Scaling BDD-based Timed Verification with Simulation Reduction

Truong Khanh Nguyen[1],Tian Huat Tan[2], Jun Sun[2], Jiaying Li[2], Yang Liu[3], Manman Chen[2], Jin Song Dong[4]

Autodesk[1] Singapore University of Technology and Design[2]
Nanyang Technological University[3] National University of Singapore[4]
truong.khanh.nguyen@autodesk.com
{tianhuat_tan,sunjun,manman_chen}@sutd.edu.sg,
jiaying_li@mymail.sutd.edu.sg,
yangliu@ntu.edu.sg,dongjs@comp.nus.edu.sg

Abstract. Digitization is a technique that has been widely used in real-time model checking. With the assumption of digital clocks, symbolic model checking techniques (like those based on BDDs) can be applied for real-time systems. The problem of model checking real-time systems based on digitization is that the number of tick transitions increases rapidly with the increment of clock upper bounds. In this paper, we propose to improve BDD-based verification for real-time systems using simulation reduction. We show that simulation reduction allows us to verify timed automata with large clock upper bounds and to converge faster to the fixpoint. The presented approach is applied to reachability and LTL verification for real-time systems. Finally, we compare our approach with existing tools such as Rabbit, Uppaal, and CTAV and show that our approach outperforms them and achieves a significant speedup.

1 Introduction

Timed automata are an extension of finite automata with clock variables which represent timed constraints [3]. Interesting model checking problems of timed automata, like the verification of the reachability and LTL properties, are shown to be decidable through the construction of region graphs [3]. However, since the size of region graphs grows exponentially with the number of clocks and the maximal clock constants, verification based on region graphs is impractical.

There are two lines of works that are proposed to address this problem. The first line of work is based on *Difference Bound Matrix* (DBM). DBM was proposed to represent a set of clock valuations satisfying a set of convex clock constraints [19] with a zone graph. The resulted zone graph is often much smaller than the region graph, which often results in efficient verification timed automata models [15]. There are several problems with DBM. First, it is difficult to verify LTL properties with non-Zeno assumption. A run is called Zeno if there are infinite actions happening in finite time. Zeno runs are unrealistic and therefore should be excluded during the system verification. However, this process has shown to be fairly non-trivial [41]. Second, DBM cannot represent non-convex zones. Some verification/reduction techniques for timed automata may result in non-convex zones, and novel techniques need to be invented for handling such cases. For instance, with a particular abstraction technique called LU abstraction [7], the resulted zone can be non-convex. In such a case, a convex subset of LU abstraction, called

$Extra^+_{LU}$ extrapolation [7], need to be used. Third, since locations and clock valuations are stored separately in zone graphs, state space explosion is often encountered with models having many processes.

The other line of work is based on digitization [29]. It replaces the continuous passage of time with a passage in discrete steps. The advantage of this approach is that, it helps transforming the problem to model checking a discrete system and techniques such as BDD-based symbolic model checking [16] can be leveraged. There are several advantages of using BDD-based verification compared to DBM-based verification. First, checking non-Zenoness with digitization and BDDs is almost trivial. Furthermore, it has been shown to be outperformed zone-based approach in many existing works (e.g., [15, 5, 9, 43, 12]). Second, we can store both locations and clock valuations together symbolically and does not have limitation with non-convex sets. However, the problem with digitization and BDD-based approach is that it does not scale for large clock constants. Large clock constants would increase significantly the number of tick transitions which denote the passage of one time unit. As a result, a large number of iterations are often necessary to completely explore the state space.

In this work, we propose the usage of *LU simulation* to address the aforementioned problem. In particular, we propose two algorithms, based on *LU simulation*, for model checking reachability and LTL properties respectively. A desired property of LU simulation is that it can be obtained *for free* in timed automata. Our algorithms depend on two clock bounds that are – the maximal lower bound and the maximal upper bound (LU bounds) [7]. By leveraging these clock bounds, we could explore the set of all reachable states from initial states in fewer iterations. Intuitively, this is achieved in two ways. First, during the verification, given a set of reachable states S encoded as BDD, we actively enlarge it by adding states which can be simulated by those in S. Thus, we have more states and it is possible to find all the reachable states with fewer iterations. Second, according to LU simulation relation, states with the clock value greater than the maximal lower bound can simulate all states with larger clock values. Therefore, our method could perform well even if the maximal upper bound is very large.

In short, we make the following technical contributions in this work:

1. We have applied simulation reduction in a BDD efficient way for both reachability and LTL properties. To the best knowledge of the authors, we are the first to apply LU simulation relation in BDD-based approach model checking of timed automata.
2. We have shown the soundness and completeness pf our proposed algorithms. In addition, we further prove that for the algorithm on verifying reachability properties, our approach always requires the same or fewer iterations than classic approaches.
3. We have compared our approaches on verifying reachability and LTL properties with state-of-the-art DBM-based and BDD-based model checkers, e.g., Uppaal [30] and Rabbit [10] on benchmark systems. The results show that our approach achieves a significant speed up and outperforms other tools.

Related Work On the effort of improving reachability analysis of timed automata, this work is related to studies on the abstraction techniques to reduce states number in zone graphs, such as [33, 13, 7, 26]. The idea is to enlarge a DBM without violating the correctness. This work continues the research on using BDDs and BDD-like data structures to improve the verification of real-time systems [15, 5, 8, 9, 43, 12, 44, 38].

This work is related to the research on simulation reduction (e.g., [20, 21]) as well as research on the emptiness checking of Timed Büchi Automata (TBA). Note that LTL verification on timed automata can be converted to the emptiness checking of TBA. In [41], Tripakis discovered that it is non-trivial to check whether a run in a zone graph can induce a non-Zeno run in the original TBA. The proposed remedy is to transform a TBA to an equivalent strongly non-Zeno TBA so that algorithms for emptiness checking of Büchi automata can be used to solve the emptiness problem of TBA. In [42], Tripakis questioned whether coarser extrapolation techniques, specifically inclusion abstraction [18] and LU extrapolation [7], can also be used to check TBA emptiness. In [28], Laarman *et al.* showed that inclusion abstraction only preserves the emptiness of TBA in one direction. In [31], Li showed that LU extrapolation indeed preserves the emptiness of TBA. One result of this work is an improved algorithm of solving the non-emptiness problem based on BDDs.

This work is closely related to [7], [31] and work on using downward closure [21] based on LU simulation relation as an abstraction. While [7] and [31] both apply LU simulation relation to DBMs ($Extra^{+}_{LU}$ extrapolation) for reachability analysis and emptiness checking respectively, we apply the LU simulation relation to BDDs for both reachability and emptiness. There are two advantages of our approach. First, given a convex set of clock valuations, $Extra^{+}_{LU}$ is a subset of LU abstraction. Our approach based on LU abstraction can be more efficient than $Extra^{+}_{LU}$ [26, 21], because a BDD can represent a non-convex set of clock valuations. Second, to handle the non-Zeno condition, [31] relies on the strongly non-Zeno transformation, which requires an additional clock and may result in a zone graph with exponentially more states [25, 24].

Organization The rest of the paper is organized as follows. Section 2 introduces timed automata and the LU simulation relation in timed automata. Section 3 presents our work on the reachability analysis. Then, Section 4 presents our work on the LTL verification. Next, Section 5 shows the experimental results. Section 6 discusses our work. Finally, Section 7 concludes our paper.

2 Preliminaries

2.1 Timed Automata

In this section, we introduce timed automata, arguably one the most popular modeling languages for real-time systems. We denote the finite alphabet by Σ. Let $\mathbb{R}_{\geq 0}$ be the set of non-negative real numbers. Let X be the set of non-negative real variables called clocks. The set $\Phi(X)$ contains all clock constraints δ defined inductively by the grammar : $\delta := x \sim c \mid x - y \sim c \mid \delta \wedge \delta$ where $x, y \in X$, $\sim \in \{<, \leq, =, \geq, >\}$, and $c \in \mathbb{N}$. Given a set of clocks X, a clock valuation $v : X \to \mathbb{R}_{\geq 0}$ is a function which assigns a non-negative real value to each clock in X. We denote $\mathbb{R}_{\geq 0}^{|X|}$ the set of clock valuations over X. A clock valuation v satisfies a clock constraint δ, written as $v \models \delta$, if and only if δ evaluates to true using the clock values given by v. We denote by $\mathbf{0}$ the clock valuation that assigns every clock the value 0. Given a clock valuation v and $d \in \mathbb{R}_{\geq 0}$, the clock valuation $v' = v + d$ is defined as $v'(x) = v(x) + d$ for all clocks x in X. For $R \subseteq X$, let $[R \mapsto 0]v$ denote the clock valuation v' such that $v'(x) = v(x)$ for all $x \in X \setminus R$ and $v'(x) = 0$ for all $x \in R$.

Definition 1. *A timed automaton is a tuple* $A = (\Sigma, X, L, l_0, T, I)$ *where*

– *Σ is the finite alphabet, X is the set of clock variables.*
– *L is the set of locations, $l_0 \in L$ is the initial location.*
– *$T \subseteq L \times \Phi(X) \times \Sigma \times 2^X \times L$ is the set of transitions (l, g, e, R, l') where l and l' are the source and destination locations of this transition respectively, $g \in \Phi(X)$ is a guard, $e \in \Sigma$ is an event name, and $R \subseteq X$ is a set of resetting clocks.*
– *$I : L \to \Phi(X)$ assigns invariants to locations.*

The (continuous) semantics of a timed automaton $A = (\Sigma, X, L, l_0, T, I)$ is a transition system $CS(A) = (S, s_0, \to)$ where $S = L \times \mathbb{R}^{|X|}_{\geq 0}$ is a set of states, $s_0 = (l_0, \mathbf{0})$ is the initial state, and $\to$ is the smallest labeled transition relation satisfying the following:

– Delay transition: $(l, v) \xrightarrow{d} (l, v + d)$ if $\forall\, 0 \leq d' \leq d, v + d' \models I(l)$
– Action transition: $(l, v) \xrightarrow{t} (l', v')$ with $t = (g, e, R)$ if there exists $(l, g, e, R, l') \in T$ such that $v \models g$, $v' = [R \mapsto 0]v$, and $v' \models I(l')$

We write $(l, v) \xrightarrow{d}\xrightarrow{t} (l', v')$ if there exists (l_1, v_1) where $(l, v) \xrightarrow{d} (l_1, v_1)$ and $(l_1, v_1) \xrightarrow{t} (l', v')$. A run of A is a sequence $(l_0, v_0) \xrightarrow{d_0}\xrightarrow{t_0} (l_1, v_1) \xrightarrow{d_1}\xrightarrow{t_1} \cdots$. A state (l_n, v_n) is reachable from (l_0, v_0) if there is a run starting from (l_0, v_0) and ending at (l_n, v_n). The duration of the run is defined as the total delay over this run, $\sum_{i \geq 0} d_i$. A run is called Zeno if there are infinite actions happening in finite time. Given a timed automaton $A = (\Sigma, X, L, l_0, T, I)$ and a location $l \in L$, reachability analysis is to decide whether a given state (l, v) is reachable from the initial state $(l_0, \mathbf{0})$. Next, we define the emptiness checking problem for timed automata. Let $Acc \subseteq L$ be the set of accepting locations. An accepting run of A is a run which visits a state in Acc infinitely often. The language of A over Acc, $\mathcal{L}(A)$, is defined as the set of accepting non-Zeno runs. The emptiness problem is to determine whether $\mathcal{L}(A)$ is empty, i.e., whether there exists an infinite run which is non-Zeno and accepting. We remark that reachability analysis is often used to verify safety problem, whereas algorithms for the emptiness checking problem can often be extended to verify liveness properties like LTL formulae.

In the above semantics, clock values are continuous and events are observed at real time points. Thus, the number of states is infinite and BDDs can not be applied to verify timed automata under this semantics. In the following, we introduce discrete semantics which is based on the assumption that events are observed at integer time points only.

2.2 Discrete Semantics

In discrete semantics, we assume that clock constraints are always closed, i.e., defined by $\delta_c := x \sim_c c \mid x - y \sim_c c \mid \delta_c \wedge \delta_c$ where $x, y \in X$, $\sim_c \in \{\leq, =, \geq\}$, and $c \in \mathbb{N}$. Timed automata with closed constraints are called *closed timed automata* [5, 23].

Given any clock $x \in X$, we write $M(x)$ to denote the maximal constant to which x is compared in any clock constraint of A. Given a clock valuation v, $v \oplus d$ denotes the clock valuation where $(v \oplus d)(x) = min(v(x) + d, M(x) + 1)$. Intuitively, for each clock x, once the clock value is greater than its maximal constant $M(x)$, its exact value is no longer important, but the fact $v(x) > M(x)$ matters.

The discrete semantics of a closed timed automaton $A = (\Sigma, X, L, l_0, T, I)$ is a transition system $DS(A) = (S, s_0, \to)$ where $S = L \times \mathbb{N}^{|X|}$ is a set of states, $s_0 = (l_0, \mathbf{0})$ is the initial state, and $\to$ is the smallest labeled transition relation satisfying the following condition:

– Tick transition: $(l, v) \xrightarrow{tick} (l, v \oplus 1)$ if $v \models I(l)$ and $v \oplus 1 \models I(l)$
– Action transition: $(l, v) \xrightarrow{t} (l', v')$ with $t = (g, e, R)$ if there exists $(l, g, e, R, l') \in T$ such that $v \models g$, $v' = [R \mapsto 0]v$, and $v' \models I(l')$

It was shown that the discrete semantics preserves untimed properties of closed timed automata [5, 23]. Thus, $DS(A)$ can be used in place of $CS(A)$ in the verification of untimed properties like untimed reachability analysis and untimed LTL verification. It follows that BDDs can be used to encode and verify the closed timed automata based on the discrete semantics. In this work, we adopt the approach presented in [35, 9] to encode $DS(A)$ in BDD. Given a timed automaton $A = (\Sigma, X, L, l_0, T, I)$, we denote $Init$, $Tick$, and $Trans$ the BDD encodings of the initial states, tick transitions, and action transitions of $DS(A)$ respectively. Note that the encoding of the transition relation of $DS(A)$ is the disjunction of $Tick$ and $Trans$. The tick transitions and action transitions are encoded separately for efficiency. The details are discussed in Section 3.

2.3 Simulation Relation

Since our model checking algorithms use the simulation relation, we introduce the simulation relation over timed automata in the following.

Definition 2. *Given a timed automaton A, a (location-based) simulation relation over states of $CS(A)$ is a binary relation $\mathcal{R} \subseteq S \times S$ such that for all $((l_1, v_1), (l_2, v_2)) \in \mathcal{R}$, it holds that:*

– $l_1 = l_2$
– *if $(l_1, v_1) \xrightarrow{d} (l_1, v_1 + d)$ then there exists d' such that $(l_2, v_2) \xrightarrow{d'} (l_2, v_2 + d')$ and $((l_1, v_1 + d), (l_2, v_2 + d')) \in \mathcal{R}$.*
– *if $(l_1, v_1) \xrightarrow{t} (l_1', v_1')$ then there exists (l_2', v_2') such that $(l_2, v_2) \xrightarrow{t} (l_2', v_2')$ and $((l_1', v_1'), (l_2', v_2')) \in \mathcal{R}$.*

A state (l_1, v_1) is simulated by another state (l_2, v_2), or equivalently (l_2, v_2) simulates (l_1, v_1)), denoted as $(l_1, v_1) \preccurlyeq (l_2, v_2)$, if there exists a simulation relation $\mathcal{R}$ such that $((l_1, v_1), (l_2, v_2)) \in \mathcal{R}$. By definition, any state simulates itself. Given a set of states $Q \subseteq S$, we define the downward closure [21] as $Down(Q) = \{s_1 \in S \mid \exists\, s_2 \in Q . s_1 \preccurlyeq s_2\}$. Intuitively, the downward closure of Q is the set of states which can be simulated by a state in Q. Since the simulation relation is reflexive, it follows that $Q \subseteq Down(Q)$.

For timed automata, it is known that there exists a simulation relation called the LU simulation relation [7]. Given a clock x, maximal lower bound $L(x)$ (respectively maximal upper bound $U(x)$) is the maximal constant k for which there exists a constraint $x > k$ or $x \geq k$ (respectively $x < k$ or $x \leq k$) in the timed automaton. If the maximal constant k does not exist, we set $L(x)$ (respectively $U(x)$) to $-\infty$. Then, given two clock valuations v and v', we denote $v \preccurlyeq v'$ if for all clocks $x \in X$, either $v'(x) = v(x)$ or $L(x) < v'(x) < v(x)$ or $U(x) < v(x) < v'(x)$. It shows the relation $\mathcal{R}_{CS} = \{((l, v), (l, v')) \mid v \preccurlyeq v'\}$ is a simulation relation based on $CS(A)$ [7]. The following proposition shows that it is also a simulation relation based on $DS(A)$.

Proposition 1. *The relation $\mathcal{R} = \{((l, v), (l, v')) \mid v, v' \in \mathbb{N}^{|X|} \wedge v \preccurlyeq v'\}$ is a simulation relation of DS(A).*

The proof is the same as Lemma 3 in [7]. For simplicity, we denote $\preccurlyeq$ the BDD encoding of the simulation relation $\mathcal{R}$ defined in Proposition 1.

Algorithm 1: Reachability Analysis

```
function $IsReach(Init, Tick, Trans, goal)$
    $Q_p = \varnothing$
    $Q = Init$
    $Q = Reach(Q, Trans)$
    while $(Q_p \neq Q)$ do
        $Q_p = Q$
        $Q{=}Q{\cup}Reach(succ(Q, Tick), Trans)$
        if $Q \cap goal \neq \varnothing$ then
            return true
        end if
    end while
    return false
end function

function $Reach(Q, R)$
    $Q_p = \varnothing$
    while $(Q_p \neq Q)$ do
        $Q_p = Q$
        $Q = Q \cup succ(Q, R)$
    end while
    return $Q$
end function
```

Algorithm 2: Reachability Analysis with Simulation

```
function $IsReach_{sim}(Init, Tick, Trans, goal)$
    $Q_p = \varnothing$
    $Q = Down(Init)$
    $Q = Reach_{sim}(Q, Trans)$
    while $(Q_p \neq Q)$ do
        $Q_p = Q$
        $Q{=}Q{\cup}Reach_{sim}(Down(succ(Q, Tick)), Trans)$
        if $Q \cap goal \neq \varnothing$ then
            return true
        end if
    end while
    return false
end function

function $Reach_{sim}(Q, R)$
    $Q_p = \varnothing$
    while $(Q_p \neq Q)$ do
        $Q_p = Q$
        $Q = Q \cup Down(succ(Q, R))$
    end while
    return $Q$
end function
```

3 Reachability Analysis Algorithm

In this section, we present the reachability analysis algorithm without the simulation reduction and the one with the reduction.

3.1 Algorithm without Simulation Reduction

Given a set of states $goal$, the reachability analysis is performed by computing the set of reachable states and checking whether this set contains some states belonging to $goal$. The problem of efficiently computing the set of reachable states in BDDs for timed systems has been investigated by Beyer in [11, 9]. There are two important observations to avoid exploding the BDD. First, separating action transitions and tick transitions is more efficient than using the union of these transitions as monolithic transitions. Second, to compute the fixpoint, using action transitions before applying tick transitions is successful to achieve smaller encodings of intermediate reachable states.

Algorithm 1 shows the reachability analysis algorithm based on Beyer's observations, without simulation reduction. The function $IsReach$ takes $Init$, $Tick$, $Trans$, and $goal$ as input. It checks whether a state in $goal$ is reachable from an initial state in $Init$ by transitions in $Tick$ or $Trans$. Moreover, given a set of states Q and a transition relation R, the utility function $Reach(Q, R)$ computes the set of states reachable from Q by transitions in R. We denote by $succ(Q, R)$ (or simply $succ(Q)$ if the transition

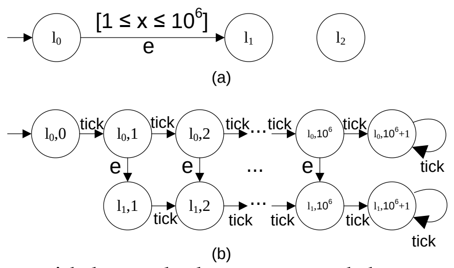

Fig. 1: Timed automaton with large clock constant and the transition system based on discrete semantics

relation R is clear from the context) the set of successor states of Q. Intuitively speaking, after the i^{th} loop (lines 5-11) iteration , Q stores the set of states reachable within i time units. The algorithm reaches the fixpoint if no new state is found in next time unit.

While Algorithm 1 is relatively efficient in computing the reachable states, it still suffers from large maximal clock constants. Models with large maximal clock constants require a large number of iterations to obtain the fixpoint. Figure 1a presents a timed automaton with large clock constant, i.e., with a maximal clock constant 10^6. We remark that in practice, large clock constants are not uncommon because different time units are often used in the same time. Figure 1b is the transition system generated by the discrete semantics. States whose location is l_2 are ignored in Figure 1b for simplicity. We denote (l_i, j) the state where the location is l_i and the clock valuation v such that $v(x) = j$. Assume the property is whether location l_2 is reachable. Then, Algorithm 1 requires $10^6 + 2$ iterations to reach the fixpoint to conclude that l_2 is unreachable. Specifically, $10^6 + 1$ iterations to find all the reachable states and the last iteration does not find any new state and concludes that the fixpoint is reached. On the contrary, with simulation reduction, our approach can verify whether l_2 is reachable within 3 iterations.

In the next section, we present our improved algorithm by using the simulation relation. We prove that the number of iterations can be reduced, and experimental results given in Section 5 confirm that our improved algorithm is much more efficient.

3.2 Algorithm with Simulation Reduction

In this section, we present our improved reachability analysis algorithm. Given a transition system $\mathcal{L}$, a simulation relation $\preccurlyeq$ over states of $\mathcal{L}$ and a set of states $goal$, our algorithm determines whether any state in $goal$ is reachable. The reachability analysis is performed similarly as Algorithm 1 by computing the set of reachable states and checking whether this set contains some state in $goal$.

We assume that the simulation on $\mathcal{L}$ is compatible with the set $goal$, i.e., for any $(s_1, s_2) \in \preccurlyeq$, $s_1 \in goal \Rightarrow s_2 \in goal$. In our reachability verification for timed automata, the LU simulation relation satisfies this condition because the reachability verification is over locations. Effectively, with simulation reduction, we would explore a reduced transition system defined as Def. 3.

Definition 3. *Given the transition system $\mathcal{L} = (C, init_c, \rightarrow)$ and the simulation relation $\preccurlyeq$, we define the transition system $\mathcal{L}' = (C', init'_c, \Rightarrow)$ such that:*

- *$C' = C$, $init'_c = Down(init_c)$*
- *Given any state $s'_1, s'_2 \in \mathcal{L}'$, there is a transition $s'_1 \Rightarrow s'_2$ in $\mathcal{L}'$ if there exists a transition $s'_1 \rightarrow s_2$ in $\mathcal{L}$ and $s'_2 \preccurlyeq s_2$.*

Note that the state space is unchanged. The initial states and transition functions are changed accordingly the simulation relation over the set of states C. Intuitively, for any transition $s'_1 \to s_2$ in $\mathcal{L}$, we allow other states simulated by s_2 to be successor states of s'_1 in $\mathcal{L}'$. Thus, given a set of states $Q \subseteq C$, $succ(Q, \Rightarrow) = Down(succ(Q, \to))$. In the following, we establish that $\mathcal{L}'$ preserves the reachability. For the sake of space, the proofs are presented in [2].

Lemma 1. *Given $q'_1 \preccurlyeq q_1$, if there exists a path of length n, $q'_1 \Rightarrow q'_2 \Rightarrow \cdots \Rightarrow q'_n$ in $\mathcal{L}'$, there exists a path of length n, $q_1 \to q_2 \to \cdots q_n$ in $\mathcal{L}$ such that $q'_i \preccurlyeq q_i$ for all $1 \leq i \leq n$.* □

Theorem 1. *Given the transition systems $\mathcal{L}$, $\mathcal{L}'$, and a set of states goal, goal is reachable in $\mathcal{L}$ if and only if goal is reachable in $\mathcal{L}'$.* □

Theorem 1 provides the relationship between transition systems $\mathcal{L}$ and $\mathcal{L}'$. Based on Theorem 1, we can use the transition system $\mathcal{L}'$ as the input for Algorithm 1. However, explicitly computing the transition relation of $\mathcal{L}'$ is computationally expensive. Instead, we apply $Down$ to the result of any call $succ(Q)$ on the fly in Algorithm 1 because $succ(Q, \Rightarrow) = Down(succ(Q, \to))$. Algorithm 2 presents our improved reachability analysis algorithm with simulation reduction. We rename the two functions as $IsReach_{sim}$ and $Reach_{sim}$ respectively. The difference between Algorithm 2 and Algorithm 1 is that in the function $IsReach_{sim}$, we first update $Q = Down(Init)$ at line 3, and subsequently, we call $Reach_{sim}(Q, R)$ and $Down(succ(Q, R))$ instead of $Reach(Q, R)$ and $succ(Q, R)$ respectively. It can be observed that we always apply $Down$ to the results of the $succ$ function.

Theorem 2. *Algorithm 2 is sound and complete.* □

Proof: As we discussed the difference between Algorithm 2 and Algorithm 1, given a transition system $\mathcal{L}$, while the function $IsReach(Init, Tick, Trans, g)$ checks the reachability of g on $\mathcal{L}$, the function $IsReach_{sim}(Init, Tick, Trans, g)$ actually checks the reachability of g on $\mathcal{L}'$. Thus, the correctness of Algorithm 2 is obtained based on Theorem 1 and the correctness of Algorithm 1.

Our algorithm is similar to the algorithm of antichain of promising states [21]. Note that in [21], the algorithm uses the Min operator while our approach uses $Down$ operator. The reason that our algorithm uses $Down$ operator is that it is efficient to compute in BDD. This algorithm is also similar to the one in [7], where LU simulation is used to improve zone-based verification of timed automata. However, the $Down$ operator here is coarser than extrapolation used in [7] (since for efficiency reasons, any extrapolation must result in convex zones).

Lemma 2. *Assume $Q' = Down(Q)$, $Q' \cup Reach_{sim}(Down(succ(Q', Tick)), Trans) = Down(Q \cup Reach(succ(Q, Tick), Trans))$.*

Lemma 3. *Assume $Q' = Down(Q)$, after n iterations, if $Reach(Q, R)$ reaches the fixpoint, $Reach_{sim}(Q', R)$ also reaches the fixpoint. Moreover the results of those functions satisfy $Reach_{sim}(Q', R) = Down(Reach(Q, R))$.*

Since the reachability analysis requires a lot of fixpoint computations, the rationale of Algorithm 2 is to converge faster to the fixpoint. In the following, we prove that

$Reach_{sim}(Down(Q))$ requires the same or smaller number of iterations to reach the fixpoint than $Reach(Q)$. In our proof, to distinguish with Algorithm 1, given any variable Q appearing in in Algorithm 2, we use the prime version Q' to denote that variable in Algorithm 2.

Theorem 3. *Algorithm 2 requires less or the same number of iterations than Algorithm 1.*

Proof: By Lemmas 2 and 3, in Algorithms 1 and 2, $Q' = Down(Q)$ holds. Thus, if Algorithm 1 terminates when $Q \cap goal \neq \varnothing$, Algorithm 2 also terminates because $Q' \cap goal \neq \varnothing$. Otherwise if $Q = Q_p$ holds in Algorithm 1, $Q' = Q'_p$ also holds in Algorithm 2. **Example.** In the following, we demonstrate how Algorithm 2 works using the example in Figure 1. The reachability problem is to check whether l_2 is reachable from the initial state l_0. According to timed automaton, we have $L(x) = 1$ and $U(x) = 10^6$. Algorithm 2 only takes 3 iterations to verify l_2 is unreachable, specifically:

- $Q'_0 = \{(l_0, 0)\}$
- $Q'_1 = \{(l_0, 0), (l_0, 1), (l_1, 1)\}$
- $Q'_2 = \{(l_0, i) \mid 0 \leq i \leq 10^6 + 1\} \cup \{(l_1, i) \mid 1 \leq i \leq 10^6 + 1\}$
- $Q'_3 = Q'_2$

Note that in the 2^{nd} iteration, at first we have $(l_0, 2), (l_1, 2) \in Q'_2$. Since $(l_0, i) \preccurlyeq (l_0, 2)$ and $(l_1, i) \preccurlyeq (l_1, 2)$ for all $i > 2$, we add all states (l_0, i) and (l_1, i) where $i > 2$ to Q'_2 by the $Down$ function. Thus, finally $Q'_2 = \{(l_0, i) \mid 0 \leq i \leq 10^6 + 1\} \cup \{(l_1, i) \mid 1 \leq i \leq 10^6 + 1\}$.

In this section, we have presented our improved algorithm for reachability verification by using the LU simulation relation. We prove that our approach in Algorithm 2 always uses less or the same number of iterations compared with the classic algorithm as in Algorithm 1. In the next section, we continue with presenting our improved emptiness checking algorithm with the simulation relation.

Algorithm 3: Algorithm *IsEmpty*

```
1:  function $IsEmpty(Init, Tr, J)$
2:      $old = \varnothing$
3:      ▷ Empty line for comparision with Algorithm 4
4:      $new = Reach(Init, Tr)$
5:      while $(new \neq old)$ do
6:          $old = new$
7:          for all $J_i \in J$ do
8:              $new = Reach(new \cap J_i, Tr)$
9:          end for
10:         while $(new \neq (new \cap$
11: $succ(new)))$ do
12:             $new = (new \cap succ(new))$
13:         end while
14:     end while
15:     return $(new = \varnothing)$
16: end function
```

Algorithm 4: Algorithm $IsEmpty_{sim}$

```
1:  function $IsEmpty_{sim}(Init, Tr, J)$
2:      $old = \varnothing$
3:      $Init = Down(Init)$
4:      $new = Reach_{sim}(Init, Tr)$
5:      while $(new \neq old)$ do
6:          $old = new$
7:          for all $J_i \in J$ do
8:              $new = Reach_{sim}(new \cap J_i, Tr)$
9:          end for
10:         while $(new \neq (new$
11: $\cap Down(succ(new))))$ do
12:             $new = (new \cap Down(succ(new)))$
13:         end while
14:     end while
15:     return $(new = \varnothing)$
16: end function
```

4 Emptiness Checking Algorithm

Under digitization and automata theory, LTL verification can be done by emptiness checking. Thus, the emptiness checking algorithm of Kesten *et al.* [27] can be used. In this section, we fist present the algorithm of Kesten. Then, we introduce our improved algorithm by using the simulation relation.

4.1 Algorithm without Simulation Reduction

Given a transition system and a set of Büchi conditions J where $J_i \in J$ is a set of states, an accepting run is an infinite run which visits a J_i-state (a state in J_i) infinitely often for all $J_i \in J$. The emptiness problem is to check whether this run exists.

For simplicity, in this section, we merge $Trans$ and $Tick$ and assume that Tr is the encoding of the whole transition system. Algorithm 3 [27] presents the symbolic emptiness checking algorithm. Specifically, function $IsEmpty$ takes the set of the initial states $Init$, the transition relation Tr, and a set of Büchi conditions J as input.

In Algorithm 3, function $IsEmpty$ searches for an accepting strongly connected component (SCC) which contains a J_i-state for every Büchi condition $J_i \in J$. The algorithm computes the set of all reachable accepting SCCs. If this set is empty, there is no accepting run in the given transition system. At line 4, new is assigned as the set of all reachable states from the initial states. Then, the while-loop (from line 5 to line 14) continuously refines the set of states new until a fixpoint is reached (i.e., $new = old$ at line 5). Inside this while-loop, first, we backup the current value of new in old (line 6). Then, from line 7 to line 9, we continue to refine new as the set of states reachable by a J_i-state for all $J_i \in J$. Next, in the inner while-loop from line 11 to line 13, we again refine new by successively removing from new states which do not have a predecessor in new (line 12). This loop is iterated until new is closed under predecessor. Thus, new is the set of all reachable SCCs. Because of the loop from line 7 to line 9, those SCCs are accepting by containg a state in J_i for all $J_i \in J$. At the end, new contains all reachable accepting SCCs in this transition system.

4.2 Algorithm with Simulation Reduction

In this section, we present our improved emptiness checking algorithm of timed automata Algorithm 4, which improves Algorithm 3 by using the simulation relation. We rename the function as $IsEmpty_{sim}$. The difference between Algorithm 4 and Algorithm 3 is that in the function $IsEmpty_{sim}$, at the beginning, we update $Init = Down(Init)$ at line 3, and throughout the algorithm, we call the functions $Reach_{sim}(Q, Tr)$ and $Down(succ(Q))$ instead of $Reach(Q, Tr)$ and $succ(Q)$, respectively. Note that the function $Reach_{sim}(Q, Tr)$ is introduced in Section 3. In other words, we always apply the function $Down$ on the results of the $succ$ function. We prove that Algorithm 4 is sound and complete as we did for Algorithm 3. First, we prove that $\mathcal{L}'$ (defined in Definition 3) also preserves the emptiness.

Lemma 4. *Given $q'_1 \preccurlyeq q_1$, if there exists a path of length n, $q'_1 \Rightarrow q'_2 \Rightarrow \cdots \Rightarrow q'_n$ in $\mathcal{L}'$, there exists a path of length n, $q_1 \to q_2 \to \cdots q_n$ in $\mathcal{L}$ such that $q'_i \preccurlyeq q_i$ for all $1 \leq i \leq n$.*

Lemma 5. *Given $q'_1 \preccurlyeq q_1$, if there exists a cycle $q'_1 \Rightarrow \cdots \Rightarrow q'_1$ in $\mathcal{L}'$ which contains a J_i-state for all $J_i \in J$, there exists a cycle $q_1 \to \cdots \to q_1$ in $\mathcal{L}$ which contains a J_i-state for all $J_i \in J$.*

Lemma 6. *If there exists an accepting run in $\mathcal{L}'$, there exists an accepting run in $\mathcal{L}$.*

Theorem 4. *Given a transition system $\mathcal{L}$, a set of Büchi conditions J, and a simulation relation $\preccurlyeq$ over states of $\mathcal{L}$, $\mathcal{L}$ has an accepting run if and only if $\mathcal{L}'$ has an accepting run.*

Following Theorem 4, we can use the transition system $\mathcal{L}'$ as the input for Algorithm 3. However, explicitly computing the transition relation of $\mathcal{L}'$ is not efficient. Instead, we apply $Down$ for the result of any call $succ(Q)$ on the fly in Algorithm 3 because of the fact that $succ(Q, \Rightarrow) = Down(succ(Q, \rightarrow))$.

Theorem 5. *Algorithm 4 is sound and complete.*

Proof: As we discussed the difference between Algorithm 4 and Algorithm 3, given a transition system $\mathcal{L}$ with a set of initial states $Init$, the transition relation Tr and a set of Büchi conditions J, while $IsEmpty(Init, Tr, J)$ is checking the emptiness of $\mathcal{L}$, $IsEmpty_{sim}(Init, Tr, J)$ is actually checking the emptiness of the transition system $\mathcal{L}'$. Thus, the correctness of Algorithm 4 is obtained based on Theorem 4. $\square$

Algorithm 4 does not guarantee that it always takes less or the same number of iterations than Algorithm 3. To distinguish between Algorithms 4 and 3, we use new' and new to denote the variable new in Algorithm 4 and Algorithm 3 respectively. Then, the reason that Algorithm 4 might take more iterations is $new' = Down(new)$ is not an invariant during the algorithm. Assume before executing the line 12, it holds that $new' = Down(new)$, then $new' = Down(new)$ may not hold after this line is executed as shown in Lemma 8 in [2]. Thus, $new' = Down(new)$ is not an invariant. Nevertheless, in our evaluation in Section 5, Algorithm 4 always outperforms Algorithm 3 and takes less number of $succ$ function calls. The reason is that during the computation of all reachable states from initial states at line 4 and the first run of the while-loop in lines 7-9, Algorithm 4 can take much lesser number of $succ$ function calls than Algorithm 3 as the result of Theorem 3 and Lemma 7 in [2]. Moreover, the computation of all reachable states (line 4) is the most expensive computation in these algorithms.

Algorithm 4 can be adopted to verify the emptiness of TBA straightforwardly. The requirement that the run must visit an accepting location infinite times and contain an infinite number of tick transitions and action transitions is represented as a set of Büchi conditions $J = \{Acc, J_0, J_1\}$ where Acc is a set of accepting locations in $DS(A)$ and J_0 (respectively J_1) is the set of states which are the destination states of the action transition (respectively tick transition). A boolean variable $isTick$ can be introduced during the encoding. For each transition, this variable is updated to false if that is an action transition. Otherwise it is updated to true. Then J_0 is the set of states where $isTick$ is false and J_1 is the set of states where $isTick$ is true.

We have presented our approach on using the LU simulation relation on the verification of reachability and LTL properties. We evaluate both algorithms next.

5 Evaluation

We conducted experiments to evaluate our approach. Specifically, we attempted to answer the following research questions:

RQ1: How is the *improvement* in the number of iterations and verification time of our methods, compared to the existing state-of-the-art BDD-based and DBM-based methods, in checking reachability and LTL properties?

RQ2: How *scalable* is our method in size of maximal clock constants and processes?

Table 1: Experimental results in the reachability verification with increasing clock constants

		PAT-Sim			PAT-NonSim			Rabbit
	MCC	#Succ	Time	Memory	#Succ	Time	Memory	Time
CSMACD	808	4,369	6	34	17,794	1,563	577	208
CSMACD	1,616	8,721	36	59	-	oot	-	1,494
CSMACD	3,232	17,425	228	181	-	-	-	oot
Fischer	256	796	14	73	2,838	1,033	1,089	58
Fischer	512	1,564	112	252	-	-	oom	1,076
Fischer	1,024	3,100	867	931	-	-	-	oom
Lynch	64	481	12	66	1,347	217	498	256
Lynch	128	929	104	287	2,627	2,163	1,562	oot
Lynch	256	1,825	859	1,003	-	-	oom	oom

Our approach has been implemented as a BDD library for the reachability and LTL verification of timed automata in the PAT framework [40]. Our implementation is based on the CUDD package [39], which is a package that provides functions to manipulate BDDs. All of the experiments are performed on a PC with Intel Core i7-2600 CPU at 3.4GHz and 4GB RAM.

To answer the research questions, we have conducted four experiments, and the results are shown in Tables 1-4. For all experiments, we measure the number of $succ$ function calls ($\#Succ$), the verification time (in seconds) ($Time$), and the memory usage of CUDD library (in MB) ($Memory$) over three benchmark systems from [1, 15, 34]: CSMACD protocol, Fischer's protocol, and Lynch-Shavit protocol. We run PAT in two settings, i.e., with and without simulation, which are referred to as *PAT-Sim* and *PAT-NonSim*. The algorithms for PAT-Sim (PAT-NonSim resp.) on verifying reachability and LTL properties are given in Algorithms 2 and 4 (Algorithms 1 and 3 resp.).

All experiments are conducted with a time limit of 2 CPU hours. An entry 'oot' in the table means that the time limit is reached, and an entry 'oom' means that the program runs out of memory. Given a benchmark system, when a smaller model is running out of time or memory, we omit the evaluation of larger models. An entry '-' means the information is unavailable.

We compare the results to three state-of-the-art model checkers, i.e., DBM-based model checker *Uppaal* [30] and *CTAV* [31], as well as BDD-based model checker *Rabbit* [10]. Although RED [43] and BDD-based version of Kronos [14] are related to our work as real time verification tools using BDD (BDD-like) data structure, Rabbit was shown to outperform them [10]. Therefore, only Rabbit is used in our experiments.

5.1 Evaluation for Reachability Properties

We evaluate our approach with Rabbit and Uppaal in the verification of reachability properties. Since our approach is digitization-based, naturally, the first question is how well the library scales with the number of clock ticks. In the first experiment (cf. Table 1), we exponentially increase the maximal clock constants while keeping the number of processes constant (we set it 4). Since Uppaal is a DBM-based model checker, its performance does not depend on the maximal clock constants; therefore, it is not used in the experiment. The column *MCC* is the maximal clock constant values in the corresponding models. Compared to PAT-NonSim, PAT-Sim takes smaller number of $succ$ function calls which can be reduced from 2 to 4 times by using simulation. Compared to Rabbit, PAT-Sim achieves a speedup from 2 to 21 times, and there are five

Table 2: Experimental results in the reachability verification with increasing number of processes

		PAT-Sim			PAT-NonSim			Rabbit	Uppaal
	#Proc	#Succ	Time	Memory	#Succ	Time	Memory	Time	Time
CSMACD	16	7,377	62	85	-	oot	-	5,638	oom
CSMACD	32	14,289	453	187	-	-	-	oot	-
CSMACD	64	26,801	3,912	477	-	-	-	-	-
Fischer	8	308	52	482	-	oot	-	7,258	0.7
Fischer	16	356	366	1,442	-	-	-	oom	oom
Fischer	32	452	3,351	1,651	-	-	-	-	-
Lynch	8	169	8	72	696	6,203	1,690	2,494	1.1
Lynch	16	217	104	290	-	-	oom	oom	oom
Lynch	32	313	2,971	1,201	-	-	-	-	-

cases where Rabbit runs out of memory or time. As a result, PAT-Sim outperforms both PAT-NonSim and Rabbit and can handle larger maximal clock constants.

In the second experiment (cf. Table 2), we compare PAT, Rabbit, and Uppaal using the same benchmark systems. The column $\#Proc$ represents the number of processes. In this experiment, we set the maximal clock constants to 64 in Fischer protocol, 16 in Lynch-Shavit protocol, and 404 in CSMACD protocol. Then, we increase the number of processes in each benchmark system to find out which tool can verify the most number of processes. By using simulation, the number of $succ$ function calls is reduced. Thus, PAT-Sim is faster and can handle larger number of processes compared to PAT-NonSim. For example, in the Lynch model with 8 processes, PAT-Sim requires 169 $succ$ function calls and takes 8 seconds, while PAT-NonSim requires 696 $succ$ function calls and takes 6,203 seconds. The verification time is thus reduced significantly. According to Table 2, PAT-Sim also outperforms Rabbit and Uppaal. Although Uppaal achieves shorter evaluation time in smaller number of processes, both Rabbit and Uppaal easily run out of memory or time when the number of processes increases. On the contrary, PAT-Sim can still verify models while both other tools are out of memory or time, for example, 64 processes in the CSMACD benchmark.

5.2 Evaluation for LTL Properties

We evaluate our approach with CTAV in the verification of LTL properties under non-Zeno condition. Note that we do not compare with Uppaal since Uppaal does not support the verification of LTL properties under non-Zeno condition. In the third experiment (cf. Table 3), to demonstrate the efficiency of our approach in the handling of large maximal clock constants, we fix the number of processes at 4 and increase the maximal clock constants. We do not compare with CTAV since it is a DBM-based model checker and its performance is not affected by maximal clock constants. According to the results, by using the LU simulation relation, the number of $succ$ function calls is reduced significantly. For example, in the Lynch protocol with $MCC = 200$, the number of $succ$ calls is reduced from 19,682 to 6,937. As a result, the verification time is improved significantly, from 2,404s to 25s.

PAT-Sim outperforms PAT-NonSim on all the models. It is faster and uses less memory. Thus, it can handle models with maximal clock constants up to thousands.

In the fourth experiment (cf. Table 4), to demonstrate the efficiency of our approach in the handling of large number of processes, we fix the maximal clock constant as

Table 3: Experimental results in the LTL verification with increasing maximal clock constants

		PAT-Sim			PAT-NonSim		
	MCC	#Succ	Time	Memory	#Succ	Time	Memory
CSMACD	404	4,334	5	36	14,169	493	876
CSMACD	808	8,608	18	75	28,257	2,857	1,489
CSMACD	1,616	16,688	35	82	-	-	oom
Fischer	200	979	2	28	2,812	417	1,101
Fischer	400	1,779	3	29	5,412	3,847	1,600
Fischer	800	3,379	8	34	-	oot	-
Lynch	200	6,937	25	53	19,682	2,404	1,434
Lynch	400	13,137	45	62	-	oot	-
Lynch	800	25,537	90	63	-	-	-

Table 4: Experimental results in the LTL verification with increasing number of processes

		PAT-Sim			PAT-NonSim			CTAV
	#Proc	#Succ	Time	Memory	#Succ	Time	Memory	Time
CSMACD	12	22,184	283	1,041	-	oot	-	562
CSMACD	16	28,972	511	756	-	-	-	oom
CSMACD	20	35,760	839	1,063	-	-	-	-
Fischer	8	608	5	39	1,974	10,275	1,689	4
Fischer	12	672	46	208	-	-	oom	oom
Fischer	16	736	310	965	-	-	-	-
Lynch	4	3,591	1	25	10,003	243	329	1
Lynch	8	9,839	42	65	-	-	oom	5
Lynch	12	19,551	585	326	-	-	-	oom

808 for CSMACD and 100 for other benchmarks. We increase the number of processes then. In this experiment, we compare our approach with CTAV tool. The results indicate PAT-Sim approach outperforms PAT-NonSim and CTAV on all the models. Specifically, it is faster and can handle more processes than PAT-NonSim and CTAV. For example, in the CSMACD model with 16 processes, PAT-Sim can verify within 511 seconds and 756 MB while PAT-NonSim runs out of time, and CTAV runs out of memory.

With the results of four experiments, we answer research questions *RQ1* and *RQ2*. Our approach improves the performance significantly by reducing the number of iterations. Furthermore, it can handle models with clock constants larger than a thousand.

6 Discussion

Limitation. A limitation of our approach is that when maximal lower and upper bounds are the same, LU abstraction would not provide better performance. This is because our method will take the same number of iterations to achieve the fixpoint, and there are overhead for calling the *Down* operator.

Complexity of Down operator [7]. For checking of reachability properties, given the maximal distance from the initial state to a state in the explored model as N, the complexity is *O(N)*. For checking of LTL properties, the time complexity is linearly dependent upon the size of the symbolic (BDD) representation in terms of the distances between states in the automaton graph, the number and arrangement of the strongly connected components in the graph, and the number of fairness conditions asserted [37]. Overall, *Down* operator can be computed efficiently. In addition, variable ordering could affect the performance of BDD. Overall, the *Down* operator can be computed efficiently. In our implementation, we make use of several well-known heuristics [22, 6, 9, 36] that can produce a fairly good ordering.

7 Conclusion

In this paper, we propose to use the simulation relation to improve the BDD-based model checking for real-time systems. Our approach is applied to verify reachability and LTL properties. Experimental results confirm that our approach achieves a significant speedup and outperforms Rabbit, Uppaal, and CTAV. As future works, first, we plan to investigate the extensibility our method to other variety of timed automata, such as parametric timed automata [4]. Second, we plan to investigate to apply other reduction techniques, e.g., interpolation [32] or IC3 [17], on top of our proposed techniques.

References

1. MCMT Benchmarks of Timed Automata. `http://crema.di.unimi.it/~carioni/mcmt_ta.html`.
2. Technical Report of Scaling BDD-based Timed Verification with Simulation Reduction. `http://tianhuat.github.io/tr_bddsr.pdf`.
3. R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
4. R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric Real-Time Reasoning. In *STOC*, pages 592–601, 1993.
5. E. Asarin, O. Maler, and A. Pnueli. On Discretization of Delays in Timed Automata and Digital Circuits. In *CONCUR*, pages 470–484, 1998.
6. A. Aziz, S. Tasiran, and R. K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In *DAC*, pages 283–288, 1994.
7. G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and Upper Bounds in Zone Based Abstractions of Timed Automata. In *TACAS*, pages 312–326, 2004.
8. G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *CAV*, pages 341–353, 1999.
9. D. Beyer. Improvements in BDD-Based Reachability Analysis of Timed Automata. In *FME*, pages 318–343, 2001.
10. D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A Tool for BDD-Based Verification of Real-Time Systems. In *CAV*, pages 122–125, 2003.
11. D. Beyer and A. Noack. Efficient Verification of Timed Automata using BDDs. In *FMICS*, pages 95–113, 2001.
12. D. Beyer and A. Noack. Can Decision Diagrams Overcome State Space Explosion in Real-Time Verification? In *FORTE*, pages 193–208, 2003.
13. P. Bouyer. Forward Analysis of Updatable Timed Automata. *Formal Methods in System Design*, 24(3):281–320, 2004.
14. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In *CAV*, pages 546–550, 1998.
15. M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some Progress in the Symbolic Verification of Timed Automata. In *CAV*, pages 179–190, 1997.
16. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.
17. A. Cimatti and A. Griggio. Software model checking via ic3. In *Computer Aided Verification*, pages 277–293. Springer, 2012.
18. C. Daws and S. Tripakis. Model Checking of Real-Time Reachability Properties Using Abstractions. In *TACAS*, pages 313–329, 1998.

19. D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989.
20. D. L. Dill, A. J. Hu, and H. Wong-Toi. Checking for Language Inclusion Using Simulation Preorders. In *CAV*, pages 255–265, 1991.
21. L. Doyen and J. Raskin. Antichain Algorithms for Finite Automata. In *TACAS*, pages 2–22, 2010.
22. H. Fujii, G. Ootomo, and C. Hori. Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams. In *ICCAD*, pages 38–41, 1993.
23. T. A. Henzinger, Z. Manna, and A. Pnueli. What Good Are Digital Clocks? In *ICALP*, pages 545–558, 1992.
24. F. Herbreteau and B. Srivathsan. Efficient On-the-Fly Emptiness Check for Timed Büchi Automata. In *ATVA*, pages 218–232, 2010.
25. F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Efficient Emptiness Check for Timed Büchi Automata. In *CAV*, pages 148–161, 2010.
26. F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Better Abstractions for Timed Automata. In *LICS*, pages 375–384, 2012.
27. Y. Kesten, A. Pnueli, and L. on Raviv. Algorithmic Verification of Linear Temporal Logic Specifications. In *ICALP*, pages 1–16, 1998.
28. A. Laarman, M. C. Olesen, A. E. Dalsgaard, K. G. Larsen, and J. van de Pol. Multi-core Emptiness Checking of Timed Büchi Automata Using Inclusion Abstraction. In *CAV*, pages 968–983, 2013.
29. L. Lamport. Real-Time Model Checking Is Really Simple. In *CHARME*, pages 162–175, 2005.
30. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *STTT*, 1(1-2):134–152, 1997.
31. G. Li. Checking Timed Büchi Automata Emptiness Using LU-Abstractions. In *FORMATS*, pages 228–242, 2009.
32. K. L. McMillan. Interpolation and sat-based model checking. In *CAV*, pages 1–13. Springer, 2003.
33. J. B. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference Decision Diagrams. In *CSL*, pages 111–125, 1999.
34. G. Morbé, F. Pigorsch, and C. Scholl. Fully Symbolic Model Checking for Timed Automata. In *CAV*, pages 616–632, 2011.
35. T. K. Nguyen, J. Sun, Y. Liu, J. S. Dong, and Y. Liu. Improved BDD-based Discrete Analysis of Timed Systems. In *FM*, pages 326–340, 2012.
36. M. Rice and S. Kulhari. A Survey of Static Variable Ordering Heuristics for Efficient BDD/MDD Construction. Technical report, University of California, Riverside, 2008.
37. K. Y. Rozier. Linear temporal logic symbolic model checking. *Computer Science Review*, 5(2):163–203, 2011.
38. S. A. Seshia and R. E. Bryant. Unbounded, Fully Symbolic Model Checking of Timed Automata using Boolean Methods. In *CAV*, pages 154–166, 2003.
39. F. Somenzi. CUDD: CU Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/`.
40. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 709–714, 2009.
41. S. Tripakis. Verifying Progress in Timed Systems. In *ARTS*, pages 299–314, 1999.
42. S. Tripakis. Checking Timed Büchi Automata Emptiness on Simulation Graphs. *ACM Transactions on Computational Logic*, 10(3):1–19, 2009.
43. F. Wang. Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram. In *FORTE*, pages 235–250, 2001.
44. F. Wang. Efficient Verification of Timed Automata with BDD-Like Data-Structures. In *VMCAI*, pages 189–205, 2003.

Appendix: Proof

3 Reachability Analysis Algorithm

Lemma 1. *Given $q'_1 \preccurlyeq q_1$, if there exists a path of length n, $q'_1 \Rightarrow q'_2 \Rightarrow \cdots \Rightarrow q'_n$ in $\mathcal{L}'$, there exists a path of length n, $q_1 \rightarrow q_2 \rightarrow \cdots q_n$ in $\mathcal{L}$ such that $q'_i \preccurlyeq q_i$ for all $1 \leq i \leq n$.*

Proof: We prove by induction that the lemma is true for all cases of n.

- $n = 1$: This is vacuously true because $q'_1 \preccurlyeq q_1$
- Suppose for any path of length $n = k$, $q'_1 \Rightarrow q'_2 \Rightarrow \cdots \Rightarrow q'_k$ in $\mathcal{L}'$, there exists a path of length k, $q_1 \rightarrow q_2 \rightarrow \cdots q_k$ in $\mathcal{L}$ such that $q'_i \preccurlyeq q_i$ for all $1 \leq i \leq k$. We prove that this is still true for $n = k + 1$. Given any path of length $k + 1$, $q'_1 \Rightarrow q'_2 \Rightarrow \cdots \Rightarrow q'_k \Rightarrow q'_{k+1}$, in $\mathcal{L}'$, by the induction assumption, there exists $q_1 \rightarrow q_2 \rightarrow \cdots q_k$ in $\mathcal{L}$ such that $q'_i \preccurlyeq q_i$ for all $1 \leq i \leq k$. By definition of $\mathcal{L}'$, $q'_k \Rightarrow q'_{k+1}$ implies $q'_k \rightarrow r'$ where $q'_{k+1} \preccurlyeq r'$. Moreover, since in $\mathcal{L}$, $q'_k \preccurlyeq q_k$, there exists a transition $q_k \rightarrow r$ in $\mathcal{L}$ such that $r' \preccurlyeq r$. Thus, $q'_{k+1} \preccurlyeq r$. Therefore, we can select $q_{k+1} = r$, and the lemma is proved. □

Theorem 1. *Given the transition systems $\mathcal{L}$, $\mathcal{L}'$, and a set of states goal, goal is reachable in $\mathcal{L}$ if and only if goal is reachable in $\mathcal{L}'$.*

Proof: If $goal$ is reachable in $\mathcal{L}$, there exists a finite path $q_1 \rightarrow q_2 \rightarrow \cdots q_n$ in $\mathcal{L}$ such that $q_1 \in init_c$ and $q_n \in goal$. As the definition of $\mathcal{L}'$, this path also exists in $\mathcal{L}'$ and $q_1 \in init'_c$. Thus, $goal$ is reachable in $\mathcal{L}'$.

If $goal$ is reachable in $\mathcal{L}'$, there exists a finite path $q'_1 \Rightarrow q'_2 \Rightarrow \cdots \Rightarrow q'_n$ in $\mathcal{L}'$ such that $q'_1 \in init'_c$ and $q'_n \in goal$. According to Lemma 1, there exists a finite path $q_1 \rightarrow q_2 \rightarrow \cdots q_n$ in $\mathcal{L}$ such that $q_1 \in init_c$ and $q'_i \preccurlyeq q_i$ for all $1 \leq i \leq n$. Thus, $q_n \in goal$. Therefore, $goal$ is also reachable in $\mathcal{L}$. □

Theorem 2. *Algorithm 2 is sound and complete.*

Proof: As we discussed the difference between Algorithm 2 and Algorithm 1, given a transition system $\mathcal{L}$, while the function $IsReach(Init, Tick, Trans, goal)$ checks the reachability of $goal$ on $\mathcal{L}$, the function $IsReach_{sim}(Init, Tick, Trans, goal)$ actually checks the reachability of $goal$ on $\mathcal{L}'$. Thus, the correctness of Algorithm 2 is obtained based on Theorem 1 and the correctness of Algorithm 1. □

Corollary 1. *The followings are hold*

1. *If $Q_1 \subseteq Q_2$, $Down(Q_1) \subseteq Down(Q_2)$.*
2. *$Down(Q_1 \cap Q_2) \subseteq Down(Q_1) \cap Down(Q_2)$.*
3. *$Down(Q_1 \cup Q_2) = Down(Q_1) \cup Down(S_2)$*
4. *$Down(succ(Q)) = Down(succ(Down(Q)))$*

Proof:

1. Given any state $s' \in Down(Q_1)$, there exists $s \in Q_1$ such that $s' \preccurlyeq s$. Since $Q_1 \subseteq Q_2$, it follows that $s \in Q_2$. Thus, $s' \in Down(Q_2)$.
2. According to Corollary 1.1, $Down(Q_1 \cap Q_2) \subseteq Down(Q_1)$ and $Down(Q_1 \cap Q_2) \subseteq Down(Q_2)$. Thus, $Down(Q_1 \cap Q_2) \subseteq Down(Q_1) \cap Down(Q_2)$.
3. According to Corollary 1.1, $Down(Q_1) \subseteq Down(Q_1 \cup Q_2)$ and $Down(Q_2) \subseteq Down(Q_1 \cup Q_2)$. Thus, $Down(Q_1) \cup Down(S_2) \subseteq Down(Q_1 \cup Q_2)$. Then, we prove that $Down(Q_1 \cup Q_2) \subseteq Down(Q_1) \cup Down(S_2)$. Given any $s' \in Down(Q_1 \cup Q_2)$, there exists $s \in Q_1 \cup Q_2$ such that $s' \preccurlyeq s$. If $s \in Q_1$, then $s' \in Down(Q_1)$. Otherwise, if $s \in Q_2$, then $s' \in Down(Q_2)$. Therefore, $s' \in Down(Q_1) \cup Down(Q_2)$.
4. Since $Q \subseteq Down(Q)$, $succ(Q) \subseteq succ(Down(Q))$. It follows that $Down(succ(Q)) \subseteq Down(succ(Down(Q)))$. We prove $Down(succ(Down(Q))) \subseteq Down(succ(Q))$. Given any $s \in Down(succ(Down(Q)))$, there exists $s_1 \in Q$, s_2, and s_3 such that $s_2 \preccurlyeq s_1$, $s_2 \rightarrow s_3$, and $s \preccurlyeq s_3$. However, since $s_2 \preccurlyeq s_1$, there exists s_3' such that $s_1 \rightarrow s_3'$ and $s_3 \preccurlyeq s_3'$. Thus, $s \preccurlyeq s_3'$. We have $s_1 \in Q$, $s_1 \rightarrow s_3'$, and $s \preccurlyeq s_3'$, therefore, $s \in Down(succ(Q))$.

□

Lemma 2. *Assume $Q' = Down(Q)$, after n iterations, if $Reach(Q, R)$ reaches the fixpoint, $Reach_{sim}(Q', R)$ also reaches the fixpoint. Moreover the results of those functions satisfy $Reach_{sim}(Q', R) = Down(Reach(Q, R))$.*

Proof: Let Q_i (respectively Q_i') be the value of Q in the function $Reach$ (respectively $Reach_{sim}$) after the i^{th} iteration in the while-loop . We prove by induction that $Q_i' = Down(Q_i)$ for all $i \geq 0$.

– $i = 0$: $Q_0' = Q' = Down(Q) = Down(Q_0)$
– Suppose $Q_k' = Down(Q_k)$, we prove that $Q_{k+1}' = Down(Q_{k+1})$. We have $Q_{k+1}' = Q_k' \cup Down(succ(Q_k')) = Down(Q_k) \cup Down(succ(Down(Q_k))) = Down(Q_k) \cup Down(succ(Q_k)) = Down(Q_k \cup succ(Q_k)) = Down(Q_{k+1})$.

Since $Q_i' = Down(Q_i)$, when the function $Reach$ convergences, $Q_i = Q_{i+1}$, $Reach_{sim}$ also convergences, $Q_i' = Q_{i+1}'$. Moreover, the results of those functions satisfy $Reach_{sim}(Q', R) = Down(Reach(Q, R))$. □

Lemma 3. *Assume $Q' = Down(Q)$, $Q' \cup Reach_{sim}(Down(succ(Q', Tick)), Trans) = Down(Q \cup Reach(succ(Q, Tick), Trans))$*

Proof: By Corollary 1.4, it follows that $Down(succ(Q', Tick))$
$= Down(succ(Down(Q), Tick)) = Down(succ(Q, Tick))$.
Thus, by Lemma 2, we obtain
$Reach_{sim}(Down(succ(Q', Tick))) = Down(Reach(succ(Q, Tick)))$. Therefore, $Q' \cup Reach_{sim}(Down(succ(Q', Tick)), Trans) = Down(Q \cup Reach(succ(Q, Tick), Trans))$ by Corollary 1.3. □

Theorem 3. *Algorithm 2 requires less or the same number of iterations than Algorithm 1.*

Proof: According to Lemmas 2 and 3, in Algorithms 1 and 2, $Q' = Down(Q)$ holds. Thus, if Algorithm 1 terminates when $Q \cap goal \neq \varnothing$, Algorithm 2 also terminates because $Q' \cap goal \neq \varnothing$. Otherwise if $Q = Q_p$ holds in Algorithm 1, $Q' = Q'_p$ also holds in Algorithm 2. □

4 Emptiness Checking Algorithm

Lemma 4. *Given $q'_1 \preccurlyeq q_1$, if there exists a path of length n, $q'_1 \Rightarrow q'_2 \Rightarrow \cdots \Rightarrow q'_n$ in $\mathcal{L}'$, there exists a path of length n, $q_1 \rightarrow q_2 \rightarrow \cdots q_n$ in $\mathcal{L}$ such that $q'_i \preccurlyeq q_i$ for all $1 \leq i \leq n$.*

Proof: We prove by induction that the lemma is true for all cases of n.

- $n = 1$: This is vacuously true because $q'_1 \preccurlyeq q_1$
- Suppose for any path of length $n = k$, $q'_1 \Rightarrow q'_2 \Rightarrow \cdots \Rightarrow q'_k$ in $\mathcal{L}'$, there exists a path of length k, $q_1 \rightarrow q_2 \rightarrow \cdots q_k$ in $\mathcal{L}$ such that $q'_i \preccurlyeq q_i$ for all $1 \leq i \leq k$. We prove that this is still true for $n = k + 1$. Given any path of length $k + 1$, $q'_1 \Rightarrow q'_2 \Rightarrow \cdots \Rightarrow q'_k \Rightarrow q'_{k+1}$, in $\mathcal{L}'$, by the induction assumption, there exists $q_1 \rightarrow q_2 \rightarrow \cdots q_k$ in $\mathcal{L}$ such that $q'_i \preccurlyeq q_i$ for all $1 \leq i \leq k$. By definition of $\mathcal{L}'$, $q'_k \Rightarrow q'_{k+1}$ implies that there exists r' such that $q'_k \rightarrow r'$ and $q'_{k+1} \preccurlyeq r'$. Moreover, since in $\mathcal{L}$, $q'_k \preccurlyeq q_k$, there exists a transition $q_k \rightarrow r$ in $\mathcal{L}$ such that $r' \preccurlyeq r$. Thus, $q'_{k+1} \preccurlyeq r$. Therefore, we can select $q_{k+1} = r$, and the lemma is proved. □

Lemma 5. *Given $q'_1 \preccurlyeq q_1$, if there exists a cycle $q'_1 \Rightarrow \cdots \Rightarrow q'_1$ in $\mathcal{L}'$ which contains a J_i-state for all $J_i \in J$, there exists a cycle $q_1 \rightarrow \cdots \rightarrow q_1$ in $\mathcal{L}$ which contains a J_i-state for all $J_i \in J$.*

Proof: We prove by induction that for all $n \geq 2$, in $\mathcal{L}$, there exists $q_1, \cdots, q_n$ such that for all $1 \leq i < n$, $q_i \rightarrow \cdots \rightarrow q_{i+1}$ and $q'_1 \preccurlyeq q_i$ for all $1 \leq i \leq n$.

- $n = 2$: Since $q'_1 \preccurlyeq q_1$, and there exists $q'_1 \Rightarrow \cdots \Rightarrow q'_1$ in $\mathcal{L}'$, by applying Lemma 4, there exists $q_1 \rightarrow \cdots \rightarrow q_2$ in $\mathcal{L}$ where $q'_1 \preccurlyeq q_2$.
- Assume if $n = k$, there exists $q_1, \cdots, q_k$ such that for all $1 \leq i < k$, $q_i \rightarrow \cdots \rightarrow q_{i+1}$ and $q'_1 \preccurlyeq q_i$ for all $1 \leq i \leq k$. Since $q'_1 \preccurlyeq q_k$ and $q'_1 \Rightarrow \cdots \Rightarrow q'_1$ in $\mathcal{L}'$, by applying Lemma 4, there exists a path $q_k \rightarrow \cdots \rightarrow r$ such that $q'_1 \preccurlyeq r$. Thus, we can select $q_{k+1} = r$, and there exists $q_1, \cdots, q_{k+1}$ such that for all $1 \leq i < k + 1$, $q_i \rightarrow \cdots \rightarrow q_{i+1}$ and $q'_1 \preccurlyeq q_i$ for all $1 \leq i \leq k + 1$.

By induction, there are infinite states $q_1, q_2, \cdots$ such that $q_i \rightarrow \cdots \rightarrow q_{i+1}$ in $\mathcal{L}$. Since the number of states in $\mathcal{L}$ is finite, there exists $q_j = q_1$ where $j > 1$. Therefore, there exists a cycle $q_1 \rightarrow \cdots \rightarrow q_1$ in $\mathcal{L}$. Moreover, since the simulation relation is compatible with the set of Büchi conditions J, the fact that the cycle $q'_1 \Rightarrow \cdots \Rightarrow q'_1$ in $\mathcal{L}'$ contains a J_i-state for all $J_i \in J$ implies that for all $1 \leq i < j$, the path $q_i \rightarrow \cdots \rightarrow q_{i+1}$ also contains a J_i-state for all $J_i \in J$. Thus, the cycle $q_1 \rightarrow \cdots \rightarrow q_1$ also contains a J_i-state for all $J_i \in J$. Therefore, there exists an accepting cycle $q_1 \rightarrow \cdots \rightarrow q_1$ in $\mathcal{L}$. □

Lemma 6. *If there exists an accepting run in $\mathcal{L}'$, there exists an accepting run in $\mathcal{L}$.*

Proof: Since the number of states in $\mathcal{L}'$ is finite, the accepting run in $\mathcal{L}'$ has a lasso form $q'_0 \Rightarrow \cdots \Rightarrow q'_m \Rightarrow (r'_0 \Rightarrow \cdots \Rightarrow r'_n)^w$ where $q'_0 \in init'_c$. By applying Lemma 4 for the path $q'_0 \Rightarrow \cdots \Rightarrow q'_m \Rightarrow r'_0$, there exists a path $q_0 \rightarrow \cdots \rightarrow q_m \rightarrow r_0$ in $\mathcal{L}$ where $q_0 \in init_c$. Then, by applying Lemma 5 for the cycle $r'_0 \Rightarrow \cdots \Rightarrow r'_n \Rightarrow r'_0$, there exists a cycle $r_0 \rightarrow \cdots \rightarrow r_k \rightarrow r_0$ in $\mathcal{L}$. Moreover, this cycle also contains a J_i-state for all $J_i \in J$. Thus, there exists an accepting run $q_0 \rightarrow \cdots \rightarrow q_m \rightarrow (r_0 \rightarrow \cdots \rightarrow r_k)^w$ in $\mathcal{L}$. □

Theorem 4. *Given a transition system $\mathcal{L}$, a set of Büchi conditions J, and a simulation relation $\preccurlyeq$ over states of $\mathcal{L}$, $\mathcal{L}$ has an accepting run if and only if $\mathcal{L}'$ has an accepting run.*

Proof: Clearly if there exists an accepting run in $\mathcal{L}$, this run also exists in $\mathcal{L}'$. In the other direction, if there exists an accepting run in $\mathcal{L}'$, by Lemma 6, there exists an accepting run in $\mathcal{L}$. Thus, the theorem is proved. □

Theorem 5. *Algorithm 4 is sound and complete.*

Proof: As we discussed the difference between Algorithm 4 and Algorithm 3, given a transition system $\mathcal{L}$ with a set of initial states $Init$, the transition relation Tr and a set of Büchi conditions J, while $IsEmpty(Init, Tr, J)$ is checking the emptiness of $\mathcal{L}$, $IsEmpty_{sim}(Init, Tr, J)$ is actually checking the emptiness of the transition system $\mathcal{L}'$. Thus, the correctness of Algorithm 4 is obtained based on Theorem 4. □

Lemma 7. *If $new' = Down(new)$, then $Reach_{sim}(new' \cap J_i) = Down(Reach(new \cap J_i))$ for any Büchi condition $J_i \in J$.*

Proof: Note that $J_i = Down(J_i)$, we prove that $new' \cap J_i = Down(new \cap J_i)$ and then apply Lemma 2. We have $Down(new \cap J_i) \subseteq Down(new) \cap Down(J_i) = new' \cap J_i$. Then, we prove $new' \cap J_i \subseteq Down(new \cap J_i)$. Given any $s' \in new' \cap J_i$, there exists $s \in new$ such that $s' \preccurlyeq s$. Moreover, $s' \in J_i$ implies $s \in J_i$. So $s \in new \cap J_i$. Therefore, $s' \in Down(new \cap J_i)$. □

Lemma 8. *If $new' = Down(new)$, $Down(new \cap succ(new)) \subseteq new' \cap Down(succ(new'))$*

Proof: We have $Down(new \cap succ(new)) \subseteq Down(new) \cap Down(succ(new)) = new' \cap Down(succ(new)) = new' \cap Down(succ(new'))$. □