Time-Darts: A Data Structure for Verification of Closed Timed Automata[*]

Kenneth Y. Jørgensen Kim G. Larsen Jiří Srba
Department of Computer Science, Aalborg University
Selma Lagerlöfs Vej 300, 9220 Aalborg East, Denmark
{kyrke,kgl,srba}@cs.aau.dk

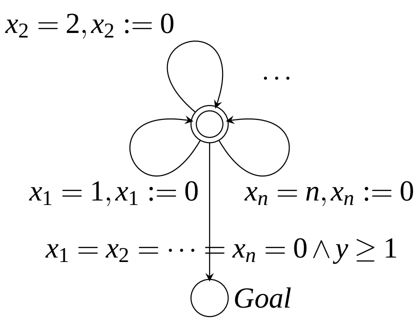
Symbolic data structures for model checking timed systems have been subject to a significant research, with Difference Bound Matrices (DBMs) still being the preferred data structure in several mature verification tools. In comparison, discretization offers an easy alternative, with all operations having linear-time complexity in the number of clocks, and yet valid for a large class of *closed* systems. Unfortunately, fine-grained discretization causes itself a state-space explosion. We introduce a new data structure called *time-darts* for the symbolic representation of state-spaces of timed automata. Compared with the complete discretization, a single time-dart allows to represent an arbitrary large set of states, yet the time complexity of operations on time-darts remain linear in the number of clocks. We prove the correctness of the suggested reachability algorithm and perform several experiments in order to compare the performance of time-darts and the complete discretization. The main conclusion is that in all our experiments the time-dart method outperforms the complete discretization and it scales significantly better for models with larger constants.

1 Introduction

Timed automata [2] are a well studied formalism for modelling and verification of real-time systems. Over the years extensive research effort has been made towards the design of data structures and algorithms allowing for efficient model checking of this modeling formalism. These techniques have by now been implemented in a number of mature tools (e.g. UPPAAL [4], IF [9], Kronos [14], PAT [21], Rabbit [6], RED [16]), with zone-based analysis [15, 5] still being predominant, stemming from that fact that Difference Bound Matrices (DBMs) offer a very compact data structure for efficient implementation of the various operations required for the state-space exploration. Still the DBM data structure suffers from the fact that all operations have at least quadratic—and the crucial closure operation even cubic—time complexity in the number of clocks (though for diagonal-free constraints the operations can be implemented in quadratic time [23]). In contrast, as advocated in [10, 19], the use of *discretization* offers an easy alternative, with all operations having linear complexity in the number of clocks, and yet valid for the large—and in practice often sufficient—class of closed systems that contain only nonstrict guards; moreover for reachability checking the continuous and discrete semantics coincide on this subclass.

As an example consider the timed automaton shown in Figure 1, containing n clocks and n self-loops where the i'th loop has the guard $x_i = i$ and resets the clock x_i. We are interested in whether or not we can reach the *Goal* location. For this to happen, all clocks $x_1, \dots, x_n$ must *simultaneously* have the value zero, corresponding effectively to calculating the least common multiple of the numbers from 1 to n. In Figure 1 we compare the verification times of the zone-based reachability performed in UPPAAL with that of a simple Python based implementation of discrete time reachability checker for timed automata.

[*]The paper was supported by VKR Center of Excellence MT-LAB.



Size	Discrete	Uppaal
4	0.2	<0.1
5	1.1	2.2
6	5.5	27.5
7	97	>300
8	out of memory	-

Figure 1: Discrete vs. zone-based reachability algorithm (time in seconds)

Opposite to what one might expect, it turns out that in this case the naive discrete implementation without any speed optimizations outperforms a state-of-the-art model checking tool.

On the other hand, the disadvantage of discretization is that the number of states to be considered explodes when the size of the constants appearing in the constraints of the timed automaton are increased. In fact, the experimental results of Lamport [19] show that the zone-based methods outperform discreterized methods when the maximum constant in the timed automaton exceeds 10. Also in [19] the BDD-based model checker SMV was applied to symbolically represent the discreterized state-space. This representation is less sensitive to the maximum constant of the model, yet in experimental results [7, 3] it appears that the zone-based method is still superior for constants larger than 16.

Inspired by the success of discretization reported in Figure 1, we revisit the problem of finding efficient data structures for the analysis of timed automata. In particular, we introduce a new data structure called *time-darts* for the symbolic representation of the state-spaces of timed automata. Compared with the complete discretization, a single time-dart allows us to represent an arbitrary large set of states, yet the time complexity of operations remain linear in the number of clocks, providing a potential advantage compared to DBMs.

We propose a symbolic reachability algorithm based on a forward search. To ensure the termination of the forward search the so-called extrapolation of time darts with respect to the maximum constant appearing in the model is required. Given the subtleties of extrapolation,[1] we prove the termination and correctness of the proposed algorithm. We perform several experiments in order to compare the performance of time-darts versus the complete discretization representation. The main conclusion is that the time-dart method consistently outperforms the complete discretization and it is particularly well suited for scaling up the constants used in the model. Given the simplicity of implementing discrete-time algorithms compared to the DBM-based ones, our method can be in practice well suited for the verification of closed time systems with moderately large constants.

2 Timed Automata

Let $\mathbb{N}$ be the set of nonnegative integers and let $\mathbb{N}^\infty = \mathbb{N} \cup \{\infty\}$. The comparison and addition operators are defined as expected, in particular $n < \infty$ and $n + \infty = \infty$ for $n \in \mathbb{N}$.

A *Discrete Timed Transition System* (*DTTS*) is a pair $T = (S, \longrightarrow)$ where S is a set of *states*, and $\longrightarrow \subseteq S \times (\mathbb{N} \cup \{\tau\}) \times S$ is a *transition relation* written $s \xrightarrow{d} s'$ if $(s, d, s') \in \longrightarrow$ where $d \in \mathbb{N}$ for *delay*

[1]Despite several earlier claims, it was not before [8] that a complete—and a quite non-trivial—proof of correctness of zone-based forward reachability was given.

actions, and $s \xrightarrow{\tau} s'$ if $(s,\tau,s') \in \longrightarrow$ for *switch actions*. By $\longrightarrow^*$ we denote the reflexive and transitive closure of the relation $\longrightarrow \overset{\text{def}}{=} \xrightarrow{\tau} \cup \bigcup_{d\in\mathbb{N}} \xrightarrow{d}$.

Let C be a finite set of clocks. A (discrete) *clock valuation* of clocks from C is a function $v : C \to \mathbb{N}$. The set of all clock valuations is denoted by $\mathscr{V}$. Let $v \in \mathscr{V}$. We define the valuation $v+d$ after a delay of $d \in \mathbb{N}$ time units by $(v+d)(x) \overset{\text{def}}{=} v(x)+d$ for every $x \in C$. For a subset $R \subseteq C$ of clocks we define the valuation $v[R := 0]$ where all clocks from R are reset to zero by $v[R := 0](x) \overset{\text{def}}{=} v(x)$ for $x \in C \setminus R$ and $v[R := 0](x) \overset{\text{def}}{=} 0$ for $x \in R$.

A nonstrict (or closed) *time interval* I is of the form $[a,b]$ or $[a,\infty)$ where $a,b \in \mathbb{N}$ and $a \le b$. The set of all time intervals is denoted by $\mathscr{I}$. We use the functions $ub, lb : \mathscr{I} \longrightarrow \mathbb{N}$ to return the upper resp. lower bound of a given interval. A *clock guard* over the set of clocks C is a function $g : C \longrightarrow \mathscr{I}$ that assigns a time interval to each clock. We denote the set of all clock guards over C by $\mathscr{G}(C)$. We write $v \models g$ for a valuation $v \in \mathscr{V}$ and a guard $g \in \mathscr{G}(C)$ whenever $v(x) \in g(x)$ for all $x \in C$.

Timed Automaton A *timed automaton* (TA) is a tuple $A = (L, C, \longrightarrow, \ell_0)$ where L is a finite set of *locations*, C is a finite set of *clocks*, $\longrightarrow \subseteq L \times \mathscr{G}(C) \times 2^C \times L$ is a finite *transition relation* written $\ell \xrightarrow{g,R} \ell'$ for $(\ell, g, R, \ell') \in \longrightarrow$, and $\ell_0 \in L$ is an *initial* location.

Note that we do not consider clock invariants as they can be substituted by adding corresponding clock guards to the outgoing transitions while preserving the answers to location-reachability checking.

A *configuration* of a timed automaton A is a pair (ℓ, v) where $\ell \in L$ and $v \in \mathscr{V}$. We denote the set of all configurations of A by $Conf(A)$. The *initial configuration* of A is (ℓ_0, v_0) where $v_0(x) \overset{\text{def}}{=} 0$ for all $x \in C$.

Discrete Semantics A TA $A = (L, C, \longrightarrow, \ell_0)$ generates a *DTTS* $T_{DS}(A) \overset{\text{def}}{=} (Conf(A), \longrightarrow_{DS})$ where states are configurations of A and the transitions are given by

$$\begin{array}{ll} (\ell,v) \xrightarrow{\tau}_{DS} (\ell', v[R := 0]) & \text{if } \ell \xrightarrow{g,R} \ell' \text{ such that } v \models g \\ (\ell,v) \xrightarrow{d}_{DS} (\ell, v+d) & \text{if } d \in \mathbb{N}. \end{array}$$

The discrete semantics clearly yields an infinite state space due to unbounded time delays. We will now recall that the reachability problem for a TA A can be solved by looking only at a finite prefix of the state space up to some constant determining the largest possible delay. Let MC be the largest integer that appears in any guard of A. Two valuations $v, v' \in \mathscr{V}$ are *equivalent* up to the maximal constant MC, written $v \equiv_{MC} v'$, if

$$\forall x \in C.\ v(x) = v'(x) \ \vee\ (v(x) > MC \ \wedge\ v'(x) > MC).$$

Observe that the equivalence relation $\equiv_{MC}$ has only finitely many equivalence classes as there are finitely many clocks and each of them is bounded by the constant MC.

Lemma 2.1 *Let $v, v' \in \mathscr{V}$ s.t. $v \equiv_{MC} v'$ and let $g \in \mathscr{G}(C)$ be a guard where $0 \le lb(g(x)) \le MC$, and $ub(g(x)) = \infty$ or $0 \le ub(g(x)) \le MC$ for all $x \in C$. Then $v \models g$ iff $v' \models g$.*

Moreover, any two configurations with the same location and equivalent valuations are timed bisimilar (for the definition of timed bisimilarity see e.g [20]).

Lemma 2.2 *The relation $B = \{((\ell,v),(\ell,v')) \mid v \equiv_{MC} v'\}$ is a timed bisimulation for any timed automaton with its maximum constant MC.*

Proof Let $((\ell,v),(\ell,v')) \in B$. We analyse only the switch and delay actions from (ℓ,v); the situation for the transitions from (ℓ,v') is symmetric.

- Assume that $(\ell,v) \xrightarrow{\tau}_{DS} (\ell',v[R:=0])$ via a transition $\ell \xrightarrow{g,R} \ell'$. Due to Lemma 2.1 and the fact that $v \models g$, we get $v' \models g$. Hence also $(\ell,v') \xrightarrow{\tau}_{DS} (\ell',v'[R:=0])$ and it is easy to verify that $v[R:=0] \equiv_{MC} v'[R:=0]$.
- Assume that $(\ell,v) \xrightarrow{d}_{DS} (\ell,v+d)$. We want to argue that also $(\ell,v') \xrightarrow{d}_{DS} (\ell,v'+d)$ such that $v+d \equiv_{MC} v'+d$, however, this is easy to see from that facts that (i) if $v(x),v'(x) > MC$ then also $(v+d)(x),(v'+d)(x) > MC$ and (ii) if $v(x) = v'(x) \leq MC$ then $(v+d)(x) = (v'+d)(x)$. ▮

We now define an alternative discrete semantics of TA with only finitely many reachable configurations. First, for the maximum constant MC, we define a bounded addition operator

$$n \oplus_{MC} m \overset{\text{def}}{=} \begin{cases} MC+1 & \text{if } n+m > MC, \\ n+m & \text{otherwise.} \end{cases}$$

The operation $\oplus_{MC}$ is in a natural way extended to functions and tuples.

Bounded Discrete Semantics A TA $A = (L,C,\longrightarrow,\ell_0)$ with the maximal constant MC generates a *DTTS* $T_{BDS}(A) \overset{\text{def}}{=} (Conf(A),\longrightarrow_{BDS})$ where states are configurations of A and the transition relation $\longrightarrow$ is defined by

$$\begin{array}{ll} (\ell,v) \xrightarrow{\tau}_{BDS} (\ell',v[R:=0]) & \text{if } \ell \xrightarrow{g,R} \ell' \text{ such that } v \models g \\ (\ell,v) \xrightarrow{d}_{BDS} (\ell,v \oplus_{MC} d) & \text{if } d \in \mathbb{N}. \end{array}$$

We say that a location ℓ_g is *reachable* in $T_{DS}(A)$ resp. in $T_{BDS}(A)$ if $(\ell_0,v_0) \longrightarrow^* (\ell_g,v)$ for some valuation v where $\longrightarrow$ is $\longrightarrow_{DS}$ resp. $\longrightarrow_{BDS}$.

We conclude that the bounded semantics preserves reachability of locations, the main problem we are interested in. This fact follows from Lemma 2.2.

Theorem 2.3 *A location ℓ is reachable in $T_{DS}(A)$ iff ℓ is reachable in $T_{BDS}(A)$.*

3 Naive Reachability Algorithm

We can now describe the naive search algorithm that explores in a standard way, point by point, the finite state-space of the bounded semantics and provides the answer to the location reachability problem. Algorithm 1 searches through all reachable states, starting from the initial location, until a goal configuration is found (returning true) or all configurations are visited (returning false). Notice that the algorithm is nondeterministic as it is not specified what element should be removed from *Waiting* at line 5 (such choice depends on the concrete search strategy like DFS or BFS). The next theorem states that Algorithm 1 is correct.

Theorem 3.1 *Let A be a timed automaton and let ℓ_g be a location. Algorithm 1 terminates, and it returns true iff ℓ_g is reachable in the discrete semantics $T_{DS}(A)$.*

Proof First notice that the algorithm terminates because there is only a finite number of configurations that can be possibly added to *Waiting*: the number of locations is finite and due to the bounded addition at line 9 the total number of configurations is finite too. Whenever a configuration is removed from the set *Waiting*, it is added to *Passed* (line 6) and can never be inserted into *Waiting* again due to the test at line 12. As we remove one element from *Waiting* each time the body of the while-loop is executed,

Algorithm 1: Naive reachability algorithm

```
   Input: A timed automaton $A = (L, C, \longrightarrow, \ell_0)$ and a location $\ell_g \in L$
   Output: true if $\ell_g$ is reachable in $T_{DS}(A)$, false otherwise
1  begin
2      $Passed := \emptyset$; $Waiting := \emptyset$;
3      AddToPW($\ell_0, v_0$)
4      while $Waiting \neq \emptyset$ do
5          remove some $(\ell, v)$ from $Waiting$
6          $Passed := Passed \cup \{(\ell, v)\}$
7          forall the $(\ell', v')$ such that $(\ell, v) \xrightarrow{\tau}_{BDS} (\ell', v')$ do
8              AddToPW($\ell', v'$)
9          AddToPW($\ell, (v \oplus_{MC} 1)$)
10     return false

11 AddToPW($\ell$, $v$)
12 if $(\ell, v) \notin Passed \cup Waiting$ then
13     if $\ell = \ell_g$ then
14         return true    /* and terminate the whole algorithm */
15     else
16         $Waiting := Waiting \cup \{(\ell, v)\}$
```

the algorithm necessarily terminates either at line 10 or even earlier at line 14 if the goal location is reachable.

Now we prove the correctness part. By Theorem 2.3 we can equivalently argue that the algorithm returns true iff ℓ_g is reachable in $T_{BDS}(A)$.

"$\Rightarrow$": Assume that Algorithm 1 returns true. We want to show that the location ℓ_g is reachable in $T_{BDS}(A)$. This can be established by the following invariant: any call of `AddToPW` with the argument (ℓ, v) implies that the configuration (ℓ, v) is reachable in $T_{BDS}(A)$. For the initialisation at line 3 this clearly holds. In the while-loop the calls to `AddToPW` are at lines 8 and 9. At line 8 we know by the invariant that (ℓ, v) is reachable and we call `AddToPW` only with (ℓ', v') such that $(\ell, v) \xrightarrow{\tau}_{BDS} (\ell', v')$, so the invariant is preserved. Similarly at line 9 for the argument $(\ell, (v \oplus_{MC} 1))$ of `AddToPW` holds that $(\ell, v) \xrightarrow{1}_{BDS} (\ell, (v \oplus_{MC} 1))$, so it is reachable as well.

"$\Leftarrow$": Assume that a configuration (ℓ', v') is reachable via n transitions in $T_{BDS}(A)$, formally $(\ell_0, v_0) \longrightarrow^n_{BDS} (\ell', v')$, where (without loss of generality) all delay transitions in the sequence are of the form $\xrightarrow{1}_{BDS}$, in other words they add exactly one-unit time delay. By induction on n we will establish that during any execution of the algorithm there is eventually a call of `AddToPW` with the argument (ℓ', v'), unless the algorithm already returned true. If $n = 0$ then the claim is trivial due to the call at line 3. If $n > 0$ then either (i) $(\ell_0, v_0) \longrightarrow^{n-1}_{BDS} (\ell, v) \xrightarrow{\tau}_{BDS} (\ell', v')$ with the last switch action or (ii) $(\ell_0, v_0) \longrightarrow^{n-1}_{BDS} (\ell, v) \xrightarrow{1}_{BDS} (\ell', v')$ with the last one-unit delay action. By induction hypothesis, unless the algorithm already returned true, there will be eventually a call of `AddToPW` with the argument (ℓ, v) and this element is added to the set *Waiting*. Because the algorithm terminates, the element (ℓ, v) will be eventually removed from *Waiting* at line 5 and its switch successors and the one-unit delay successor will become arguments of the call to `AddToPW` at lines 8 and 9. Hence the induction hypothesis for the

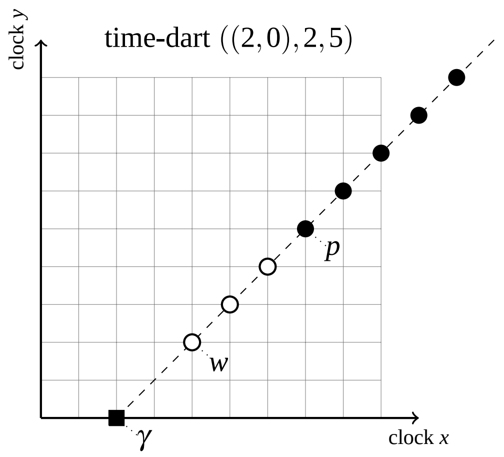


Figure 2: A time-dart (γ, w, p) where $\gamma(x) = 2$, $\gamma(y) = 0$, $w = 2$ and $p = 5$

cases (i) and (ii) is established. ∎

4 Time-Dart Data Structure

We shall now present a novel symbolic representation of the discrete state space. The symbolic structure, we call it a *time-dart*, allows us to represent a number of concrete configurations in a more compact way so that time successors of a configuration are stored without being explicitly enumerated. We start with the definition of an *anchor point*, denoting the beginning of a time-dart.

An *anchor point* over a set of clocks C is a clock valuation $\gamma : C \longrightarrow \mathbb{N}$ where $\gamma(x) = 0$ for at least one $x \in C$. We denote the set of all anchor points over a set of clocks C by $Anchors(C)$. Now we are ready to define *time-darts*.

Time-Dart A *time-dart* over a set of clocks C is a triple (γ, w, p) where $\gamma \in Anchors(C)$ is an anchor point, $w \in \mathbb{N}$ is a waiting distance, and $p \in \mathbb{N}^\infty$ is a passed distance such that $w \leq p$.

The intuition is that a time-dart describes the corresponding passed and waiting sets in a given location. Figure 2 shows a dart example with two clocks x and y, anchor point $(2,0)$, waiting distance 2 and passed distance 5. The empty circles represent the points in the waiting set and the filled circles represent the points in the passed set, formally defined by: $Waiting(\gamma, w, p) = \{(\gamma + d) \mid w \leq d < p\}$ and $Passed(\gamma, w, p) = \{(\gamma + d) \mid d \geq p\}$.

The *passed-waiting list* is represented as a function from locations and anchor points to the corresponding waiting and passed distances (here $\perp$ represents the undefined value):

$$PW : L \times Anchors \longrightarrow (\mathbb{N} \times \mathbb{N}^\infty) \cup \{\perp\} \ .$$

Such a structure can be conveniently implemented as a hash map. A given passed-waiting list PW defines the sets of passed and waiting configurations.

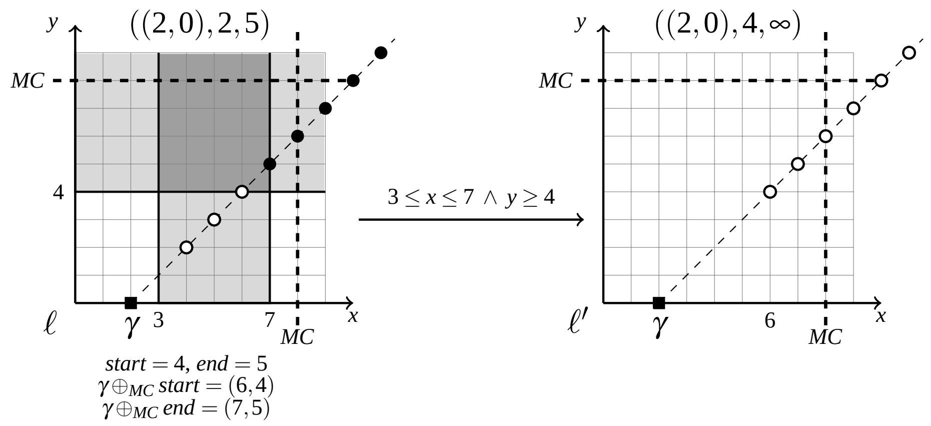
$$Waiting(PW) = \{(\ell, v) \mid \exists \gamma. PW(\ell, \gamma) = (w, p) \neq \perp \text{ and } v \in Waiting(\gamma, w, p)\}$$
$$Passed(PW) = \{(\ell, v) \mid \exists \gamma. PW(\ell, \gamma) = (w, p) \neq \perp \text{ and } v \in Passed(\gamma, w, p)\}$$

Algorithm 2: Time-dart reachability algorithm

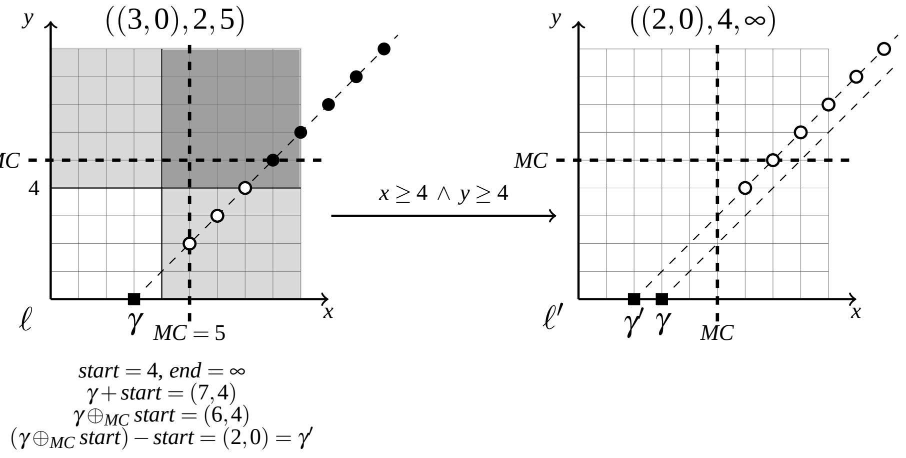
```
Input: A timed automaton $A = (L, C, \longrightarrow, \ell_0)$ and a location $\ell_g \in L$
Output: true if $\ell_g$ is reachable in $T_{DS}(A)$, false otherwise

1  begin
2      $PW(\ell,\gamma) := \perp$ for all $(\ell,\gamma)$   /* default value */
3      AddToPW($\ell_0, \gamma_0, 0, \infty$) where $\gamma_0(x) := 0$ for all $x \in C$
4      while $\exists(\ell,\gamma).\, PW(\ell,\gamma) = (w,p)$ and $w < p$ do
5          $PW(\ell,\gamma) := (w,w)$
6          foreach $(\ell, g, R, \ell') \in \longrightarrow$ do
7              $start := \max(w, \max(\{lb(g(x)) - \gamma(x) \mid x \in C\}))$
8              $end := \min(\{ub(g(x)) - \gamma(x) \mid x \in C\})$
9              if $(start < p \wedge start \leq end)$ then
10                 if $R = \emptyset$ then
11                     AddToPW($\ell', (\gamma \oplus_{MC} start) - start, start, \infty$)
12                 else
13                     $stop := \max\{start, MC + 1 - \min_{x \in C \setminus R} \gamma(x)\}$
14                     for $n := start$ to $\min(end, p-1, stop)$ do
15                         AddToPW($\ell', (\gamma \oplus_{MC} n)[R := 0], 0, \infty$)
16     return false

17 AddToPW($\ell, \gamma, w, p$)
18 if $\ell = \ell_g$ then
19     return true   /* and terminate the whole algorithm */
20 if $PW(\ell,\gamma) = \perp$ then
21     $PW(\ell,\gamma) := (w,p)$
22 else
23     $(w', p') := PW(\ell,\gamma)$
24     $PW(\ell,\gamma) := (\min(w,w'), \min(p,p'))$
```

5 Reachability Algorithm Based on Time-Darts

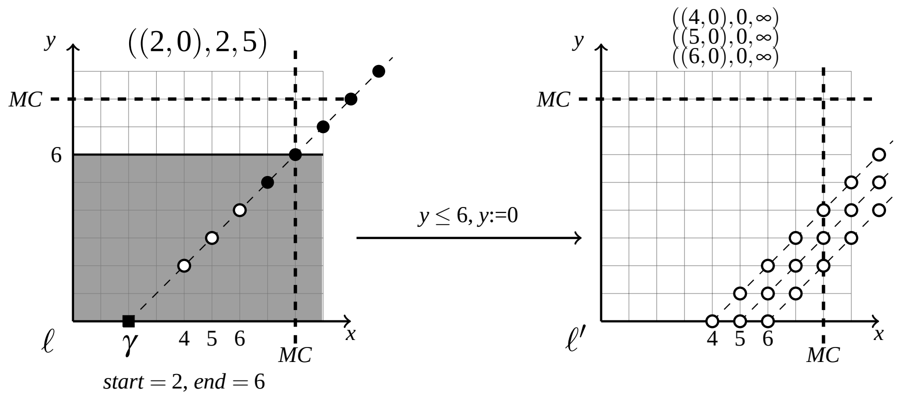
We can now present Algorithm 2 showing us how time-darts can be used to compute the set of reachable states of a timed automaton in a compact and efficient way. The algorithm repeatedly selects from the waiting list a location ℓ with a time-dart (γ, w, p) that still contains some unexplored points ($w < p$). Then for each edge $\ell \xrightarrow{g,R} \ell'$ in the timed automaton it computes the *start* and *end* delays from the anchor point such that *start* is the minimum delay where the guard g gets first enabled and *end* is the maximum possible delay so that g is still enabled. Depending on the concrete situation it will add a new time-dart (or a set of darts) with location ℓ' to the waiting list by calling `AddToPW`. A switch transition is always followed by a delay transition that is computed symbolically (including in a single step all possible delays). There are several cases that determine what kinds of new time-darts are generated. Figure 3 gives a graphical overview of the different situations. In Figure 3a we illustrate the produced time-dart that serves as the argument for the call to `AddToPW` at line 11 of the algorithm (no clocks are reset). Here the anchor point γ is not modified because $((\gamma \oplus_{MC} start) - start) = \gamma$. Figure 3b shows another example



(a) Unchanged anchor point

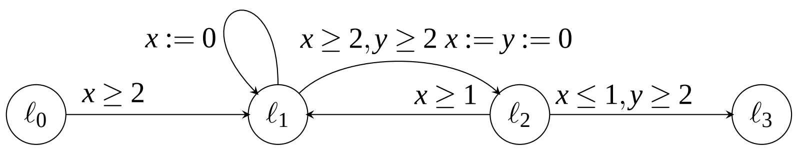


(b) Shift of anchor point



(c) Reset of a clock

Figure 3: Successor generation for a selected time-dart



Location	Anchor	0	1	2	3	4	5	6	7
ℓ_0	$(0,0)$	$(\mathbf{0},\infty)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$
ℓ_1	$(0,0)$	$\perp$	$(\mathbf{2},\infty)$	$(2,2)$	$(2,2)$	$(2,2)$	$(\mathbf{1},\mathbf{2})$	$(1,1)$	$(1,1)$
ℓ_1	$(0,1)$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$(\mathbf{0},\infty)$	$(0,0)$
ℓ_1	$(0,2)$	$\perp$	$\perp$	$(\mathbf{0},\infty)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$
ℓ_1	$(0,3)$	$\perp$	$\perp$	$(0,\infty)$	$(\mathbf{0},\infty)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$
ℓ_2	$(0,0)$	$\perp$	$\perp$	$(0,\infty)$	$(0,\infty)$	$(\mathbf{0},\infty)$	$(0,0)$	$(0,0)$	$(0,0)$

Figure 4: Example of an execution of the algorithm (columns represent the number of iterations of the main while-loop; all unlisted pairs of locations and anchor points are constantly having the value $\perp$)

of a call at line 11 where the anchor point changes. Finally, Figure 3c explains the case where some clocks are reset and several new darts are added in the body of the for-loop at line 15 of the algorithm (the for-loop starts from *start* and stops as soon as either *end*, the beginning of the passed list, or the number *stop*—used for performance optimization–is reached). We note that the figures show the time-darts that the function `AddToPW` is called with; inside the function the information already stored in the passed-waiting list for the concrete anchor point and location is updated so that we take the minimum of the current and new waiting and passed distances (line 24 of the algorithm).

Let us now demonstrate the execution of Algorithm 2 on the automaton depicted in Figure 4, where we ask if the goal location ℓ_3 is reachable from the initial state (ℓ_0, v_0) where $v_0(x) = v_0(y) = 0$. The values stored in the passed-waiting list after each iteration of the while-loop are shown in the table such that a column labelled with a number i is the status of the passed-waiting list after the i'th execution of the body of the while-loop; all values for anchor points not listed in the table are constantly $\perp$.

Initially we set $PW(\ell_0,(0,0)) = (0,\infty)$, meaning that all points reachable from the initial valuation after an arbitrary delay action belong to the waiting list and should be explored. As ℓ_0 is not the goal state, the algorithm continues with the execution of the main while-loop. In the first iteration of the loop we pick the only element in the waiting list so that $\ell = \ell_0$, $\gamma = (0,0)$, $w = 0$ and $p = \infty$. Then we update $PW(\ell_0,(0,0))$ to $(0,0)$[2] according to line 5 of the algorithm, meaning that all points on the dart are now in the passed list. After this we consider the transition from ℓ_0 to ℓ_1 with the guard $x \in [2,\infty)$ (and the implicit guard $y \in [0,\infty)$) and calculate the values of *start* (minimum delay from the anchor point to satisfy the guard and at the same time having at least the delay w where the waiting list starts) and *end* (maximum delay from the anchor point so that the guard is still satisfied). In our example we have $start = \max(0,(2-0),(0-0)) = 2$ and $end = \min((\infty-2),(\infty-0)) = \infty$.

Next we consider the test at line 9 that requires that the minimum delay *start* to enable all guards is not in the region of already passed points ($start < p$) and at the same time that it is below the maximum delay after which the guard become disabled ($start \leq end$). If this test fails, there is no need to do

[2] In each column we mark by bold font the element that is picked in the next iteration of the while-loop.

anything with the currently picked element from the waiting list. As the values in our example satisfy the condition at line 9 and no clocks are reset, we update according to line 11 of the algorithm the value of $(\ell_1,(0,0))$ to $(2,\infty)$. This means that in the future iterations we have to explore in location ℓ_1 all points $(2,2),(3,3),(4,4),\ldots$. Note that the addition and subtraction of *start* at line 11 had no effect as none of the clocks after the minimum delay exceeded the maximum constant 2; should this happen the values exceeding the maximum constant get truncated to $MC+1$.

In the second iteration of the while-loop we select the location and anchor point $(\ell_1,(0,0))$, with $w=2$ and $p=\infty$, set it to $(2,2)$ in the table and mark it in bold as the selected point in the previous column. This time we have to explore two edges. First, we select the self-loop that resets the clock x and we get $start=2$ and $end=\infty$. Now we execute the lines 10 to 15 as the edge contains a reset. The for-loop will be run for the value of n from 2 to 3. The upper-bound of 3 for the for-loop follows from the fact that $MC=2$ and the maximum value of a clock that is not reset in the anchor point is 0. In the for-loop we add two successors (line 15) at the location ℓ_1 with the anchor points $(0,2)$ and $(0,3)$. Second, if we consider the edge from ℓ_1 to ℓ_2 we can see that in location ℓ_2 the anchor point $(0,0)$ is set to $(0,\infty)$.

The remaining values stored in the passed-waiting list are computed in the outlined way. We can notice that after the 7th iteration of the while-loop the set $Waiting(PW)$ is empty and the algorithm terminates. As the location ℓ_3 has not been discovered during the search, the algorithm returns false.

The correctness theorem requires a detailed technical treatment and its complete proof is given in the full version of this paper. Termination follows from the fact that newly added anchor points are computed as $(\gamma \oplus_{MC} start) - start$ or $(\gamma \oplus_{MC} n)[R:=0]$ which ensures a finite size of the passed-waiting list and that every time-dart (γ,w,p) on the list satisfies $0 \leq w \leq MC$, $w<p$, and $p \leq MC$ or $p=\infty$. Soundness proof is by a case analysis establishing a loop-invariant that every call to `AddToPW` only adds time-darts that represent reachable configurations in the bounded semantics. Finally, the completeness proof is done by induction on the length of the computation leading to a reachable configuration, taking into account the nondeterministic nature of the algorithm, the fact that $\equiv_{MC}$ is a timed bisimulation, and it makes a full analysis of the different cases for adding new time-darts present in the algorithm for its performance optimization.

Theorem 5.1 *Let A be a timed automaton and let ℓ_g be a location. Algorithm 2 terminates, and it returns true iff ℓ_g is reachable in the discrete semantics $T_{DS}(A)$.*

6 Experiments

We have conducted a number of experiments in order to test the performance of the time-dart state-space representation. The experiments were done within the project opaal [12], a model-checking framework designed explicitly for fast prototyping and testing of verification algorithms using the programming language Python. The tool implements the pseudocode of both the fully discrete (called naive in the tables) as well as the time-dart reachability algorithms based on passed-waiting list presented in Section 4.

The experiments were conducted on Intel Core 2 Duo P8600@2.4Ghz running Ubuntu linux. The verification was interrupted after five minutes or when the memory limit of 2GB RAM was exceeded (marked in the tables as OOM). The number of discovered symbolic states corresponds to the total number of calls to the function `AddToPW` (including duplicates) and the number of stored states is the size of the passed-waiting list at the termination of the algorithm. Verification times (in seconds) are highlighted in the bold font. The examples and tool implementation are available at `http://people.cs.aau.dk/~kyrke/download/timedart/timedart.tar.gz`.

Model	#	**Naive**	Discovered	Stored	**Darts**	Discovered	Stored
T55	4	**3.8**	81,062	38,906	**1.9**	34,012	3,654
T55	5	**13.0**	254,969	110,907	**5.9**	111,543	10,739
T55	6	**43.8**	727,712	297,026	**17.6**	336,527	29,378
T55	7	**95.7**	1,431,665	524,270	**32.0**	607,483	51,730
T55	8	**OOM**			**91.8**	1,740,066	136,639
T55	9	**-**	-	-	**255.7**	4,700,607	347,136
T55	10	**-**	-	-	**>300**	-	-
T125	4	**0.3**	2,609	2,050	**0.2**	198	139
T125	5	**1.6**	18,394	14,772	**0.2**	713	503
T125	6	**5.5**	61,242	48,600	**0.5**	2,916	1,769
T125	7	**20.3**	205,808	161,394	**1.4**	10,337	6,102
T125	8	**93.4**	82,4630	529,032	**4.7**	39,242	20,392
T125	9	**OOM**	-	-	**13.7**	111,438	56,191
T125	10	**-**	-	-	**34.5**	274,939	126,895
T125	11	**-**	-	-	**OOM**	-	-
T155	4	**0.4**	5,796	3,048	**0.3**	1,532	467
T155	5	**1.1**	23,454	9,740	**0.4**	4,572	1,195
T155	6	**19.9**	433,674	14,2861	**3.6**	62,771	8,859
T155	7	**28.5**	577,179	18,7857	**4.5**	74,105	10,725
T155	8	**32.3**	620,138	203,178	**4.8**	78,093	11,508
T155	9	**34.1**	626,100	205,646	**4.9**	79,111	11,753
T155	10	**60.8**	1,035,226	329,193	**7.5**	241,44	18,574
T155	11	**OOM**	-	-	**31.2**	514,959	67,592
T155	12	**-**	-	-	**37.7**	608,974	80,634
T155	13	**-**	-	-	**105.7**	1,684,525	205,087
T155	14	**-**		-	**158.3**	2,316,474	284,859
T155	15	**-**	-	-	**164.5**	2,409,417	298,288
T155	16	**-**	-	-	**OOM**	-	-

Figure 5: Results for three different TGS scaled by the number of tasks

6.1 Task Graph Scheduling

The task graph scheduling problem (TGS) is the problem of finding a feasible schedule for a number of parallel tasks with given precedence constraints and processing times on a fixed number of homogeneous processors [17]. The chosen task graphs for two processors were taken from the benchmark [22] such that several scheduling problems with different degree of concurrency are included. The models are scaled by the number of tasks in the order given by the benchmark and the verification query performed a full state-space search. The experimental results are displayed in Figure 5. The data confirm that the time-dart verification technique saves both the number of stored/discovered states and noticeably improves the verification speed, in particular in the model T155.

6.2 Bridge Crossing Vikings

The bridge crossing Vikings is a slightly modified version of the standard planning problem available in the official distribution of UPPAAL; we only eliminated the used integer variables that are not supported in our opaal implementation and are simulated by new locations. The query searched the whole state-

#	**Naive**	Discovered	Stored	**Darts**	Discovered	Stored
2	**0.2**	295	152	**0.1**	87	46
3	**0.2**	2,614	1,263	**0.2**	754	336
4	**0.8**	16,114	7,588	**0.5**	4,902	1,759
5	**4.9**	90,743	42,294	**2.5**	29,144	8,308
6	**33.2**	501,958	235,635	**13.4**	165,535	38,367
7	**OOM**	-	-	**74.8**	900,439	177,807
8	**-**	-	-	>**300**	-	-

Figure 6: Results for bridge crossing scaled by the number of Vikings

#	**Naive**	Discovered	Stored	**Darts**	Discovered	Stored
1	**0.2**	173	139	**0.2**	21	15
2	**1.5**	16684	11042	**0.3**	1140	647
3	**OOM**	-	-	**4.6**	40671	21721
4	**-**	-	-	**OOM**	-	-

Figure 7: Results for train level crossing scaled by the number of trains

space. Verification results are given in Figure 6. The performance of the time-dart algorithm is again better than the full discretization, even though in this case the constants in the model are relatively small (proportional to the number of Vikings), meaning that the potential of time-darts is not fully exploited.

6.3 Train Level Crossing

In train level crossing we consider auto-generated timed automata templates constructed via automatic translation [11] from timed-arc Petri net model of a train level-crossing example. The auto-generated timed automata were produced by the tool TAPAAL [13] and have a rather complex structure that human modelers normally never design and hence we can test the potential of the discrete-time engine also for the models translated from other time-dependent formalisms. The query we asked searches the whole state-space. We list the results in Figures 7 and the experiment demonstrates again the advantage of the time-dart verification method.

6.4 Fischer's Protocol

The discrete-time techniques are sensitive to the size of the constants present in the model. We have therefore scaled our next experiment by the size of the maximal constant (MC) that appears in the model in order to demonstrate the main advantage of the time-dart algorithm. For this we use the well known Fischer's protocol for ensuring a mutual exclusion between two or more parallel processes [18]. It is a standard model for testing the performance of verification tools; we replaced one open interval in the model with a closed one such that mutual exclusion is still guaranteed. The concrete version of the protocol we verified was created by a translation from timed-arc Petri net model of the protocol [1] available as a demo example in the tool TAPAAL [13]. We searched the whole state-space and the results are summarized in Figure 8. It is clear that time-darts are superior w.r.t. the scaling of the constants in the model, allowing us to verify (within the given limit of 300 seconds) models where the maximum constant is 66, opposed to only 18 when the full discretization is used.

MC	**Naive**	Discovered	Stored	**Darts**	Discovered	Stored
3	**3.1**	36,774	25,882	**1.1**	9,464	6,238
4	**4.5**	53,655	38,570	**1.4**	14,341	8,725
5	**6.2**	74,415	54,513	**1.8**	20,226	11,548
9	**17.8**	202,965	157,555	**3.8**	53,846	26,200
15	**64.5**	569,280	466,888	**10.4**	133,796	58,258
18	**111.8**	850,164	710,857	**14.2**	187,595	78,823
19	**OOM**	-	-	**15.7**	207,544	86,350
25	**-**	-	-	**26.1**	348,406	138,568
38	**-**	-	-	**61.3**	778,095	293,203
50	**-**	-	-	**114.7**	1,325,931	486,343
66	**-**	-	-	**217.2**	2,214,846	795,808

Figure 8: Experimental results for Fischer's mutual exclusion protocol

7 Conclusion

We have introduced a new data structure of time-darts in order to represent the reachable state-space of closed timed automata models. We showed on a number of experiments that our time-dart reachability algorithm achieves a consistently better performance than the explicit search algorithm, improving both the speed and memory requirements. This is obvious in particular on models with larger constants (as demonstrated in the Fischer's experiment or T155 task graph) where time-darts provide a compact representation of the delay successors and considerably improve both time and memory.

The algorithms were implemented in the interpreted language Python without any further optimizations techniques like partial order and symmetry reductions and advanced extrapolation techniques and with only one global maximum constant. This does not allow us to compare its performance directly with the state-of-the-art optimized tools for real-time systems.

An advantage of time-darts and explicit state-space methods in general is that it is relatively easy to extend them with additional modelling features like clock invariants and diagonal guards. In our future work we will implement the time-dart algorithm in `C++` with additional optimizations (e.g. considering local constants instead of the global ones) and we shall also consider the verification of liveness properties. It is clear that for large enough constants the DBM-search engine will always combat the explicit methods (see [19]); our technique can be so seen as a practical alternative to the DBM-engines on the subset of models that for example use counting features (like in our introductory example) and where DBM state-space representation explodes even for models with small constants. Another line of research will focus on further optimizations of the time-dart technique by considering federations of time-darts so that the data structure becomes even less sensitive to the scaling of the constants.

Acknowledgements. We would like to thank the anonymous reviewers for their comments.

References

[1] P.A. Abdulla & A. Nylén (2001): *Timed Petri Nets and BQOs*. In: *Proceedings of the 22nd International Conference on Application and Theory of Petri Nets (ICATPN'01), LNCS* 2075, Springer-Verlag, pp. 53–70, doi:10.1007/3-540-45740-2_5.

[2] R. Alur & D. Dill (1994): *A Theory of Timed Automata*. *Theoretical Computer Science (TCS)* 126(2), pp. 183–235, doi:10.1016/0304-3975(94)90010-8.

[3] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli & A. Rasse (1997): *Data-Structures for the Verification of Timed Automata*. In Oded Maler, editor: *HART*, *LNCS* 1201, Springer, pp. 346–360, doi:10.1007/BFb0014737.

[4] G. Behrmann, A. David & K.G. Larsen (2004): *A Tutorial on* UPPAAL. In: *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*, *LNCS* 3185, Springer-Verlag, pp. 200–236, doi:10.1007/978-3-540-30080-9_7.

[5] B. Berthomieu & M. Menasche (1983): *An Enumerative Approach for Analyzing Time Petri Nets*. In: *IFIP Congress*, pp. 41–46.

[6] D. Beyer, C. Lewerentz & A. Noack (2003): *Rabbit: A Tool for BDD-Based Verification of Real-Time Systems*. In Warren A. Hunt Jr. & Fabio Somenzi, editors: *CAV*, *LNCS* 2725, Springer, pp. 122–125, doi:10.1007/978-3-540-45069-6_13.

[7] D. Beyer & A. Noack (2003): *Can Decision Diagrams Overcome State Space Explosion in Real-Time Verification?* In Hartmut König, Monika Heiner & Adam Wolisz, editors: *FORTE*, *LNCS* 2767, Springer, pp. 193–208, doi:10.1007/978-3-540-39979-7_13.

[8] P. Bouyer (2003): *Untameable Timed Automata!* In Helmut Alt & Michel Habib, editors: *STACS*, *LNCS* 2607, Springer, pp. 620–631, doi:10.1007/3-540-36494-3_54.

[9] M. Bozga, S. Graf & L. Mounier (2002): *IF-2.0: A Validation Environment for Component-Based Real-Time Systems*. In Ed Brinksma & Kim Guldstrand Larsen, editors: *CAV*, *LNCS* 2404, Springer, pp. 343–348, doi:10.1007/3-540-45657-0_26.

[10] M. Bozga, O. Maler & S. Tripakis (1999): *Efficient Verification of Timed Automata Using Dense and Discrete Time Semantics*. In: *Proceedings of Correct Hardware Design and Verification Methods (CHARME'99)*, *LNCS* 1703, Springer, pp. 125–141, doi:10.1007/3-540-48153-2_11.

[11] J. Byg, K.Y. Jørgensen & J. Srba (2009): *An Efficient Translation of Timed-Arc Petri Nets to Networks of Timed Automata*. In: *Proc. of the 11th International Conf. on Formal Engineering Methods (ICFEM'09)*, *LNCS* 5885, Springer-Verlag, pp. 698–716, doi:10.1007/978-3-642-10373-5_36.

[12] A.E. Dalsgaard, R.R. Hansen, K.Y. Jørgensen, K.G. Larsen, M.Chr. Olesen, P. Olsen & J. Srba (2011): *opaal: A Lattice Model Checker*. In: *Proceedings of the 3rd NASA Formal Methods Symposium (NFM'11)*, *LNCS* 6617, Springer-Verlag, pp. 487–493, doi:10.1007/978-3-642-20398-5_37.

[13] A. David, L. Jacobsen, M. Jacobsen, K.Y. Jørgensen, M.H. Møller & J. Srba (2012): *TAPAAL 2.0: Integrated Development Environment for Timed-Arc Petri Nets*. In: *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, *LNCS* 7214, Springer-Verlag, pp. 492–497, doi:10.1007/978-3-642-28756-5_36.

[14] C. Daws, A. Olivero, S. Tripakis & S. Yovine (1996): *The Tool* KRONOS. In: *Proc. Hybrid Systems III: Verification and Control (1995)*, *LNCS* 1066, Springer-Verlag, pp. 208–219, doi:10.1007/BFb0020947.

[15] D.L. Dill (1989): *Timing Assumptions and Verification of Finite-State Concurrent Systems*. In Joseph Sifakis, editor: *Automatic Verification Methods for Finite State Systems*, *LNCS* 407, Springer, pp. 197–212, doi:10.1007/3-540-52148-8_17.

[16] P.-A. Hsiung, F. Wang & R.-Ch. Chen (2000): *On the verification of Wireless Transaction Protocol using SGM and RED*. In: *RTCSA*, IEEE Computer Society, pp. 379–383, doi:10.1109/RTCSA.2000.896414.

[17] Y.-K. Kwok & I. Ahmad (1999): *Benchmarking and Comparison of the Task Graph Scheduling Algorithms*. *Journal of Parallel and Distributed Computing* 59(3), pp. 381 – 422, doi:10.1006/jpdc.1999.1578.

[18] L. Lamport (1987): *A Fast Mutual Exclusion Algorithm*. *ACM Transactions on Computer Systems* 5(1), pp. 1–11, doi:10.1145/7351.7352.

[19] L. Lamport (2005): *Real-Time Model Checking Is Really Simple*. In: *CHARME, LNCS* 3725, Springer, pp. 162–175, doi:10.1007/11560548_14.

[20] K.G. Larsen & Y. Wang (1997): *Time-Abstracted Bisimulation: Implicit Specifications and Decidability*. *Information and Computation* 134(2), pp. 75 – 101, doi:10.1006/inco.1997.2623.

[21] Y. Liu, J. Sun & J.S. Dong (2008): *An Analyzer for Extended Compositional Process Algebras*. In: *ICSE Companion*, ACM, pp. 919–920, doi:10.1145/1370175.1370187.

[22] Kasahara Laboratory at Waseda University: *Standard Task Graph Set*. Http://www.kasahara.elec.waseda.ac.jp/schedule/.

[23] J. Zhao, X. Li & G. Zheng (2005): *A quadratic-time DBM-based successor algorithm for checking timed automata*. *Information Processing Letters* 96(3), pp. 101–105, doi:10.1016/j.ipl.2005.05.027.