Masaryk University
Faculty of Informatics

Untimed LTL Model Checking of Timed Automata

Master's thesis

Jan Havlíček

Brno, 2013

Declaration

Hereby I declare, that this thesis is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: doc. RNDr. Jiří Barnat, Ph.D

Acknowledgement

I would like to thank to all membes of the ParaDiSe laboratory and everybody else who participated on DiVinE development. Namely, I would like to thank to doc. RNDr. Jiří Barnat, Ph.D for supervising this thesis and consultations, and to Vojtěch Havel for his contributions to the implementation of the timed automata interpreter.

Abstract

This thesis presents a new option to verify timed automata that has been implemented into the parallel model checker DiVinE. This allows DiVinE to verify LTL properties on models of real-time systems designed by Uppaal. We overview abstractions and reduction techniques commonly used in verification of timed automata and their extensions and discuss their usability for LTL model checking. We also perform experimental evaluation of our implementation, compare it to Uppaal and measure how much can it be sped-up by using multiple threads.

Keywords

Timed Automata, Model Checking, Verification, Parallel, DiVinE, Uppaal, LTL

Contents

1 Introduction

As the complexity of hardware and software systems is steadily increasing, it is getting harder to ensure their quality. Testing cannot guarantee system correctness, especially with parallelism. For this reason, formal verification has become a part of the development process for industries where ensuring correctness is critical, such as chip manufacturing.

One of the well-known formal verification methods is model checking. Its idea is to prove that the system under verification has the specified property by traversing all its configurations reachable from the initial one. However, the number of configurations can be exponential in the size of the system specification and, for many complex systems, it may be infeasible to fully explore all of them. This phenomenon is called the state space explosion and several techniques have been researched with the aim to reduce its impact. For example, abstracting away some information can decrease the number of states, but can also lead to incorrect results under some circumstances. As a different approach, some of the modern model checkers, as well as many other pieces of computationally intensive software, try to exploit the architecture of current hardware by running the computation on multiple threads or even distributing it over the network.

In order to formally analyse or verify a system, it has to be described in a suitable formalism that can fully capture all its important properties. The behaviour of some systems or their correctness may depend on time. For example, they can be time-critical in a sense that their response has to come in a given time-period, aside from being correct. For this kind of systems, timed automata are widely used as the modeling formalism, because they allow to specify a fixed number of clocks to measure the elapsed time, query their values and reset them independently. Since clocks are defined as real-valuated, strict interpretation of timed automata semantics leads to an infinite system. However, by allowing clocks to be compared only with integral values, many states cannot be distinguished by such a comparison and a finite system with the same behaviour can be constructed. This construction is known as the region abstraction. Despite the unique theoretic properties of the region abstraction, it is rarely used in practical applications, since the number of regions depends on the value of constants used in guards. Coarser abstractions based on zones are used instead.

Not only the system itself, but also the properties to be verified have to be formally specified. The linear temporal logic (LTL) is one of the commonly used ways to formulate claims about system behaviour. LTL can express claims that should hold in every state (assertions), various liveness properties (responsiveness), fairness constraints and their arbitrary combinations, while still being efficiently decidable.

The aim of this thesis is to overview abstractions and other reduction techniques commonly used in timed automata verification, determine the possibility of their use for LTL model checking of timed automata and to implement an extension to the model checker DiVinE allowing to verify real-time systems with respect to LTL properties. The verification can be run on multiple threads or even in a distributed environment to take advantage of the current hardware architecture. We measure the impact of different zone-based abstractions, compare the performance of our implementation to Uppaal and also evaluate how much can the verification be sped-up by increasing the number of parallel workers.

The support of full LTL allows us to verify properties that are beyond capabilities of Uppaal [1], which can be considered an industry standard for verification of real-time systems, or parallel reachability-focused verification tools, like LTSmin with `opaal` [2]. And even though there already are some tools allowing LTL model checking of timed automata, like [3] and [4], they are either discontinued, publicly unavailable or prototype implementations. On the other hand, our implementation is incorporated into an existing open-source LTL model checker DiVinE [5, 6] that can already be used to verify models specified using a variety of input languages and a subset of C/C++ programs. To our knowledge, that makes DiVinE the first parallel and distributed LTL model checker for timed automata and also the first tool allowing LTL model checking of timed automata containing clock difference constraints.

This thesis is organized as follows. Chapter 2 formally defines timed automata, outlines some of their extensions and the Uppaal modeling language and overviews commonly used abstractions and reduction techniques that can be used to simplify the verification of timed automata. The linear temporal logic (LTL) and its timed extension (TLTL) is presented in Chapter 3 together with techniques and algorithms designed to perform the LTL model checking. Chapter 4 aims to describe our implementation and Chapter 5 provides an evaluation of its performance. The thesis is concluded in Chapter 6.

2 Timed automata

This chapter defines some of the notions regarding timed automata and overviews algorithms, abstractions and other techniques applicable to timed automata with the focus on basics and techniques that are actually used in our implemenatation. We refer to [7] and [8] for more extensive and detailed surveys on this topic.

2.1 Syntax

A timed automaton is essentially a finite transition system with finitely many non-negative real-valuated variables, called clocks, whose values can be tested and changed when traversing edges. Formally, we will define a timed automaton as a tuple $\langle L, X, l_I, E, Inv \rangle$, where:

- L is the finite set of locations,
- X is the finite set of clocks,
- $l_I \in L$ is the initial location,
- $E \subseteq L \times G(X) \times 2^X \times L$ is the set of edges labelled by guards and sets of clocks to be reset,
- $Inv : L \to G(X)$ is the mapping that assigns an invariant to every location.

$G(X)$ denotes the set of all possible clock constraints that are defined inductively as:

$$g ::= x_1 \lesseqgtr n \mid x_1 - x_2 \lesseqgtr n \mid g_1 \wedge g_2 \mid true$$

where

$$\begin{aligned} x_1, x_2 &\in X, \\ n &\in \mathbb{Z}, \\ \lesseqgtr &\in \{<, \leq, \geq, >\}. \end{aligned}$$

Constrains involving clock differences are called *clock difference constraints* or *diagonal constraints*. An automaton that does not contain any clock difference constraints is called *diagonal-free*. Section 2.5.5 focuses on clock difference constraints and their pitfalls. The purpose of invariants is to limit how long can the system stay in a specific location, so only constraints of the form $x_1 < n$ and $x_1 \leq n$ are usually allowed to appear in invariants. An example of a simple timed automaton with guards, a clock reset and an invariant can be seen on Figure 2.2.

For the purpose of this thesis, we do not consider timed automata to be language acceptors, because we use them solely to model behaviour of a specific system. However, it is possible to label edges by symbols from an input alphabet and add an accepting condition for finite or infinite words. A *timed word* accepted by such automaton is a sequence of pairs (a_i, t_i), where a_i is an input symbol and $t_i \in \mathbb{R}_+$ is a time in which the symbol was read since the automaton was started — the so-called time-stamp ($\mathbb{R}_+$ is a set of non-negative real numbers). Stripping all the time-stamps gives us an *untimed word*. It is a PSPACE-complete problem to decide if a timed automaton accepts non-empty language. We refer to [9] for detailed description of languages accepted by timed automata and their properties.

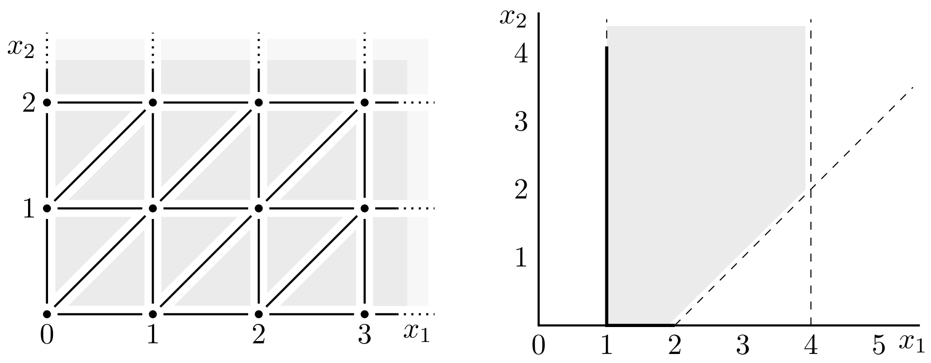
2.2 Semantics

A clock *valuation* is a function $\nu : X \to \mathbb{R}_+$. Let ν be a clock valuation, $\delta \in \mathbb{R}_+$ and $Y \subseteq X$. We will define $\nu + \delta$ to be a valuation for which $(\nu + \delta)(x) = \nu(x) + \delta$ for every clock $x \in X$, $\nu[Y := 0]$ to be the valuation for which $\nu[Y := 0](x) = 0$ if $x \in Y$ and $\nu[Y := 0](x) = \nu(x)$ otherwise. And finally, $\nu_{\vec{0}}$ will denote the valuation such that $\nu_{\vec{0}}(x) = 0$ for every clock x.

The semantics of a timed automaton $\mathcal{A} = \langle L, X, l_I, E, Inv \rangle$ can be defined by the transition system $\mathcal{TS}(\mathcal{A}) = \langle S, s_I, \longrightarrow \rangle$ where:

- $S = L \times \mathbb{R}_+^X$ is the set of states (configurations of $\mathcal{A}$),
- $s_I = (l_I, \nu_{\vec{0}})$ is the initial state,
- there is an action transition $(l_1, \nu_1) \longrightarrow (l_2, \nu_2)$ if and only if there exists an edge $(l_1, g, R, l_2) \in E$ such that $\nu_1 \models g$, $\nu_2 = \nu_1[R := 0]$ and $\nu_2 \models Inv(l_2)$,
- there is a time transition $(l, \nu_1) \longrightarrow (l, \nu_2)$ if there is $\delta \in \mathbb{R}_+$ so that $\nu_2 = \nu_1 + \delta$ and $\nu_2 \models Inv(l)$.

However, this transition system can be infinite, so another way to express the semantics is necessary. It has been observed that many valuations cannot be distinguished by any guard. To put it formally, let $M(x)$ denote the highest constant that the clock x is compared to, $\lfloor r \rfloor$ and $fr(r)$ denote integral and



Visualization of regions
for $M(x_1) = 3$ and $M(x_2) = 2$

Zone induced by constraints
$x_1 \geq 1 \land x_1 < 4 \land x_2 \geq 0 \land x_1 - x_2 < 2$

Figure 2.1: Regions and zones.

fractional part of r. Then we can define valuations ν_1 and ν_2 to be *region equivalent* if all three following points hold for all clocks $x, y \in X$:

1. $\lfloor \nu_1(x) \rfloor = \lfloor \nu_2(x) \rfloor \lor (\nu_1(x) > M(x) \land \nu_2(x) > M(x))$
2. $\nu_1(x) \leq M(x) \implies (fr(\nu_1(x)) = 0 \iff fr(\nu_2(x)) = 0)$
3. $(\nu_1(x) \leq M(x) \land \nu_1(y) \leq M(y)) \implies (fr(\nu_1(x)) < fr(\nu_2(x)) \iff fr(\nu_1(y)) < fr(\nu_2(y)))$

According to this definition, a symbolic representation of the state space that uses sets of equivalent valuations (regions) can be constructed. Such representation is called *region graph* and each its vertex corresponds to one or infinitely many vertices in $\mathcal{TS}(\mathcal{A})$ it simulates. Even though the number of regions is finite, it is exponential in the number of clocks and linear in $\prod_{x \in X} M(x)$, which means the number of regions is likely to be very high unless the number of clocks and their bounds are fairly low. Many proofs and theoretic constructions concerning timed automata are based on regions, but most tools use *zones* instead. Figure 2.1 shows an example of a region graph and a zone. All points, line segments and areas between them are separate regions.

2.3 Zones

The construction that is used in practice to represent the state space of a timed automaton is based on the so-called *zones*. A zone is a (convex) set of valuations expressible by constraints from $G(X)$. Let us denote the set of all zones as $Zones(X)$ and if Z is a zone and $Y \subseteq X$, we will define the "up" operation $Z^{\uparrow} \overset{def}{=} \{\nu + \delta \mid \nu \in Z\}$, the restriction of a zone by a clock constraint $Z \barwedge g \overset{def}{=} \{\nu \mid \nu \in Z, \nu \models g\}$ and the clock reset $Z[Y := 0] \overset{def}{=} \{\nu[Y := 0] \mid \nu \in Z\}$.

The *zone graph* of a timed automaton $\mathcal{A} = \langle L, X, l_I, E, Inv \rangle$ is defined to be a transition system $\mathcal{ZG}(\mathcal{A}) = \langle S, s_I, \Longrightarrow \rangle$ where:

- $S = L \times Zones(X)$ is the set of symbolic states,
- $s_I = (l_I, \{\nu_{\vec{0}}\})$ is the initial state,
- $(l_1, Z_1) \Longrightarrow (l_2, Z_2)$ if there is an edge $(l_1, g, R, l_2) \in E$ for which it holds that $(Z_1 \barwedge g)[Y := 0]^{\uparrow} \barwedge Inv(l_2) = Z_2$ and $Z_2 \neq \emptyset$.

Zones can be efficiently stored as matrices of constraints called *difference bound matrices* (DBM). DBM is defined to be a square matrix of pairs $D = \langle c_{ij}, \prec_{ij} \rangle_{i<n, j<n}$ with $n = |X| + 1$ rows and columns, where either $c_{ij} \in \mathbb{Z}$ and $\prec_{ij} \in \{<, \leq\}$ or $c_{ij} = \infty$ and $\prec_{ij} = <$. Such matrix D defines a zone $Z_D = \{\nu \mid \forall 0 \leq i, j < n : \nu(x_i) - \nu(x_j) \prec_{ij} c_{ij}\}$ where $x_1, \ldots, x_{n-1}$ are all clocks and $\nu(x_0) = 0$ for any valuation.

DBMs also have an important property that they can be transformed into canonical forms. This helps immensely in model checking since it allows for simple equality testing and use of hash-tables. A DBM is in a canonical form if $\forall i, j, k < n : (c_{ij}, \prec_{ij}) \leq (c_{ik}, \prec_{ik}) + (c_{kj}, \prec_{kj})$ assuming the addition and comparison of DBM elements is defined in a following way:

$$(c_1, \prec_1) + (c_2, \prec_2) = \begin{cases} (\infty, <) & \text{if } c_1 = \infty \text{ or } c_2 = \infty \\ (c_1 + c_2, \leq) & \text{if } \prec_1 = \prec_2 = \leq \\ (c_1 + c_2, <) & \text{otherwise} \end{cases}$$

$$\begin{aligned} (c_1, \prec_1) \leq (c_2, \prec_2) \quad & \text{if } c_1 < c_2 \text{ or the following case is true} \\ (c, \prec_1) \leq (c, \prec_2) \quad & \text{if } \prec_1 = \prec_2 \text{ or } \prec_2 = \leq \end{aligned}$$

The transformation to the canonical form (also called DBM closing) can be done in $\mathcal{O}(n^3)$ time using the Floyd-Warshall's algorithm. See [7] for description of other operations on difference bound matrices, like conjunction and the "up" operation.

The zone graph can still be infinite and an additional methods, called extrapolations or normalizations, have to be employed to address this issue. Section 2.5 describes these techniques in detail.

```
Input: Source state $(l, Z)$
federation $F_{cur}$, $F_{next}$, $F_{src}$
$F_{next} \leftarrow \{Z\}$
foreach priority $p$ in descending order do
    $F_{cur} \leftarrow F_{next}$
    foreach edge $e$ with guard $g$ and priority $p$ from $l$ do
        $F_{src} \leftarrow F_{cur} \barwedge g$
        foreach zone $Z \in F_{src}$ do
            $generateSuccessors(l, Z, e)$
        end
        // Perform federation subtraction
        $F_{next} \leftarrow F_{next} - F_{src}$
    end
end
if $F_{next} \neq \emptyset$ then
    possible time-lock
end
```

Algorithm 2.1: Successor generation with priorities.

2.4 Extensions

2.4.1 Non-zero updates

In the classical timed automata formalism, the only possible update of a clock is the reset to zero ($x := 0$). Allowing different kinds of updates affects the expressive power (from the language point of view) and even decidability of reachability properties [10]. For example:

- Updates of the form $x := c$, $c \in \mathbb{N}_0$ do not increase expressiveness, but can lead to exponentially more concise automata [11]. Updates of this form are supported both by Uppaal and our implementation.
- Updates of the form $x := x + 1$ or $x_1 := x_2 + c$, $c \in \mathbb{N}_0$ do not increase expressiveness of diagonal-free automata, but can lead to undecidability if used in conjunction with diagonal constraints.
- Updates of the form $x := x - 1$ always lead to undecidable formalism.

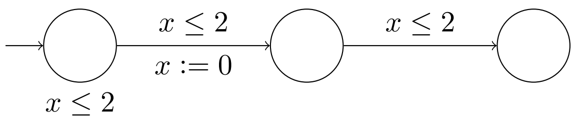


Figure 2.2: An example of a timed automaton.

2.4.2 Networks of timed automata

It is impractical to model systems consisting of multiple independent components as a single timed automaton and *network of timed automata* is a more suitable formalism since it is easily composable and scalable. Networks are usually defined by the means of the parallel composition operator ($\|$) from process algebra [12] and some form of inter-process communication.

For example, Uppaal allows binary synchronization on arbitrary edges, n-ary synchronization with some restrictions and it is possible to pass values during synchronization using an auxiliary variable. Moreover, it is also possible to assign priorities to communication channels and processes to reduce non-determinism. The construction of a product automaton from the composition is usually done on-the-fly.

However, priorities may cause that the set of valuations for which certain action can be performed is not expressible as a zone. For example, if an edge with a high priority has a guard $x \geq 1 \land x \leq 2$, then some low priority edge with no guard can be used only if $x \in [0, 1) \cup (2, \infty)$. The general solution for successor generation with priorities is based on zone subtractions [13] and the algorithm itself is outlined as Algorithm 2.1. This algorithm needs to work with arbitrary sets of valuations, which are usually represented as unions of zones called DBM federations. Most of the DBM operations can be extended to federations simply by performing them on all zones in the union. However, the subtraction algorithm is more complicated and needs $\mathcal{O}(m \cdot n)$ operations, where m and n are numbers of zones in each fedration.

2.5 Abstractions and reductions

2.5.1 Location-based simulation

This section introduces the notion of location-based simulation and bisimulation, which is crucial for definitions of various abstractions and their properties. Intuitively, these kinds of relations are analogous to well-known simulation and bisimulation relations, but they focus solely on time, since only states with equal locations can be related.

Let $\mathcal{A} = \langle L, X, l_I, E, Inv \rangle$ be a timed automaton and $\preccurlyeq$ be a a binary relation on $L \times \mathbb{R}_+^X$. We say $\preccurlyeq$ is a *location-based simulation* (or *time-abstracting simulation*) if it satisfies following properties:

1. if $(l_1, \nu_1) \preccurlyeq (l_2, \nu_2)$, then $l_1 = l_2$
2. if $(l, \nu_1) \preccurlyeq (l, \nu_2)$ and $(l, \nu_1) \longrightarrow (l', \nu_1')$, then there is (l', ν_2') so that $(l, \nu_2) \longrightarrow (l', \nu_2')$ and $(l', \nu_1') \preccurlyeq (l', \nu_2')$
3. if $(l, \nu_1) \preccurlyeq (l, \nu_2)$ and $(l, \nu_1) \longrightarrow (l', \nu_1 + \delta)$, then there is $\delta' \in \mathbb{R}_+$ so that $(l, \nu_2) \longrightarrow (l', \nu_2 + \delta')$ and $(l', \nu_1 + \delta) \preccurlyeq (l', \nu_2 + \delta')$

If both $\preccurlyeq$ and $\preccurlyeq^{-1}$ are location-based simulations, then $\preccurlyeq$ is location-based bisimulation. [14]

Let $\preccurlyeq$ be a location-based simulation, then we can define an *abstraction* in the context of timed systems to be a mapping $\mathfrak{a} : L \times 2^{\mathbb{R}_+^X} \to L \times 2^{\mathbb{R}_+^X}$ satisfying that whenever $(l_2, Z_2) = \mathfrak{a}((l_1, Z_1))$, then $l_1 = l_2$, $Z_1 \subseteq Z_2$, and for each $\nu_2 \in Z_2$, there is $\nu_1 \in Z_1$ so that $(l_2, \nu_2) \preccurlyeq (l_1, \nu_1)$.

If the abstraction $\mathfrak{a}$ returns a zone for every zone on the input, so it can be restricted to a mapping $\mathfrak{a} : L \times Zones \to L \times Zones$, we will call it a zone-based abstraction. All abstractions described in following subsections are zone-based. If the location part of states is fixed or unimportant, we will consider an abstraction $\mathfrak{a}$ to be mapping from the set of zones to itself.

2.5.2 Zone extrapolation

As was mentioned in the previous section, the region abstraction is rarely used in practice, since the number of regions can become needlessly high even for relatively small system and for this reason, most of the tools working with timed automata use zones represented by difference bound matrices (DBMs).

The number of zones produced by any zone-base abstraction is always countable, but it can still be infinite. Similarly to the region abstraction, this issue can be dealt with by exploiting the fact that any timed automata is unable to distinguish values of clocks after they exceed a certain bound. Let $M(x)$ denote the maximal constant involved in any invariant or guard with the clock x. Then any zone can be transformed by removing constraints $x \preceq c$ and $x - x' \preceq c$ for $\preceq \in \{<, \leq\}$ in case that $c > M(x)$ and replacing the right side of the constraints $x \succeq c$ and $x - x' \succeq c$ by $M(x)$ if $c > M(x)$. This transformation is called *maximal bounds extrapolation* and it is the baseline for more advanced extrapolation techniques used in model checkers [7].

Formally, it is based on the following relation:

$$\nu \equiv_M \nu' \overset{def}{\iff} \forall x \in X : \nu(x) = \nu'(x) \vee (\nu(x) > M(x) \wedge \nu'(x) > M(x))$$

For every zone W, we can express the extrapolated zone as $\mathfrak{a}_{\equiv_M}(W) = \{\nu' \mid \nu \in W, \nu' \equiv_M \nu\}$. The relation $\mathcal{R} = \{((l,\nu),(l,\nu')) \mid \nu \equiv_M \nu'\}$ is a location-based bisimulation, which means it preserves all important properties of the state space, such as reachability and presence of deadlocks and so-called time-locks — states, where the time can pass, but no action transition will ever become enabled.

Since DBMs are the most common way to store zones, we will describe extrapolations as transformations from DBM $\langle(c_{ij}, \prec_{ij})\rangle_{i,j=0,\dots,n}$ to another DBM $\langle(c'_{ij}, \prec'_{ij})\rangle_{i,j=0,\dots,n}$. For the maximal bounds extrapolation, the rule is following:

$$(c'_{ij}, \prec'_{ij}) = \begin{cases} \infty & \text{if } c_{ij} > M(x_i) \\ (-M(x_j), <) & \text{if } -c_{ij} > M(x_j) \\ (c_{ij}, \prec_{ij}) & \text{otherwise} \end{cases} \qquad \mathit{Extra}_M$$

2.5.3 LU extrapolation

Several extensions of the maximal bounds extrapolation were introduced in [14]. The LU extrapolation takes advantage of the fact that constants the clocks are compared to from above and below often differ.

Let us define the lower bound $L(x)$ to be the highest value the clock x is compared to from below (comparisons of the form $x > c$ or $x - x' > c$) and similarly for the upper bound $H(x)$. Naturally, it holds that $M(x) = \max(L(x), U(x))$. The *LU extrapolation* can be defined in the form of DBM transformation in the following way:

$$(c'_{ij}, \prec'_{ij}) = \begin{cases} \infty & \text{if } c_{ij} > L(x_i) \\ (-U(x_j), <) & \text{if } -c_{ij} > U(x_j) \\ (c_{ij}, \prec_{ij}) & \text{otherwise} \end{cases} \qquad \mathit{Extra}_{LU}$$

Another observation described in [14] is that if a whole zone is above the maximal bound for some clock x, diagonal constraints involving x can be removed without a change in the system behaviour. This of course requires the timed automaton to be diagonal-free, but clock difference constraints are problematic even with the maximal bounds extrapolation — see Section 2.5.5 for details. Diagonal variants of both the maximal bounds extrapolation and the LU extrapolation that use this rule are defined below.

$$(c'_{ij}, \prec'_{ij}) = \begin{cases} \infty & \text{if } c_{ij} > M(x_i) \\ \infty & \text{if } -c_{0i} > M(x_i) \\ \infty & \text{if } -c_{0j} > M(x_j),\ i \neq 0 \\ (-M(x_j), <) & \text{if } -c_{ij} > M(x_j),\ i = 0 \\ (c_{ij}, \prec_{ij}) & \text{otherwise} \end{cases} \qquad \mathit{Extra}^{+}_{M}$$

$$(c'_{ij}, \prec'_{ij}) = \begin{cases} \infty & \text{if } c_{ij} > L(x_i) \\ \infty & \text{if } -c_{0i} > L(x_i) \\ \infty & \text{if } -c_{0j} > U(x_j),\ i \neq 0 \\ (-U(x_j), <) & \text{if } -c_{ij} > U(x_j),\ i = 0 \\ (c_{ij}, \prec_{ij}) & \text{otherwise} \end{cases} \qquad \mathit{Extra}^{+}_{LU}$$

In the same way as the maximal bounds extrapolation is based on the relation $\equiv_M$, the LU extrapolation is based on the so-called *LU preorder* $\preccurlyeq_{LU}$. It holds that $\nu \preccurlyeq_{LU} \nu'$ if the following is true for all clocks x:

- either $\nu(x) = \nu'(x)$
- or $L(x) < \nu(x) < \nu'(x)$
- or $\nu(x) > \nu'(x) > U(x)$

However, we cannot simply construct an abstraction from the LU preorder, because such abstraction can produce sets of valuations that cannot be expressed as zones. The LU extrapolation obviously always produces zones, but it is not as coarse as the abstraction constructed from the LU preorder would be.

Let us note that the LU extrapolation has a few disadvantages over the classical maximal bounds extrapolation. First of all, it is not an exact abstraction with respect to bisimulation, which means it may not preserve all properties of the state space. Namely, it does not preserve so-called time-locks — states where time can pass, but no action transition can be performed in future. Uppaal considers time-locks to be special cases of deadlocks, our implementation can be run with time-lock detection enabled or disabled.

Let us demonstrate the problem of time-lock detection under the LU extrapolation on Figure 2.2. A time-lock can happen in the central location, since there is no outgoing edge for $x \in (2, \infty)$, but not in the initial one, since it is prevented by the location invariant. The lower bound for x is $-\infty$ and the upper bound is 2. The initial state will be entered with a zone, where $x = 0$, which will be expanded using the "up" operation to a zone $[x \geq 0 \wedge x \leq 2]$. However, since $2 < L(x)$, the extrapolation $Extra_{LU}$ will transform it into a zone $[x \geq 0]$, which is the same zone that would be produced if there was no invariant. This means that any time-lock detection method would falsely find a time-lock in the initial state if the LU extrapolation is used.

The second drawback of the LU extrapolation is that it makes Zeno runs more difficult to detect [15]. Under the LU extrapolation it is $\mathcal{NP}$-hard to decide if an abstract run can be instantiated to at least one non-Zeno run. Zeno runs are described in more detail in Section 3.3.

2.5.4 Location-dependent bounds

Extrapolations described so far use global bounds for individual clocks, but it is also possible to compute separate bound for each location. This technique, often called *location-dependent* abstraction, was introduced in [16] and it surpassed previously used technique — the *active clock reduction*.

With the active clock reduction, the set of clocks that can be accessed before their value is reset (Act_l) is constructed for each location l. The system behaviour clearly does not depend on the values of clocks outside this set, which means they can be abstracted-out (set to a fixed value) to potentially reduce the number of different zones.

Formally, we can define Act_l as the smallest set satisfying following:

- $x \in Act_l$ if x is referenced in $Inv(l)$ or a guard g for some edge $(l, g, R, l') \in E$
- $(Act_{l'} \setminus R) \subseteq Act_l$ for any edge $(l, g, R, l') \in E$ that resets clocks in R

This set can be easily computed using a backwards reachability on the timed automaton in question and for a network of timed automata, the set of active clocks can be computed on-the-fly as the union of active clocks for each of its components.

The local bounds for each location can be computed in a similar way. The local maximal bound $M_l(x)$ is the lowest number from $\mathbb{Z} \cup \{-\infty\}$ satisfying:

- $M_l(x) \geq c$ if $x \lesseqgtr c$ is a part of $Inv(l)$ or g for some $(l, g, R, l') \in E$
- $M_l(x) \geq M_{l'}(x)$ if there is an edge $(l, g, R, l') \in E$ and $x \notin R$

The local lower and upper bounds can be expressed in the same way — we just need to take into account only constraints from below or above in the first bullet.

It was shown that location dependent abstraction subsumes the active clock reduction, because if $x \notin Act_l$, then $M_l(x) = -\infty$ and all possible values of x in l will be above $M_l(x)$ and thus considered equal by the relation $\equiv_M$.

If the LTL verification is being performed and the formula contains a clock constraint, we potentially need to decide the validity of such constraint and its negation in every state. Therefore, we implemented a rule that prevents the lower and upper local bounds for any clock x to be set below c if the LTL formula being verified contains a comparison of the clock x to the constant c.

2.5.5 Diagonal constraints

Extrapolations described to this point did not consider diagonal constraints. The reason is that using even the basic maximal bounds extrapolation for a timed automaton with diagonal constraints can lead to incorrect reachability results if more than three clocks are used [17]. Other reason why diagonal constraints are often excluded from constructions involving timed automata is that they do not add any expressive power, so that an equivalent diagonal-free automaton always exists. However, their use can significantly simplify the model construction in some cases and they are very useful when modeling scheduling problems.

Even though an equivalent automaton without diagonal constraints can always be constructed, the removal of every diagonal constraint can double the automaton size, resulting in an exponentially larger automaton [18]. This blow-up generally cannot be avoided, since automata with diagonal constraints allow for exponentially more concise representation of some behaviour than diagonal-free automata, as was shown in [11].

Other methods are based either on encoding truth-values of each diagonal constraints into states of the zone-graph or on zone slicing. We decided to employ a method based on [19], which modifies the extrapolation procedure so that the set of diagonal constraints that hold in each zone is not changed by the extrapolation. This is done by slicing the zone by all diagonal constraints used in the automaton, extrapolating its parts separately and ensuring the result is contained in the corresponding slice. Formally, if G_{diag} is the set of all diagonal constraints and $Extra$ is an extrapolation operation, then the new extrapolation operation that guarantees correctness even with the presence of diagonal constraints can be defined as:

$$Extra_{diff}(Z) = \{Extra(Z \,\overline{\wedge}\, s) \,\overline{\wedge}\, s \mid s \in Slices \wedge (Z \,\overline{\wedge}\, s) \neq \emptyset\}$$

where

$$Slices = \left\{ \bigwedge_{g \in A} g \wedge \bigwedge_{g \in G_{diag} \setminus A} \neg g \;\middle|\; A \subseteq G_{diag} \right\}.$$

The advantage of this method is that it works on-the-fly and does not cause any overhead if no diagonal constraints are used.

Even though the original proposal used the maximal bounds extrapolation, we made it to work with the LU extrapolation by setting the lower and upper bounds of clocks x_1 and x_2 to at least $|c|$ if there is a diagonal constraint $x_1 - x_2 \lesseqgtr c$ either in the model or the LTL property. We found no other complications that would prevent combining LTL verification with diagonal constraints.

As a different approach to diagonal constraints, [20] proposed to use the (incorrect) original algorithm and then analyse whether the traces it returns are feasible or not. The advantage is that the potentially exponential blow-up of the whole state space is avoided. However, this method was aimed only on reachability analysis and it is not suitable for LTL model checking.

2.5.6 Subsumption

Since zones are sets of valuations, states with equal locations can be naturally ordered with respect to inclusion of their zones. If $Z \subseteq Z'$ we say that state (l, Z') subsumes (l, Z) and since (l, Z') simulates (l, Z), we do not need to process (l, Z) if we already processed (l, Z'). On the other hand, if we visit (l, Z') after (l, Z), we can save some memory by storing only the subsuming state, but we do not save any time if we already generated successors of the subsumed one. This means that the actual savings brought by subsumption highly depend on the search order, as reported by [2].

Furthermore, subsumption is meant to be used for reachability analysis and it would be very difficult or impossible to use it for LTL verification. The reason is that the only information we need to store during the reachability analysis is whether the given state was visited or not and if we visit a state, marking all its subsumed states as visited preserves correctness of the reachability algorithm. However, this is not necessarily true for complex additional information that needs to be stored for every state by algorithms performing LTL model checking, like OWCTY or MAP.

2.6 Uppaal

In this section, we overview the syntax and semantics of the Uppaal [1] modeling language with focus on topics not covered in the official documentation[1] or otherwise unclear or non-intuitive.

2.6.1 Type system and variables

Uppaal offers three basic variable types. The integral type, `int`, has a default range of $\langle -32768, 32767\rangle$, which can be modified to any range inside $\langle -2^{31}, 2^{31}-1\rangle$ using rectangular brackets (`int[-1,10] x;`). Any attempt to assign a value outside the specified range causes the verification performed by Uppaal to terminate, but any intermediate computations are done on signed 32-bit integers without any range checks. If a variable is declared, but not explicitly initialized, its initial value depends on its range. If the range contains zero, the value is set to zero. Otherwise, it is set to the minimal value in the range.

The boolean type, `bool`, is essentially equivalent to `int[0,1]`, which means all arithmetical operations can be performed on booleans and all logical operators also work on integers. In this situation, any non-zero value is considered to be true and `true` evaluates to 1 so that `2*true == 2`. The `scalar` type is designed solely to allow symmetry reduction and does not support any operation other that check for equality with another scalar.

Additional types can be defined using the `typedef` keyword or creating structures or arrays from existing types. Constant variables can be declared by prepending `const` to the type, allowing these variables to be used in range specifications and array sizes. Few examples:

```
struct R { int min; int max; };
const R r = {-1, 10};
typedef int[r.min, r.max] myint;
myint arr[2][3] = {{0,1,2}, {3,4,5}};
```

Structures and arrays can be assigned as a whole using single assignment if their types and dimensions are compatible.

Rules regarding identifier naming and most of the arithmetic operations are the same as in C and C++ programming languages. This includes bit-wise operators (`&`, `|`, `<<`, `>>`, `^`, but not `~`), modulo (`%`), pre-increment and post-increments operators, shortcut operators (`+=`, `*=`, `<<=`, ...) and the ternary

1. Available on-line at `http://www.it.uu.se/research/group/darts/uppaal/help.php?file=System_Descriptions/Introduction.shtml`

operator. Assignments can be done using both `:=` and `=`, all C-like relational operators are supported (`==`, `!=`, `<=`, `<`, ...) and the standard set of logical operations (`!`, `||`, `&&`) is supplemented by `not`, `or`, `and` (and `imply`) with the same meaning, but lower precedence, which allows to exploit the short-circuit behaviour in expressions like `a = a/2 or b *= 2`, which doubles `b` if the new value of `a` is zero. Also, comma can be used to join multiple statements into one.

Other newly introduced operators are the minimum and maximum operators (the result of `3 <? 2` is 2 and `3 >? 2` evaluates to 3), and the universal and existential quantifiers that can be used in the following way:

```
forall (x: int[0,5]) a[t] == 0
exists (x: int[0,4]) b[t] > t
```

Note that the quantified expressions must not have side-effects.

Variables declared using the `meta` keyword can be used in the same way as regular variables, but states that differ only in meta variables are considered equal, which means only one of them will be processed further. Therefore, value of meta variables is not deterministic and cannot be relied upon. The official documentation suggest they can be used for passing a value from one process to another during synchronization.

2.6.2 Clocks and channels

Clocks and channels have to be declared in the same way as variables. A clock is declared as `clock x;` and it is even possible, although not very useful in practice, to declare an array of clocks. Similarly, channels are declared using `chan c;` and an array of channels is also possible. Prepending `urgent`, prevents time from passing if any edge labelled by this channel is enabled and if the `broadcast` modifier is present, the channel in question will be used for n-ary synchronization instead of binary synchronization. Note that `urgent broadcast chan ch` is the correct declaration, but `broadcast urgent chan ch` is an error. Following sections explain synchronizations and urgency in more detail.

2.6.3 System declaration

A system is declared as a synchronous product of one or processes using the `system` keyword followed by a comma-separated list of processes. The Uppaal GUI can be used to design timed automata templates that can be instantiated to create actual processes by assigning values to all their unbound parameters. If a template `T` has a single parameter `int[0,2] x`, then `p = T(1)` declares a process. If a template with unbound parameters is used directly in the list after the `system` keyword, instances for all possible combination of its arguments is created, which means that `system T;` is equivalent to `p0 = T(0); p1 = T(1); p2 = T(2); system p0, p1, p2`. Note that it is not possible to write `system T(0);`, since only templates or previously instantiated processes can be used in the system declaration. If `Q` is a template with two integral parameters, it can be partially specialized using the following syntax: `T(const int[1,4] x) = Q(x, 2*x)`.

Parameters are not limited to integers — it is possible to use arrays, structures, references to variables or even to clocks and channels. References are indicated using the ampersand symbol and any change to a reference is projected to the referenced variable. References can also be used when declaring custom functions, which makes it possible to create a function that swaps values of two integer variables:

```
void swap(int& a, int& b)
{
  int c = a;
  a = b;
  b = c;
}
```

Priorities can be assigned to both channels and processes. Statement `chan priority` followed by a list of channels, where a comma means equal priority and '<' means strictly lower priority, is used to set the channel priorities. If a whole array of channels is mentioned in the declaration, all channels in it will have equal priority, but it is also possible to list individual elements to give them different priorities. Additionally, the `default` keyword can be used to set the priority to all channels not otherwise mentioned in the priority declaration and to all non-synchronizing transitions. Process priorities are embedded directly into the system declaration and use a similar syntax:

```
chan priority c1, c2 < default < ca[0] < ca[1];
system a < b, c;
```

If the default keyword is not used, the priority of the last channel specified is used for all remaining channels. Therefore, `c1 < c2` is equivalent to `c1 < default`.

Priorities are resolved in the following way: A number is assigned to each priority level (a higher number means higher priority) and for each possible transition involving n processes, a vector $\langle p_0, p_1, \ldots, p_n\rangle$ is constructed so that p_0 is the priority of its channel and p_1 to p_n are priorities of all participating processes listed in decreasing order (so that $p_i \geq p_{i+1}$ for $1 \leq i < n$). Then all vectors are truncated to the length of the shortest one and compared lexicographically to find transitions with the highest priority.

This comparison has some non-intuitive properties. For example, if all channels have priority 1 and a process with priority 2 can perform a binary synchronization with either a process with priority 1 or priority 2, then only one of these transitions will be enabled since $\langle 1, 2, 2\rangle$ is lexicographically greater than $\langle 1, 2, 1\rangle$. But if we add a non-synchronizing transition involving priority 1 process, then both synchronizations will be possible, since their priority vectors would be truncated to match the length of newly added $\langle 1, 1\rangle$ and that would make them equal.

Note that priorities influence only action transitions and delays are completely unaffected.

2.6.4 Locations and edges

Each automaton has to have exactly one initial location and each location can also be marked as urgent or committed. If any process is in a committed location, no delay transitions can be performed. Committed locations also prevent the time from passing, but additionally require that no process in a non-committed location can perform any action if at least one process is in a committed location.

Each edge in the automaton can be labelled by *select*, *guard*, *synchronization* and *assign* statements. The select label can be used to declare one or more variables (with the same syntax as the inside of a `forall` expression) that can be used in other statements on the same edge as a non-deterministic choice. Guards consist of a conjunction of clock comparisons and arbitrary expressions involving non-clock variables without side-effects. If `x` is a clock and `i` an integer, `x < 0 && (i == 1 || i == 3)` is a valid guard, while the expression `(x > 1) || (x > 2 && i > 2)` is not, since the clock constraints do not form a conjunction on the outermost level. This rule exists to guarantee that the set of valuations satisfying any guard forms a zone. For the same reason, inequality is not allowed in clock constraints. Clocks can be compared to non-constant integer variables and expressions, but doing so can dramatically decrease the abstraction effectiveness, especially in case of diagonal constraints.

If the guard of an edge is satisfied, it is considered *enabled*, but that does not automatically mean the edge can actually be traversed — that also depends on synchronizations and invariants. This notion is important in case of urgent channels or when priorities are used. For example, if a location has one low priority edge with no guard and one high priority edge with the guard `x == 3` leading to a location with invariant `false`, then there will be no successor for valuations where `x` is 3. Even though the high priority edge cannot be traversed because of the invariant, it is enabled and blocks the low priority transition.

The synchronization label consists of an expression evaluating to a channel followed by the exclamation mark or the question mark signalling the sending and receiving end of a synchronization. The assign statement can contain any expressions involving non-clock variables and assignments of a non-negative values to clocks. When a synchronization is performed, it is guaranteed that an assignment on the sending edge is performed before the receiving edges, which allows inter-process value passing using an auxiliary variable.

The use of a synchronization label on an edge imposes some limitations on its guard. Namely, clocks constraints cannot be present on broadcast receivers or edges synchronizing over an urgent channel, even though the semantics definition of the official documentation seems to suggest otherwise. The purpose of these limitations most likely is to lower the complexity of successor generation.

2.6.5 Formal semantics

If L denotes a vector of locations and v is a valuation that assigns values to both clocks and variables, then we can define the semantics of an Uppaal model as follows:

A delay transition $(L, v) \xrightarrow{\delta} (L, v')$ can be performed if:

- v' is created from v by advancing all clocks by $\delta \in \mathbb{R}_+$,
- $v' \models Inv(L)$,
- there are no committed or urgent locations in L,
- (L, v) has no enabled outgoing transition that can synchronize over an urgent channel.

An internal action transition $(L, v) \longrightarrow (L', v')$ over an edge e from the location l to l' can be performed if:

- v satisfies the guard of e,
- e has no synchronization label,
- l is in L and $L' = L[l'/l]$,

- v' is created from v by performing the assignment described by e,
- $v' \models Inv(L')$,
- there are no committed locations in L or l is a committed location,
- no other enabled transition from (L, v) has a strictly higher priority.

A binary synchronization transition $(L, v) \longrightarrow (L', v')$ over edges $e_1 = (l_1, l'_1)$ and $e_2 = (l_2, l'_2)$ belonging to different processes can be performed if:

- v satisfies the guard of e_1 and e_2,
- the synchronization label evaluates to $ch!$ for e_1 and to $ch?$ for e_2, where ch is a binary synchronization channel,
- l_1 and l_2 are in L and $L' = L[l'_1/l_1][l'_2/l_2]$,
- v' is created from v by performing the assignment described by e_1 and then the assignment of e_2,
- $v' \models Inv(L')$,
- there are no committed locations in L or at least one location from $\{l_1, l_2\}$ is committed,
- no other enabled transition from (L, v) has a strictly higher priority.

A broadcast synchronization transition $(L, v) \longrightarrow (L', v')$ over $n > 0$ edges $e_i = (l_i, l'_i)$ for $1 \leq i \leq n$ belonging to n different processes can be performed if:

- v satisfies the guard of e_i for every $1 \leq i \leq n$,
- the synchronization label of e_1 evaluates to $ch!$ where ch is a broadcast channel and $\{e_i \mid 2 \leq i \leq n\}$ is exactly the set of edges enabled in (L, v) with synchronization labels evaluating to $ch?$,
- $l_1, \ldots, l_n$ are all in L and $L' = L[l'_1/l_1] \cdots [l'_n/l_n]$,
- for every two edges e_j, e_k with $2 \leq j < k$ holds that e_j belongs to a process that appeared sooner in the system declaration than the process which e_k belongs to,
- v' is created from v by successively performing assignments described by $e_1, \ldots, e_n$

- $v' \models Inv(L')$,
- there are no committed locations in L or at least one location from $\{l_i \mid 1 \leq i \leq n\}$ is committed,
- no other enabled transition from (L, v) has a strictly higher priority.

If an error is encountered during the evaluation of guards, invariants, synchronization labels or assignments, Uppaal terminates the verification. Possible errors include assignment to a variable outside the range of its type, accessing an element outside an array bounds, division by zero, shift by a negative number of bits, invalid function call and assigning negative value to a clock. Note that shift by a number of bits greater than 32 is not an error, but the result is undefined and may depend on hardware.

3 Model checking

The purpose of model checking is to decide whether a system satisfies the given property or, to put it differently, whether the system is a model for the given formula. The verification itself is done by exploring the state space, although the specific algorithm and complexity depends on the specific formalism and property.

Even though several logics of varying strength can be used to express properties to be verified, the *linear temporal logic* (LTL) and *computation tree logic* (CTL) are — aside from simple reachability properties — the most commonly used ones since they provide a good balance between the expressive power and verification complexity.

Uppaal supports five kinds of properties from the fragment common to LTL and CTL, namely:

- `A[] p` describes that p is an invariant — holds in every state. This can be written in CTL as $\mathbf{AG}\, p$, which is equivalent to $\mathbf{G}\, p$ in LTL.
- `E<> p` says that a state satisfying p is reachable, which is $\mathbf{EF}\, p$ or $\neg\, \mathbf{AG}\, \neg p$ in CTL. Even though LTL can not express this property directly, a system violates the LTL property $\mathbf{G}\, \neg p$ ($\equiv \neg\, \mathbf{F}\, p$) if and only if it satisfies $\mathbf{EF}\, p$.
- `A<> p` expresses an eventuality — all possible executions will eventually reach a state satisfying p. The corresponding CTL formula is $\mathbf{AF}\, p$, which is expressible in LTL as $\mathbf{F}\, p$.
- `E[] p` says that there is a finite or infinite execution of the system such that p holds in all its states. The CTL formula $\mathbf{EG}\, p$, or equivalently $\neg\, \mathbf{AF}\, \neg p$, is not directly expressible in LTL, but again, we can use its negation $\mathbf{F}\, \neg p$ ($\equiv \neg\, \mathbf{G}\, p$) instead and interpret the result differently.
- `p --> q` is considered to be a shortcut for `A[] (p imply A<> q)` even though nested formulae are otherwise not supported by Uppaal. This corresponds to the CTL formula $\mathbf{AG}(p \implies \mathbf{AF}\, q)$ and its LTL equivalent is $\mathbf{G}(p \implies \mathbf{F}\, q)$.

3.1 LTL

The linear temporal logic (LTL) is used to describe properties of infinite runs. We will first define its syntax and semantics on infinite words from AP^ω, where AP is a set of atomic propositions, and then extend this definition to infinite runs. For any transition system, a *run* can be defined as a sequence of states connected by transitions, but for timed systems, only runs consisting of alternating delay and action transitions are usually considered. Let the transition $\xrightarrow{\delta,a}$ represent a non-negative delay followed by an action transition, then a run of a timed system is an infinite sequence of the form $s_0 \xrightarrow{\delta,a} s_1 \xrightarrow{\delta,a} s_2 \cdots$.

Syntactically, each $a \in AP$ is an LTL formula and if ϕ and ϕ' are LTL formulae, then $\neg\phi$, $\phi \wedge \phi'$, $\mathbf{X}\,\phi$ and $\phi\ \mathbf{U}\ \phi'$ are also valid LTL formulae. The intuitive meaning of the $\mathbf{X}\,\phi$ is that ϕ holds in the next state and the formula $\phi\ \mathbf{U}\ \psi$ holds if ψ will eventually hold and ϕ is true until then. Aside from the common syntactic shortcuts $\vee$, $\implies$, *true* and $false$, we will later define unary temporal operators **F**, **G** and binary operators **R** and **W** by the means of the operator **U**. Also note that operators **X**, **F** and **G** are written as $\bigcirc$, $\Diamond$ and $\Box$ in some literature.

For an infinite word $w = w_0 w_1 w_2 \ldots$, we define the i-th suffix of w as $w^i = w_i w_{i+1} \ldots$. For an LTL formula ϕ, the relation $w \models \phi$ is defined inductively:

- $w \models a$ for $a \in AP$ if $a \in w_0$
- $w \models \neg\phi$ if $w \not\models \phi$
- $w \models \phi_1 \wedge \phi_2$ if $w \models \phi_1$ and $w \models \phi_2$
- $w \models \mathbf{X}\,\phi$ if $w^1 \models \phi$
- $w \models \phi_1\ \mathbf{U}\ \phi_2$ if there is $i \in \mathbb{N}_0$ so that $w^i \models \phi_2$ and for each $0 \leq j < i$, $w^j \models \phi_1$

Even though it suffices to define only the operators "next" (**X**) and "until" (**U**), LTL usually uses following syntactic shortcuts:

- The "future" operator $\mathbf{F}\,\phi \equiv true\ \mathbf{U}\ \phi$
- The "globally" operator $\mathbf{G}\,\phi \equiv \neg\,\mathbf{F}\,\neg\phi$
- The "weak until" has the same meaning as **U**, except it does not require the right side to eventually become true, and can be defined as $\phi\ \mathbf{W}\ \psi \equiv (\phi\ \mathbf{U}\ \psi) \vee \mathbf{G}\,\phi$
- The "release" operator $\phi\ \mathbf{R}\ \psi \equiv \neg(\neg\phi\ \mathbf{U}\ \neg\psi) \equiv \psi\ \mathbf{W}\ (\phi \wedge \psi)$ states that ψ either always holds or holds until the point, where ϕ becomes true (including that point).

Motivation for the last two operators is to simplify the transformation of LTL formulae to normal forms and other algorithmic processing by providing a way to negate the "until" operator. Instead of using the equality $\neg(\phi \mathbf{U} \psi) \equiv (\phi \wedge \neg\psi) \mathbf{U} (\neg\phi \wedge \neg\psi) \vee \mathbf{G}(\phi \wedge \neg\psi)$, that can cause exponential blow-up if we use it to negate $\mathbf{U}$, we can use the equivalence $\neg(\phi \mathbf{U} \psi) \equiv (\phi \wedge \neg\psi) \mathbf{W} (\neg\phi \wedge \neg\psi)$ instead, which avoids adding additional temporal operator, but still duplicates sub-formulae, or $\neg(\phi \mathbf{U} \psi) \equiv \neg\phi \mathbf{R} \neg\psi$, which completely avoids it.

Let $\rho = s_0 \xrightarrow{\delta,a} s_1 \xrightarrow{\delta,a} s_2 \cdots$ be an infinite run of a timed system. Then we can define an infinite word $trace(\rho) = w_0 w_1 w_2 \ldots$, where $w_i = \{a \in AP \mid s_i \models a\}$ and say that ρ satisfies an LTL formula ϕ (written as $\rho \models \phi$) if and only if $trace(\rho) \models \phi$. An LTL property holds for the given system if it holds for all its runs starting in its initial state. Due to the nature of this definition, LTL has a non-intuitive property that, even though ϕ holds for a run iff $\neg\phi$ does not hold for this run, a system may satisfy neither ϕ nor $\neg\phi$. However, this property is essential for LTL model checking. In the case when LTL formula ϕ does not hold for the system in question, it does not hold for at least one of its runs. Such run has to satisfy $\neg\phi$, so we can prove that ϕ holds for a system by showing that it has no run satisfying $\neg\phi$ (a counter-example).

For any LTL formula ϕ, we can construct a *Büchi automaton* that accepts exactly the infinite words, that satisfy ϕ. Formally, a Büchi automaton is a tuple $\mathcal{B} = (Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, $q_0 \in Q$ is an initial state, $\delta : Q \times \Sigma \to 2^Q$ is the transition function and $F \subseteq Q$ is the set of accepting states. A run of a Büchi automaton for an infinite word $w = w_1 w_2 \ldots \in \Sigma^\omega$ is an infinite sequence of states $q_0 q_1 q_2 \ldots$, where $q_i = \delta(q_{i-1}, w_i)$. The word w is accepted if any such run contains a state from F infinitely many times. A Büchi automaton accepts a non-empty set of words if and only if it has a cycle containing an accepting state that is reachable from q_0 [21].

To determine if a transition system $\mathcal{TS} = (S, s_0, \to)$ contains a run whose trace is accepted by a Büchi automaton $\mathcal{B} = (Q, \Sigma, q_0, \delta, F)$ with $\Sigma = 2^{AP}$, we can construct their product and check its non-emptiness. This is the basic idea of the automata-based approach to model checking. The product automaton $\mathcal{TS} \times \mathcal{B}$ has a set of states corresponding to $S \times Q$. Each state (s, q) is accepting if $q \in F$ and the transition $(s_1, q_1) \longrightarrow (s_2, q_2)$ is possible whenever $s_1 \to s_2$ and $q_2 \in \delta(q_1, \{a \in AP \mid s_2 \models a\})$. For timed systems, this construction effectively results in a timed Büchi automaton. The correctness of its construction with the presence of various zone-based abstraction is not obvious and we refer to [3] and [4] for complete proofs.

3.2 Timed LTL

For comparison to the untimed case, this section introduces a timed variant of the linear temporal logic (TLTL). Motivation for a timed logic is that even if we allow clock constraints to appear as atomic propositions, it is impossible to express properties like: Whenever A happens, then B has to happen in 10 time units.

Timed LTL is described in [22] as an extension of LTL by two kind of expressions:

- $\lhd_a \in I$ holds if the time since the action a occurred last time lies in the interval I.
- $\rhd_a \in I$ holds if the time until the action a will occur next lies in the interval I.

Intervals can be closed, open or half-open, but their boundaries have to be from the set $\mathbb{N}_0 \cup \{\infty\}$.

TLTL defined in this way is suited for the action-based approach to model checking and even though it is possible to define a state-based variant, it would get more complicated since it needs to distinguish cases where propositions A and B are required to hold in 5 time units and where their conjunction is required to hold in 5 time units.

Instead of supporting TLTL, we decided to allow clock constraints, including diagonal ones, to be used as atomic propositions in LTL formulae.

3.3 Zeno runs

Some infinite runs of timed systems, as we defined them, do not correspond to any realistic behaviour. These so-called *Zeno runs* are defined as infinite runs for which the sum of all delays is finite. The presence of Zeno runs is undesirable in the timed automata verification since they can cause some properties to be falsely identified as unsatisfied in the case that all violating runs are Zeno.

The simplest solution for this problem is described in [23]. It ensures that at least one time unit has to pass in every accepting cycle, which can be done by modifying the product of the timed system and Büchi automaton in a following way:

- An auxiliary clock aux is added.
- For each accepting state q, an accepting copy q' is created and q is no longer accepting. q' has the same outgoing edges as q.

- If q previously was accepting and there is an edge $(s, g, R, q) \in E$, it is replaced by edges $(s, g \wedge aux < 1, R, q)$ and $(s, g \wedge aux \geq 1, R \cup \{aux\}, q')$

It is a well-known fact that due to this construction, the size of the zone-graph can increase exponentially in the number of clocks. It was shown in [15], that in the presence of coarse abstractions, such as the LU-based one, it is an $\mathcal{NP}$-complete problem to decide if an abstract run in the zone-graph instantiates to at least one non-Zeno run. Therefore, the exponential blow-up seems to be unavoidable for this case.

On the other hand, weaker abstractions allow for more efficient solutions. The algorithm presented in [4] does not eliminate all Zeno runs, but guarantees that each abstract run can be instantiated to at least one non-Zeno run and requires only linear overhead. However, their solution seems to be tied to DFS, which is unsuitable for parallelization and it is also arguable whether the potential gains can justify the use of weaker abstraction.

3.4 Accepting cycle detection algorithms

As we shown in previous sections, LTL verification can be reduced to a graph problem called accepting cycle detection problem (or fair cycle detection problem) on a graph whose vertices correspond to states of the product of a Büchi automaton and the system under verification. Explicit-state model checkers process every state separately, symbolic ones work with whole sets of states described by predicates in a suitable formalism. Both approaches have their advantages and disadvantages, but since our main concern is DiVinE, we will focus on explicit-state model checkers.

The well-known time-optimal algorithm for accepting cycle detection — the nested DFS — runs in linear time, but its correctness depends on the order in which vertices are explored (so-called post-order), so there is no straight-forward way to parallelize it. One possible way to utilize more threads is to run Nested-DFS on all of them, but change the order of successors for each thread. This, so-called swarmed approach, can find a counter-example very quickly, but if none exists, all workers have to explore the whole graph. Other approaches have built upon this idea by adding limited information sharing between individual workers, which leads to a better distribution of work among the individual threads and performs really well if there is a counter-example [24]. However, all workers may still explore the whole graph in the worst case.

A different branch of accepting cycle detection algorithms is based on

BFS, which can be parallelized easily, but these algorithms have to either sacrifice the time optimality or they are no longer on-the-fly, which means the whole graph has to be explored even if there is a counter-example. This includes both MAP and OWCTY which are implemented in DiVinE.

The *maximal accepting predecessor algorithm* (MAP) [25] is based on propagating an identifier of each accepting vertex along edges. Identifiers can have arbitrary (but fixed) order and if a vertex can receive multiple identifiers, the maximal one is chosen and propagated further. The aim is to compute the maximal accepting predecessor for each state and if it is the state itself, it has to be a part of a reachable accepting cycle. Since identifiers may be propagated multiple times along the same edge, each iteration can be super-linear and the time complexity of the whole algorithm is $\mathcal{O}(a^2 \cdot m)$, where m is the number of edges and a is the number of accepting states, but typical complexity is much lower according to the original paper.

The *one way catch them young* (OWCTY) algorithm was originally proposed in [26] as an adaptation of an algorithm used for symbolic model checking and later improved in [27]. It computes the set A of states that are reachable from an accepting cycle by first initializing it to the set of all reachable states and then iteratively removing states that are not reachable from accepting states in A and states unreachable from cycles contained in A. To do that, the number of predecessors still contained in A is computed for each state and then states for which this number is zero are removed from A. Each iteration is linear in the number of edges and the number of iterations can be at most equal to the height of the graph, but it is very low in practice. The improvement presented in [27] adds one iteration of the MAP algorithm without re-propagation to the initialization phase, which does not change the time complexity, but allows early termination. The resulting algorithm was shown to be time-optimal for an important class of weak LTL properties.

4 Implementation

4.1 Used technologies

We developed an interpreter of timed automata in the UPPAAL format and incorporated it into DIVINE [5, 6]. DIVINE is an explicit-state LTL model-checker written in C++ with heavy use of features from C++11 and templates to maximize its run-time performance. Even though its focus platform is Linux, DIVINE can also be compiled and run on Windows, since most of the code is platform-independent.

To implement the timed automata interpreter, we used the UTAP[1] library to read the UPPAAL models from `xml` files and to provide an efficient representation of timed automata and expressions on their edges.

States of the system are constructed from this representation and the UPPAAL DBM library[2] is used to manipulate the part of each state that represents a matrix of clock constraints. This library also provides tools to work with DBM federations that can represent arbitrary sets of clock valuations, which is necessary when performing zone subtractions necessary for Algorithm 2.1.

If an `ltl` file is provided alongside the `xml` file containing the UPPAAL model, LTL properties are loaded from it and when a specific property is chosen, it is negated and a Büchi automaton is created from it using the toolkit already present in DIVINE. The Büchi automaton is multiplied with the timed automata on-the-fly to create a transition system on which the reachability analysis or accepting cycle detection algorithms are run.

4.2 Source code organization

Both the tool iself and its source code is available at its website[3] under the simplified BSD and GNU licenses. Parts relevant to timed automata can be found in the `divine/timed` subdirectory with the exception of the file `divine/generator/timed.h` and both aforementioned libraries that reside in `external`.

The file `timed.h` contains the interface between DIVINE and the interpreter. Additionally, the handling of LTL properties including the on-the-fly multiplication with the resulting Büchi automaton is done here. Most of the

1. Available under the LGPL license at http://people.cs.aau.dk/~adavid/utap/
2. Available under the GPL license at http://people.cs.aau.dk/~adavid/UDBM/
3. http://divine.fi.muni.cz/

```
Input: Source state $(l, Z)$
Output: Set of successors
$Succs \leftarrow \emptyset$
foreach edge $(l, g, R, l') \in E$ do
    $Z' \leftarrow Z \mathbin{\overline{\wedge}} g$
    if $Z' \neq \emptyset$ then
        $Z' \leftarrow Z'[R := 0]$
        // See the definition of $Extra_{diff}$ on the page 14
        foreach $\overline{Z} \in Extra_{diff}(Z')$ do
            if $(l', \overline{Z})$ is not urgent then
                $\overline{Z} \leftarrow \overline{Z}^{\uparrow}$
            end
            $\overline{Z} \leftarrow \overline{Z} \mathbin{\overline{\wedge}} Inv(l')$
            if $\overline{Z} \neq \emptyset$ then
                $Succs \leftarrow Succs \cup \{(l', \overline{Z})\}$
            end
        end
    end
end
return $Succ$
```

Algorithm 4.1: Successor generation.

methods present in this file end up calling corresponding methods of the class `TAGen`, where the actual state generation is done.

The successor generation is outlined as Algorithm 4.1. It uses the extrapolation procedure $Extra_{diff}$ defined on the page 14 and to support process and channel priorities, the whole algorithm has to be enclosed in the procedure listed as Algorithm 2.1 on the page 7.

Files `gen.h` and `gen.cpp` contain the definition and implementation of the class `TAGen`, that takes care of reading the input model, extrapolation with the presence of clock difference constraints (as described in Section 2.5.5) and the actual successor generation.

The class `Evaluator`, defined in `eval.h` and `eval.cpp`, was designed to encapsulate variable representation, expression evaluation, clocks and also the computation of the location-dependent bounds. Since the type system is quite complex, this class has to contain quite a lot of code to implement every possible operation.

Other files contain utility classes and functions. For example, the class

`Clocks` is used to encapsulate all calls to the DBM library.

4.3 User manual

Almost all models supported by Uppaal can be verified using DiVinE. The basic syntax and semantics of the Uppaal modeling language is summarized in Section 2.6. We tried to achieve as high compatibility as possible, but some features are still left unimplemented in the current version, namely:

- Meta variables are treated as regular variables, which means that their value is deterministic and states that differ only in meta variables are treated as different states. This does not affect verification results, but may lead to a bigger state space.
- Only structures containing integers are supported. If the input model uses any structures consisting of arrays or other structures, it cannot be loaded by DiVinE.
- Clock constraints involving non-constant integers are limited. Even though we implemented basic heuristics that try to find the maximal and minimal possible value of a non-constant expressions and it is possible to compare single clocks to such expressions, we only support the option to compare clock differences to constants.

We offer an important feature that is not available in Uppaal — the generation of error states. If an error occurs during the verification, Uppaal terminates, but provides no information about the kind of the error or where it occurred. DiVinE, on the other hand, generates an error state corresponding to the type of the error and continues with the verification. These error states are found automatically during the reachability analysis and full trace from the initial state is generated, which makes it easy to track down the sequence of steps that caused the error and possibly fix it.

To run DiVinE on a Uppaal model, use the command:

```
divine verify model.xml
```

It runs the reachability analysis which tries to find any error states or deadlocks. By default, the LU extrapolation is in use, so the time-lock detection is disabled (reasons for this are explained in Section 2.5). The `--no-reduce` option can be used to force the maximal bounds extrapolation, thus allowing the time-lock detection.

To check LTL properties, a file with the same base name as the model and an `ltl` extension has to be created. This file can contain one or more properties constructed using temporal operators `G`, `F`, `X`, `U`, `R`, `W`, boolean operators written as `&&`, `||`, `!`, `->`, `<->`, brackets and atomic propositions. Due to limitations of the tool that DiVinE uses to transform LTL formulae to corresponding Büchi automata, atomic propositions can contain only lower case letters, digits, underscore and must not begin with a digit. For this reason, C-like defines can be used before property declarations to name any boolean expression in the Uppaal modeling language with a suitable name. The only additional limitation is that all clock constrains have to be defined as separate atomic propositions.

All lines in the LTL file that do not begin with `#define` or `#property` are ignored and can be used for comments. An example of a valid LTL file is given below, other ones are shipped with DiVinE.

```
#define time1 (time <= 60)
#define time2 (time < 70)
#define safe1 (V1.safe)
#define safeall (V1.safe && V2.safe && V3.safe && V4.safe)

#property !F(time1 && safeall)
#property !((!safe1) U (time2 && safeall))
```

Verification of the first property in the corresponding LTL file is run using the following command:

```
divine verify model.xml -p 0
```

To exclude all Zeno runs during the LTL verification (see Section 3.3 for description of Zeno runs), the `-f` option can be used, and to obtain a full report of the verification including an error trace, add `-r` to the arguments. DiVinE also provides options to choose the number of worker threads, request a specific algorithm to be run or enable specific reductions. The complete list of available options can be found using the `divine help` command or in the manual that is a part of DiVinE installation.

4.4 Property semantics

Semantics of timed models come from Uppaal and we have briefly described them in Section 2.6. However, semantics of property specifications are slightly different from Uppaal. First of all, our tool considers only runs where action and delay transitions alternate. This means that DiVinE, unlike Uppaal, does not consider runs where the system stays in one location for an infinite amount of time.

The temporal operator **X** considers the next state to be a state that can be reached by performing exactly one action transition followed by an arbitrary delay transition. For the purpose of LTL verification, all runs ending in a deadlock are transformed to infinite ones by adding an artificial loop over the final state. This ensures that the LTL verification is meaningful even on systems, where not all runs are infinite. Without this transformation, the property $\mathbf{F}\, False$ would hold on a timed automaton with one state and no transitions, because it does not have any infinite runs, which means it cannot have any infinite runs violating this property. If the LU reduction is not applied, time-locks are also considered for this transformation.

5 Experiments

5.1 Source models

To evaluate the performance of our implementation, we have run several tests on the following models and properties:

- `bridge.xml` (distributed with Uppaal) — reachability analysis
- `boxes.xml` (distributed with Uppaal) — reachability analysis
- Variants of `fischer.xml` (distributed with Uppaal) with 4 to 9 processes. The properties being verified are:
  1. $\mathbf{G}\,\mathbf{F}\,c_0 \land \mathbf{G}\,\mathbf{F}\,c_1$, where c_i is true if exactly i processes are in a critical section
  2. $\mathbf{G}(P.req \implies \mathbf{F}\,P.wait)$, where P one of the processes
- Variants of `train-gate.xml` (distributed with Uppaal) with 4 to 9 processes. The properties being verified are:
  1. safety property — the queue never overflows
  2. $\mathbf{G}(T.Appr \implies \mathbf{F}\,T.Cross)$, where T is one of the trains
- A model of energy regulatory networks discussed in [28]. We will refer to it as `bio_bistable`[1].

All measurements were performed on a machine with Intel Xeon X7560 2.27 GHz processors (64 cores total) using DiVinE 3.0 release candidate and Uppaal 4.0.13. All listed numbers are averages of at least three measurement results, except the one case when the verification took over three days.

5.2 Comparison with Uppaal

First of all, we compared time and memory requirements of our implementation to Uppaal and then measured how much can our results be improved by employing multiple parallel workers.

Table 5.1 shows the time and memory required to perform verification using DiVinE and Uppaal as well as the number of states that DiVinE

1. The model is available as `new_br_NoTab_time_g1s_bistable.xml` at `http://anna.fi.muni.cz/~xsafran1/Work/cs2bio/`

Model	DiVinE Time [s]	DiVinE Memory [MB]	DiVinE State count	Uppaal Time [s]	Uppaal Memory [MB]
bio_bistable[d]	25.6	348	780673	303.9	107
boxes[d]	0.1	152	8884	0.1	14
bridge[d]	0.0	143	206	0.0	12
fischer4[p2]	0.1	143	368	0.0	15
fischer5[p2]	0.1	143	1615	0.1	17
fischer6[p2]	0.2	147	7322	1.7	42
fischer7[p2]	0.7	160	33525	57.2	213
fischer8[p2]	3.7	220	153044	5073.2	1989
fischer9[p2]	20.2	581	692699	323234.6	15660
train-gate4[p2]	0.1	143	698	0.0	13
train-gate5[p2]	0.1	143	3716	0.0	13
train-gate6[p2]	0.6	152	22998	0.1	15
train-gate7[p2]	4.0	216	164078	0.6	26
train-gate8[p2]	38.4	867	1330442	5.5	127
train-gate9[p2]	382.1	8014	12097536	60.3	990

[d] — the deadlock-freedom property
[p2] — property 2 for corresponding model expressed as $\mathbf{G}(\phi \implies \mathbf{F}\,\psi)$ or ϕ `-->` ψ

Table 5.1: Comparison to Uppaal.

visited. DiVinE was run with default settings, which means either OWCTY or reachability algorithm is run with two parallel workers. Uppaal was run with memory optimizations turned off (`-C` option). The results on small models turned out according to expectations — DiVinE requires comparable amounts of time, but needs significantly more memory. This happens due to the lack of memory optimizations and because of the fact that the DiVinE executable alone requires around 140 megabytes of memory to run.

On larger models, the results were quite varied. When run on the model `bio_bistable`, DiVinE was over ten times faster than Uppaal, but required three times as much memory. Note that this model uses meta variables, which are treated as regular variables by DiVinE. This means that DiVinE actually might have generated more states than Uppaal. The results also show that DiVinE performs really well on larger variants of the Fischer's protocol. Not only does DiVinE need less memory than Uppaal for variants with more than 7 processes, but the gap in the time requirements is astronomical. The verification of `fischer9` took Uppaal over 3 days, while DiVinE finished in 20 seconds when run on two threads and in 6 seconds when run on 8 threads. On the other hand, DiVinE was both slower and needed more memory on `train-gate` models. However, when 16 threads are used, DiVinE manages to finish the the same task in 57 seconds, which is already marginally faster than Uppaal and it can be sped-up even further.

Different effectiveness of extrapolation and subsumption on different models is most likely the reason why the results are so varied. The larger variants of the Fisher's protocol seem to be particularly unfavourable for Uppaal and after enabling space optimizations on these models, Uppaal needed even more time — the verification of `fischer8` took over 9 hours, which is seven times slower than verification with the `-C` option. For comparison, the difference in time on all versions of `train-gate` between the run with or without memory optimizations was less than 10 %.

Because DBMs are stored as a part of every state, DiVinE currently uses significantly more memory than Uppaal to store the same state space in many cases. However, in the time of writing of this thesis, a tree-based compression technique for DiVinE was in development with the aim to solve this problem and it will most likely be included in DiVinE 3.1 release. The preliminary results are quite promising — reachability on `fischer9` can be done in 220 MB of memory and for `fischer11`, this compression reduces memory requirements from 120 GB to 3 GB with negligible impact on the verification speed. See [29] for detailed description of this compression technique and its experimental evaluation.

Hash compaction [30] is a different approach to memory requirements

	fischer9				train-gate9			
	prop. 1		prop. 2		prop. 1		prop. 2	
	time	MB	time	MB	time	MB	time	MB
1	82.0	609	29.6	416	202.7	3357	497.7	5873
2	64.0	973	20.7	579	115.8	3555	395.8	8009
4	35.4	1139	11.2	611	61.0	3645	207.3	8938
8	18.3	1273	6.2	656	32.3	3717	111.0	9448
16	9.6	1449	3.2	820	16.1	3837	54.8	9823
32	5.2	1765	1.8	1119	8.7	4227	29.0	10262

Table 5.2: Scalability — time and memory requirements.

reduction. Its main idea is to store only hashes in the hash table and not the states themselves. This can greatly reduce memory requirements, but since some states may be left unvisited due to hash collisions, there is a small probability that an existing counter-example may be omitted when the hash comapction is enabled.

5.3 Scalability

The scalability of our implementation was measured on the two biggest models, `train-gate9` and `fischer9`. The nested DFS algorithm was run in the case of one worker and the remaining measurements were using OWCTY, since there is no reason to run OWCTY on one thread. Figures 5.1, 5.2 and Table 5.2 show that even on 32 threads, the verification can be sped-up at least 16 times in comparison to the single-threaded NDFS. We have also measured the scalability on `bio_bistable`, but it turned out that is has only 1762116 edges for 780673 states and also has a fairly linear structure that prevents any algorithm to fully utilize multiple threads.

One of the reasons for reasonably good scalability on most models is that the generation of successor states for timed models is quite expensive when compared to processing simpler modeling languages as `dve`, because many DBM operations have quadratic or cubic complexity in the number of clocks and operations on federations can be even more expensive. This means a lot of time is spent by generating states and the communication with other workers does not occur so frequently.

5.4 Extrapolation comparison

Other batch of measurements aimed to evaluate the effect of different extrapolation techniques on different models. Table 5.3 shows the size of the

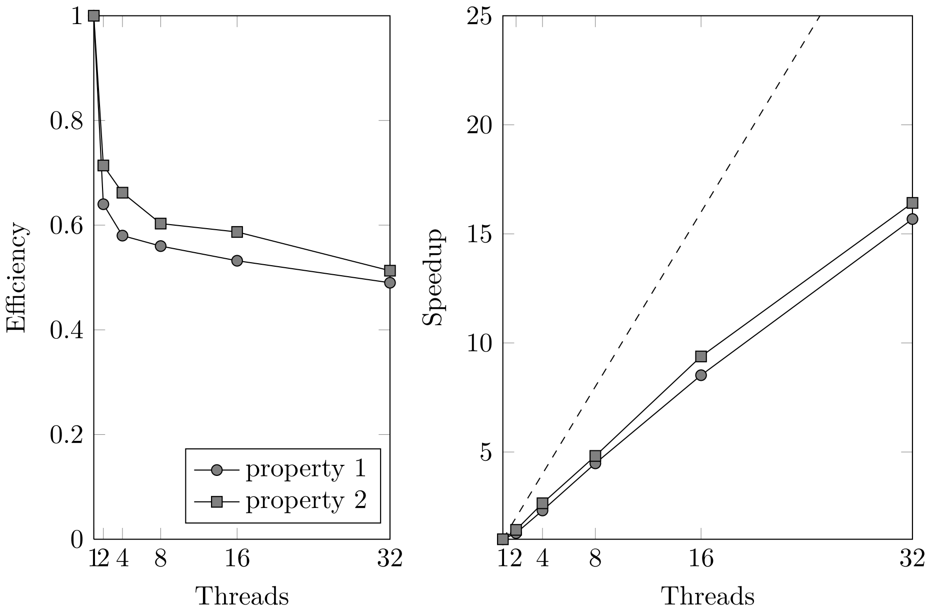


Figure 5.1: Scalability on `fischer9`.

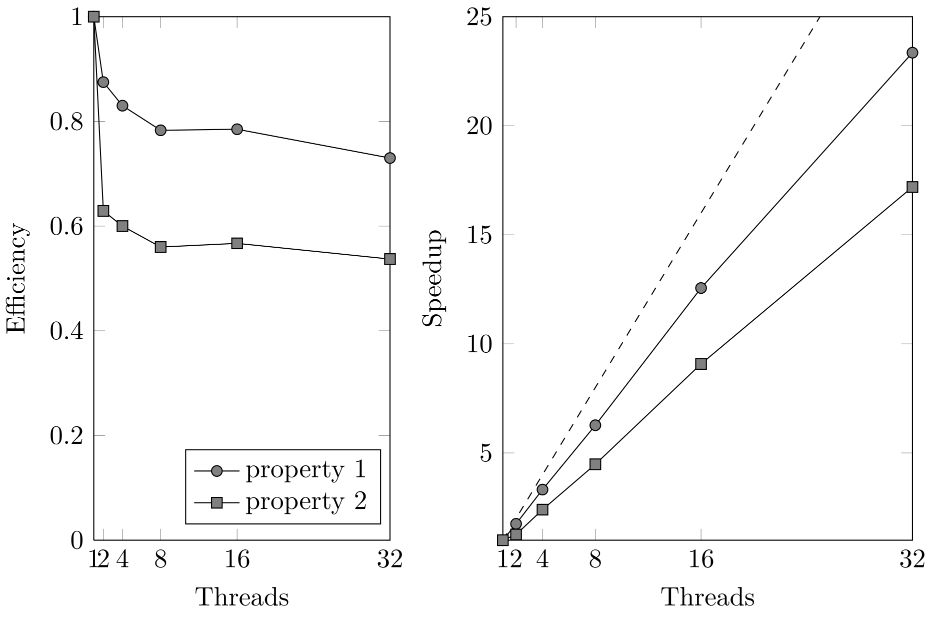


Figure 5.2: Scalability on `train-gate9`.

Model	M	Md	LU	LUd	LBMd	LBLU	LBLUd
boxes	117506	12088	117506	11990	8962	12175	8884
bridge	9315	723	206	206	288	206	206
bio_bistable	780673	780673	780673	780673	780673	780673	780673
fischer4	4209	1792	4209	1792	915	293	293
fischer5	63561	15142	63561	15142	7431	1278	1278
fischer6	1146589	140716	1146589	140716	66609	5799	5799
fischer7	24095709	1425818	24095709	1425818	655075	26652	26652
train-gate4	18869	9977	16997	8633	413	413	413
train-gate5	553276	200776	493476	170976	2141	2141	2141
train-gate6	20093023	3923713	17776483	3283573	12955	12955	12955
train-gate7	—	73427012	—	61105808	90833	90833	90833

Table 5.3: Size of the tate space with different extrapolations.

state space of various models under different extrapolations. 7 different extrapolation technique combinations were used, **M**, **Md**, **LU** and **LUd** stand for extrapolations $Extra_M$, $Extra_M^+$, $Extra_{LU}$ and $Extra_{LU}^+$ introduced in Section 2.5. **LB** marks extrapolations where all bounds are computed locally for each location instead of using global bounds.

As you can see from the results, differences between the best extrapolation and the basic one are immense and it seems that the use of location-dependent bounds has the biggest impact overall. The benefit of separating lower and upper bounds or using diagonal variants of extrapolations highly varies for different models. It may seem strange that the state space of `bio_bistable` has the same number of states under all extrapolations. The reason for this is that all of its clocks are compared to integer variables, which means that clock bounds can be computed only from ranges of these variables (possibly improved using heuristics) and are always the same regardless of the way we compute them.

The release version of DiVinE allows users to use extrapolations **LBMd** (`--no-reduce` option) or **LBLUd** (default), since there is virtually no reason to use the other ones. However, they can still be accessed by enabling respective directives in `eval.cpp` and `clocks.h` and recompiling DiVinE.

6 Conclusions

We have presented an implementation of a timed automata interpreter incorporated into DiVinE that allows parallel and distributed LTL verification of timed systems. Uppaal models with minor restrictions can be directly used as the input and the support of full LTL allows us to verify of a wider variety of properties than Uppaal does. We also performed experimental evaluation showing that our implementation is faster than Uppaal on many models and can be sped-up many times by employing parallelization. Moreover, our tool seems to be the fist non-prototype LTL model checker for timed automata supporting clock difference constraints and very efficient extrapolations.

For the future work, it would be beneficial to implement methods to reduce the memory consumption and by the time of writing this thesis, one technique based on tree compression was already under development.

Bibliography

[1] Behrmann, G., David, R., and Larsen, K. G. A Tutorial on Uppaal. In Bernardo, M. and Corradini, F., eds., *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer Berlin Heidelberg, 2004.

[2] Dalsgaard, A. E., Laarman, A. W., Larsen, K. G., Olesen, M. C., and van de Pol, J. C. Multi-Core Reachability for Timed Automata. In Jurdzinski, M. and Nickovic, D., eds., *10th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS 2012, London, UK*, volume 7595 of *Lecture Notes in Computer Science*, pages 91–106, London, September 2012. Springer Verlag.

[3] Li, G. Checking Timed Büchi Automata Emptiness Using LU-Abstractions. In *Formal Modeling and Analysis of Timed Systems (FORMATS 2009)*, volume 5813 of *LNCS*, pages 228–242. Springer, 2009.

[4] Herbreteau, F., Srivathsan, B., and Walukiewicz, I. Efficient emptiness check for timed Büchi automata. *Formal Methods in System Design*, 40:122–146, 2012.

[5] Barnat, J., Brim, L., Češka, M., and Ročkai, P. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, pages 4–7. IEEE, 2010.

[6] Barnat, J., Brim, L., Havel, V., Havlíček, J., Kriho, J., Lenčo, M., Ročkai, P., Štill, V., and Weiser, J. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)*, page 6. LNCS, 2013. To appear.

[7] Bengtsson, J. and Yi, W. Timed Automata: Semantics, Algorithms and Tools. In Desel, J., Reisig, W., and Rozenberg, G., eds., *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer Berlin Heidelberg, 2004.

[8] Waez, M. T. B., Dingel, J., and Rudie, K. Timed Automata for the Development of Real-Time Systems. Technical Report 2011-579, School of Computing, Queen's University, ON, Canada, September 2011.

[9] Alur, R. and Dill, D. L. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.

[10] Bouyer, P., Dufourd, C., Fleury, E., and Petit, A. Updatable timed automata. *Theor. Comput. Sci.*, 321(2-3):291–345, August 2004.

[11] Bouyer, P. and Chevalier, F. On conciseness of extensions of timed automata. *J. Autom. Lang. Comb.*, 10(4):393–405, apr 2005.

[12] Milner, R. *Communication and concurrency*. Prentice-Hall international series in computer science. Prentice Hall, 1989.

[13] David, A., Håkansson, J., Larsen, K. G., and Pettersson, P. Model checking timed automata with priorities using DBM subtraction. In *Proceedings of the 4th international conference on Formal Modeling and Analysis of Timed Systems*, FORMATS'06, pages 128–142, Berlin, Heidelberg, 2006. Springer-Verlag.

[14] Behrmann, G., Bouyer, P., Larsen, K. G., and Pelánek, R. Lower and upper bounds in zone-based abstractions of timed automata. *Int. J. Softw. Tools Technol. Transf.*, 8(3):204–215, June 2006.

[15] Herbreteau, F. and Srivathsan, B. Coarse abstractions make zeno behaviours difficult to detect. In *Proceedings of the 22nd international conference on Concurrency theory*, CONCUR'11, pages 92–107, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] Behrmann, G., Bouyer, P., Fleury, E., and Larsen, K. G. Static guard analysis in timed automata verification. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 254–270, Berlin, Heidelberg, 2003. Springer-Verlag.

[17] Bouyer, P. Untameable Timed Automata! In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '03, pages 620–631, London, UK, UK, 2003. Springer-Verlag.

[18] Bérard, B., Petit, A., Diekert, V., and Gastin, P. Characterization of the expressive power of silent transitions in timed automata. *Fundam. Inf.*, 36(2-3):145–182, November 1998.

[19] Bengtsson, J. and Yi, W. On Clock Difference Constraints and Termination in Reachability Analysis of Timed Automata. In Dong, J. and

Woodcock, J., eds., *Formal Methods and Software Engineering*, volume 2885 of *Lecture Notes in Computer Science*, pages 491–503. Springer Berlin Heidelberg, 2003.

[20] Bouyer, P., Laroussinie, F., and Reynier, P. A. Diagonal Constraints in Timed Automata: Forward Analysis of Timed Systems. In Pettersson, P. and Yi, W., eds., *Formal Modeling and Analysis of Timed Systems*, volume 3829 of *Lecture Notes in Computer Science*, pages 112–126. Springer Berlin Heidelberg, 2005.

[21] Baier, C. and Katoen, J. P., eds. *Principles of model checking*. MIT Press, Cambridge, Massachusetts, 2008.

[22] D'Souza, D. A logical characterisation of event clock automata. *International Journal of Foundations of Computer Science*, 14(04):625–639, 2003.

[23] Tripakis, S. Verifying Progress in Timed Systems. In *Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, ARTS '99, pages 299–314, London, UK, UK, 1999. Springer-Verlag.

[24] Laarman, A., Langerak, R., Van De Pol, J., Weber, M., and Wijs, A. Multi-core nested depth-first search. In *Proceedings of the 9th international conference on Automated technology for verification and analysis*, ATVA'11, pages 321–335, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] Brim, L., Černá, I., Moravec, P., and Šimša, J. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, volume 3312 of *LNCS*, pages 352–366. Springer-Verlag, 2004.

[26] Černá, I. and Pelánek, R. Distributed Explicit Fair Cycle Detection. In *SPIN'03*, volume 2648 of *LNCS*, pages 49–73. Springer, 2003.

[27] Barnat, J., Brim, L., and Ročkai, P. A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In *Formal Methods and Software Engineering (ICFEM 2009)*, volume 5885 of *LNCS*, pages 407–425. Springer, 2009.

[28] Van Goethem, S., Jacquet, J., Brim, L., and Šafránek, D. Timed Modelling of Gene Networks with Arbitrary Expression Level Discretization. In *Proceedings of 3rd International Workshop on Interactions between Computer Science and Biology*, pages 1–15, 2012.

[29] Štill, V. *State space compression for the DiVinE model checker*. Bachelor's thesis. Masaryk University, Faculty of Informatics, 2013. Available at <http://is.muni.cz/th/373979/fi_b/>.

[30] Barnat, J., Havlíček, J., and Ročkai, P. Distributed LTL Model Checking with Hash Compaction. In *Proceedings of PASM/PDMC 2012*. Electronic Notes in Theoretical Computer Science, 2013.