototype implemented for this study handles only a minimalist communicating timed automata model. Future work is needed for enriching the model, e.g., with local/shared data, data communication over synchronization, shared clocks, location and transition weights [6], etc. Although we do not aim for a full-fledged bounded model checker, it would be interesting to provide counterexample generation for more complex temporal logic properties.

This paper is dedicated to Susanne Graf on the occasion of her anniversary event, as a mark of my admiration and respect for her scientific achievements and for her human qualities. It is an honor and an inspiration to have her as colleague and friend.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
2. Audemard, G., Cimatti, A., Kornilowicz, A., Sebastiani, R.: Bounded model checking for timed systems. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 243–259. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36135-9_16
3. Behrmann, G., David, A., Larsen, K.G., Pettersson, P., Yi, W.: Developing UPPAAL over 15 years. Softw. Pract. Exper. **41**(2), 133–142 (2011)
4. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial order reductions for timed systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 485–500. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055643
5. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
6. Bouyer, P.: Weighted timed automata: model-checking and games. Electron. Notes Theor. Comput. Sci. **158**, 3–17 (2006). Proceedings of the 22nd Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXII)
7. Bozga, M., Graf, S., Mounier, L., Ober, I., Roux, J.-L., Vincent, D.: Timed extensions for SDL. In: Reed, R., Reed, J. (eds.) SDL 2001. LNCS, vol. 2078, pp. 223–240. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48213-X_14
8. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF toolset. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 237–267. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_8
9. Bu, L., Cimatti, A., Li, X., Mover, S., Tonetta, S.: Model checking of hybrid systems using shallow synchronization. In: Hatcliff, J., Zucca, E. (eds.) FMOODS/FORTE -2010. LNCS, vol. 6117, pp. 155–169. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13464-7_13
10. Damm, W., Olderog, E.-R. (eds.): FTRTFT 2002. LNCS, vol. 2469. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45739-9
11. David, A., Möller, M.O., Yi, W.: Formal verification of UML statecharts with real-time extensions. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 218–232. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45923-5_15
12. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52148-8_17
13. Fränzle, M., Herde, C.: Efficient proof engines for bounded model checking of hybrid systems. Electron. Notes Theor. Comput. Sci. **133**, 119–137 (2005)
14. Graf, S.: Expression of time and duration constraints in SDL. In: Sherratt, E. (ed.) SAM 2002. LNCS, vol. 2599, pp. 38–52. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36573-7_3

15. Graf, S.: OMEGA: correct development of real time and embedded systems. Softw. Syst. Model. **7**(2), 127–130 (2008)
16. Gurobi Optimization Inc., Reference manual v. 7.5. https://www.gurobi.com/documentation/7.5/refman.pdf. Accessed on 8 June 2018
17. Knapp, A., Merz, S., Rauh, C.: Model checking - timed UML state machines and collaborations. In: Damm and Olderog [10], pp. 395–416
18. Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Clock difference diagrams. Nord. J. Comput. **6**, 271–298 (1999)
19. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. Theor. Comput. Sci. **345**(1), 27–59 (2005)
20. Malinowski, J., Niebert, P.: SAT based bounded model checking with partial order semantics for timed automata. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 405–419. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_34
21. MathSAT 5. http://mathsat.fbk.eu. Accessed 8 June 2018
22. Möller, O.: CSMA/CD protocol specification (UPPAAL benchmark). https://www.it.uu.se/research/group/darts/uppaal/benchmarks/#CSMA. Accessed 8 June 2018
23. Nemhauser, G.L., Wolsey, L.A.: Integer and Combinatorial Optimization. Wiley-Interscience, New York (1988)
24. Niebert, P., Mahfoudh, M., Asarin, E., Bozga, M., Maler, O., Jain, N.: Verification of timed automata via satisfiability checking. In: Damm and Olderog [10], pp. 225–244
25. M. Sorea. Bounded model checking for timed automata. Electron. Notes Theor. Comput. Sci. **68**(5), 116–134 (2003). MTCS 2002, Models for Time-Critical Systems (CONCUR 2002 Satellite Workshop)
26. Wang, F.: Efficient verification of timed automata with BDD-like data structures. STTT **6**(1), 77–97 (2004)
27. Yovine, S.: KRONOS: A verification tool for real-time systems. STTT **1**(1–2), 123–133 (1997)