Hypervolume Approximation in Timed Automata Model Checking

Víctor Braberman[1,2], Jorge Lucángeli Obes[1], Alfredo Olivero[3], and Fernando Schapachnik[1]

[1] Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina *
{vbraber,lucangeli,fschapac}@dc.uba.ar
[2] CONICET
[3] Facultad de Ingeniería y Cs. Exactas,
Universidad Argentina de la Empresa, Buenos Aires, Argentina **
aolivero@uade.edu.ar

Abstract. Difference Bound Matrices (DBMs) are the most commonly used data structure for model checking timed automata. Since long they are being used in successful tools like KRONOS or UPPAAL.
As DBMs represent convex polyhedra in an n-dimensional space, this paper explores the idea of using its hypervolume as the basis for two optimization techniques. One of them is very simple to implement. The other, an improvement over the first, requires more involved programming. Each of them saves verification time (up to 19% in our case studies), with a modest increase of memory requirements. Their impact differs among the different case studies but, as they can be combined, there is no need to choose a priori.

1 Introduction

In current days timed systems are both pervasive and critical, ranging from embedded and PDAs to plant and flight controllers. Their complexity is ever increasing so unassisted ways of verifying them make sense. Automated methods, however, are known to suffer from scalability problems: their time and memory requirements grow exponentially as systems increase in size. This is why any technique that can palliate such problems can be useful.

We focus on model checking of timed automata [1], an extension of finite automata with the possibility to model dense time. Many model checking property validation problems can be reduced to forward reachability [2], that is, an exploration of the model starting by its initial state and trying to find a state tagged with a particular boolean property, called it p. The basic (conceptual) procedure is straight forward: insert the initial state in the *Pending* queue and initialize an empty *Visited* set. Then, while *Pending* is not empty, take its next

* Partially supported by UBACyT 2004 X020, PICT 32440 and 11738.
** Partially supported by STIC-AmSud project Tapioca.

state and check whether the property p holds for it. If it does, finish with a "YES" result, otherwise, put it in *Visited*, compute its (timed) successors (one for each outgoing transition). To ensure termination, before putting them in *Pending*, it should be checked that they are not *included* in any other state from *Visited*. In an untimed exploration the check would be simpler: is the same state already *present* in *Visited*? In the timed (symbolic) framework, clocks values conform multi-dimensional polyhedra. As such, if a new state is included in an already explored one there is no point in revisiting it, even if it is not exactly equal. Some more detail on the reachability procedure is presented in Fig. 1, on page 5.

From the outline of the reachability algorithm it should be clear that the inclusion operation between polyhedra is a critical one, being responsible for an important fraction of the total running time. As such, finding ways to speed it up is always a good idea.

Although many optimizations have been developed (see, for example, [3–9]), the inclusion checking operation, although linear in many cases, has a worst case of $O(n^2)$ where n is the number of clocks in the system. In this article we focus on the hypervolume of the above mentioned polyhedra. If it could be easily computed, an $O(1)$ check could be performed prior to the expensive inclusion algorithm: if $\mathit{hypervolume}(A) > \mathit{hypervolume}(B)$ it is impossible for A to be included in B. Of course, if the hypervolume of A is less or equal to B's, then A can be included or not, and the full check needs to be performed.

Computing exact hypervolume for n-dimensional polyhedra is a known hard problem, but luckily with the traditional data structures used for timed model checking (Difference Bound Matrices) an approximation can be computed very cheaply. This approximation is safe, in the sense that the observations from the previous paragraph still hold (Theorem 1 expresses the property formally). A related idea might be comparing the bounding box in each dimension. It has the advantage of being less suceptible to overflow (see Section 3.2), but that would require $O(n)$ extra storage and comparisons, instead of $O(1)$.

We take advantage of the hypervolume approximation in two ways: firstly, inclusion checks are avoided in the sense of the preceding paragraphs. Secondly, the *Visited* set is ordered by (approximate) hypervolume. In this way, it is not necessary to check a new state against the complete set, but only against the states that have a bigger (approximate) hypervolume, saving an important number of inclusion checks (see Section 4).

Neither technique increments the number of visited states. They can be used independently and obtain interesting speedups. Some models benefit more with one of them, and some with the others, with acceleration of up to 19% in our experiments. They can also be combined, although the speedups are not additive, because there is some mutual cancelation. The good news is that there is no need to speculate on which one to use, because using both is generally as good as using the best of them.

Further, we explain why it doesn't makes sense to order also *Pending* by its hypervolume.

Although the idea of approximations is not new (see, for instance the convex-hull abstraction at [10]), they generally lead to approximated answers also (i.e., they might state with certainty that the property is not reached, or that it might –or not– be reached). Ours, however, gives exact answers.

The rest of the article is structured as follows: Section 2 gives the basic background on the timed automata formalism and related definitions. The following section presents the details of the proposed techniques. In Section 4 empirical evidence, based on known case studies from the literature, is presented. After that, Section 5 discusses future work and concludes the article.

2 Background

Timed automata [1] are a widely used formalism to model and analyze timed systems. They are supported by several tools such as KRONOS [11] or UPPAAL [12]. Their semantics are based on labeled state-transition systems and time-divergent runs over them. Here we present their basic notions and refer the reader to [1, 11] for a complete formal presentation.

Definition 1 (Timed automaton). *A* timed automaton *(TA) is a tuple* $\mathcal{A} = \langle L, X, \Sigma, E, I, \mathsf{l}_0 \rangle$*, where* L *is a finite set of locations,* X *is a finite set of clocks (non-negative real variables),* Σ *is a set of labels,* E *is a finite set of edges,* $I : L \stackrel{tot}{\rightarrow} \Psi_X$ *is a total function associating to each location a clock constraint called the location's invariant, and* $\mathsf{l}_0 \in L$ *is the initial location. Each edge in* E *is a tuple* $\langle \mathsf{l}, a, \psi, \alpha, \mathsf{l}' \rangle$*, where* $\mathsf{l} \in L$ *is the source location,* $\mathsf{l}' \in L$ *is the target location,* $a \in \Sigma$ *is the label,* $\psi \in \Psi_X$ *is the guard,* $\alpha \subseteq X$ *is the set of clocks reset at the edge. The set of clock constraints* Ψ_X *for a set of clocks* X *is defined according to the following grammar:* $\Psi_X \ni \psi ::= x \sim c | \psi \wedge \psi$*, where* $x \in X, \sim \in \{<, \leq, =, >, \geq\}$ *(although invariants restrict* $\sim$ *to* $\{<, \leq\}$*) and* $c \in \mathbb{N}$.

Usually, a TA $\mathcal{A}$ has an associated mapping $Pr : L \mapsto 2^{Props}$ which assigns to each location a subset of propositional variables from the set $Props$.

The parallel composition $\mathcal{A}_1 \parallel \mathcal{A}_2$ of TAs $\mathcal{A}_1$ and $\mathcal{A}_2$ is defined using a label-synchronized product of automata [1, 11]. At any time, the *state* of the system is determined by the location and the values of clocks, which must satisfy the location invariant. The system can evolve in two different ways: either an enabled transition is taken, changing the location and (maybe) resetting some clocks while the others keep their values unaltered (a discrete step), or it may let some amount of time pass (a timed step). In the last case the system remains in the same location and all clocks increase according to the elapsed time, while still satisfying the location invariant.

To model a complex system, an automaton can be expressed as the parallel composition of the automata representing each component. A location of the obtained automaton, called *global location* or *composed automata node*, is a tuple consisting of a location of each component. Similarly, a state of the automaton (*global state*) is a global location plus the values of all clocks. Such sets of automata are usually called *a network*.

Definition 2 (Clock valuations). *A* valuation *is a total function from the clock set X into $\mathbb{R}^+$ (i.e., the reading of each clock in a particular moment). The valuation set over X, $\mathcal{V}_X$ is defined as $\{v : X \stackrel{tot}{\rightarrow} \mathbb{R}^+\}$. For each $v \in \mathcal{V}_X$ and $\delta \in \mathbb{R}^+$, $v+\delta$ stands for the valuation defined as $(\forall x \in X)(v+\delta)(x) = v(x)+\delta$.*

To deal with infinite state manipulation, convex sets of clock valuations are symbolically represented as conjunctions of inequalities (e.g., $1 \leq x \leq 5 \wedge x - y > 8$). Each of these conjunction represents a convex set of points, and is referred to as a *zone*. A data structure called Difference Bound Matrices (DBM) [13] is typically used to manipulate such kind of information.

Definition 3 (Difference Bound Matrices). *DBMs are $(n+1)^2$ matrices, where n is the number of clocks in the system. Diagonal cells are void, but all the others contain a tuple $\langle \prec, c \rangle$ called* bound*, where $\prec \in \{<, \leq\}$ and $c \in \mathbb{Z} \cup \{\infty\}$. If in a DBM M cell (i,j) (noted $M[i,j]$) contains $\langle \prec, c \rangle$ it means that $x_i - x_j \prec c$, where x_i and x_j are the i-th and j-th clocks in the system (counting 0 as a special clock, used to express $x_i \prec c$ as $x_i - 0 \prec c$). The only valid use of ∞ in a cell is to mean that the difference of two clocks is unbounded. I.e., $x_i - x_j < \infty$.*

Although a zone *can be represented by a* DBM*, we will abuse both terms using them interchangeably when there is no confusion.*

During the reachability algorithm (see Fig. 1), states are represented by a pair (l, z) where l is a location and z a timed zone. Given a state, the successor set is computed by the $suc_\triangleright$ operator, which is defined as follows:

$$suc_\triangleright(\mathsf{l}, z) = \{(\mathsf{l}', z') / \langle \mathsf{l}, a, \psi, \alpha, \mathsf{l}' \rangle \in E \wedge z' = suc_\tau(reset_\alpha(z \cap \psi)) \cap I(\mathsf{l}')\}$$

Where $reset_\alpha$ means putting the clocks in α to zero and $suc_\tau(\psi)$ means replacing the constraints of the form $x \prec c$ by $x < \infty$ while leaving the rest intact. To avoid clutter, calls to cf, the *canonical form* function, are not shown. It expresses every constraint as tight as possible (see, for instance, [14], for the details) and should be called after intersection, reset and suc_τ.

Not every constraint needs to be present for all the operations. Actually, a reduced version of the constraint systems can be used for most operations, thus saving memory [15]. In practice, most tools use a variation of DBMs, called *Minimal Constraint Representation*, which employs that idea. As these DBMs are sparse, they are not stored like proper matrices, but as a linked list of constraints, in order to save space. This minimal representation sets the context of this article.

The classical algorithm for inclusion checking in such data structure is shown in Fig. 2. It expects the first zone to be in canonical form and the second one to be *minimized*, i.e., as the above mentioned Minimal Constraint Representation. If a different representation was chosen for the DBMs, the algorithms would change importantly.

This algorithm is easily generalized to check if a zone is included in a set of zones (IsIncludedInSet).

```
1: function ForwardReach(Property φ)
2:     Visited ← ∅
3:     Pending ← {(l_0, z_0)}
4:     while Pending ≠ ∅ do
5:         (l, z) ← next(Pending)
6:         Add((l, z), Visited)
7:         if (l, z) ⊨ φ then return YES
8:         end if
9:         Z_l ← ⋃_{(l,z')∈(Visited∪Pending)} z'
10:        for (l', z') ∈ suc_▷(l, z) do
11:            if ¬ IsIncludedInSet(z', Z_l) then
12:                Add((l', z'), Pending)
13:            end if
14:            if ∃z'' ∈ Pending_l/ IsIncluded(z'', z') then
15:                Delete((l', z''), Pending)
16:            end if
17:        end for
18:    end while
19:    return NO
20: end function
```

Fig. 1. Forward reachability algorithm.

```
1: function IsIncluded(DBM z_1, z_2)
2:     // Require: z_1 is in canonical form, z_2 is minimized.
3:     for i = 0 to #clocks do
4:         for j = 0 to #clocks do
5:             if i ≠ j ∧ ¬(z_1[i, j] ≤ z_2[i, j]) then
6:                 return NO
7:             end if
8:         end for
9:     end for
10:    return YES
11: end function
```

Fig. 2. Inclusion checking algorithm.

3 Using the Hypervolume

3.1 Avoiding Checks by Comparing Hypervolume Approximations

Let's start by defining which approximation to the hypervolume we use. The idea is to compute the hypervolume of the smallest hypercube containing the polyhedron defined by the clocks' values.

Definition 4 (Hypervolume approximation). *Let z be a DBM with n clocks. The hypervolume $HVol(z)$ is defined as $\Pi_{1\leq i\leq n}(const(z[i,0]) - |const(z[0,i])|)$, where $const(x \prec c) = c$.*

Note that for the purpose of the hypervolume approximation there is no need to differentiate among $\prec$, and that $const(z[0,i])$, the lower bound, is negative and thus the need for modulus (i.e., $x_1 > 7$ is expressed in DBMs as $z[0,1] = (<,-7)$).

If $z[i,0]$ is $< \infty$, use the maximum constant against which the clock is compared in the system, plus one (cf. [3]).

As can be seen from its definition, the computation of $HVol(z)$ is linear in the number of clocks. However, there is no need to recompute it after every operation that manipulates the zone. It only needs to be calculated as the last step of suc_τ, previous to the inclusion check. The overhead is mild, as the immediate previous operation is usually the transformation of the zone to its canonical form, which is $O(n^3)$.

The approximation is safe, as stated by Theorem 1.

Theorem 1. *If $HVol(z_1) > HVol(z_2)$ then $z_1 \not\subseteq z_2$.*

Proof. Let's assume $HVol(z_1) > HVol(z_2) \wedge z_1 \subseteq z_2$. As the operation IsIncluded() is sound and complete w.r.t. $\subseteq$, it means that $(\forall i,j)\ 0 \leq i,j \leq n$, $i \neq j \implies const(z_1[i,j]) \leq const(z_2[i,j])$. Then, as $HVol(z_1)$ and $HVol(z_2)$ are both products of the same quantity of positive terms, $HVol(z_1)$ is the product of positive numbers which are all less or equal to the corresponding ones in $HVol(z_2)$, contradicting the possibility of $HVol(z_1)$ being greater than $HVol(z_2)$.

Although hypervolume approximation resembles the convex-hull abstraction [10], there is a very important difference. Ours is an exact technique, meaning that the answer to the reachability question is responded *yes* or *no* with certainty. In convex-hull, on the other hand, zones corresponding to the same location are joined to their convex-hull over approximation. If the property is not reached, then the answer is precise, but if it is, then the answer is *maybe*.

3.2 Implementations Notes

Care must be taken when computing $HVol(z)$ as to not overflow the capacity of the container integer variable. Which type of integer variable to use for storing the $HVol$ of a zone has consequences in both memory overhead and precision.

The lower the number of bits reserved for the *HVol*, the sooner it will saturate, not allowing to avoid some inclusion checks. On the other hand, if too many bits are used, the memory overhead can be considerable. The exact hypervolume would require $O(\log \Pi_{1 \leq i \leq n} |C_i|)$, where C_i is the biggest constant compared against the x_i clock in the system. As with many others time-vs-memory trade offs, experimentation should be used to find a convenient balance.

Note that the overhead depends on the implementation of DBMs. If a proper matrix is used, a `long int`, which is 8 bytes on 32 bits machines, usually provides a good amount of check saving and requires little space compared to the DBM itself. On more sophisticated representations, which leverage on Minimal Constraint Representation [15] to use a variant of linked lists of bounds, the overhead can be variable. Our experimentation shows an average of 2% extra memory. Although Section 4 shows the experimental results, let's suppose we are dealing with a system with ten clocks (a conservative assumption). A proper DBM will have a hundred bounds. Being conservative, assume that the minimal representation has approx. 30% of the bounds. For 30 bounds and 12 bits per bound (usually enough to represent constants on the hundreds), the 360 bits can be packed in 12 `long int`. For these figures, an extra `long int` represents less than 9% of extra memory.

3.3 Sorting *Visited*

As can be seen in Fig. 1, when a new state (l, z) is found, the $Visited_{\mathsf{l}}$ set (i.e., the restriction of *Visited* [4] to states that have l as location) has to be fully explored, comparing the new zone against all the ones contained in that set.

If each zone in *Visited* has an *HVol*, then it can be turned into a priority queue, where the zones with greater *HVol* are checked first. Let z_n be the zone of the new state and z_q be the zone of $next(Visited_{\mathsf{l}})$. $HVol(z_q)$ is bigger or equal than $HVol(z)$ for every $z \in Visited_{\mathsf{l}}$. So, if $HVol(z_n)$ is greater than $HVol(z_q)$, then, because of Theorem 1, z_n is not included neither in z_q, nor in the rest of $Visited_{\mathsf{l}}$. In consequence, there is no need to continue checking, thus reducing the number of inclusion checks performed, as can be seen in the experimentation.

A problem of implementing the above mentioned technique with a traditional priority queue is that iteration is done by the successive elimination of *next* elements, thus requiring to re-insert them afterwards. For a queue with k elements, the total cost with, e.g., a heap, is $O(k \log k)$. The memory management involved in removing and adding elements can make the constants considerable, importing a noticeable overhead that can easily counter the gain from the inclusion checks avoided.

To overcome this problem, we chose a van Emde Boas tree [16], which permits nondestructive iteration. It is also convenient from a theoretical point of view: visiting the first k' elements costs $O(k' \log \log k)$. Actually van Emde Boas

[4] Although we use a unified storage of $Visited \cup Pending$ as proposed in [4], separate indexes allows us to traverse them independently.

trees provide all of their operations in $O(\log \log k)$, at the penalty of only supporting integer numbers from a fixed interval as keys. Our experience with it was that although it can be quite difficult to implement, it provides very good performance.

The resulting algorithm is shown in Fig. 3.

```
1: function IsIncludedInOrdederSet(DBM z, DBM ordered set Z)
2:     for all z' ∈ Z do
3:         if HVol(z) >HVol(z') then
4:             return NO
5:         else if IsIncluded(z, z') then
6:             return YES
7:         end if
8:     end for
9:     return NO
10: end function
```

Fig. 3. Inclusion checking algorithm (zone in ordered set of zones).

Having $Visited_{\mathsf{l}}$ sorted by *HVol* is not only useful when the new zone is not included. If it is, as the bigger zones are checked first, there is a good chance that the detection occurs earlier (for instance, universal zones are always the first to be checked, whereas in a traditional implementation of $Visited_{\mathsf{l}}$ they could be "buried" deep into the set).

It should be noted that it makes no sense to check if a new state (l', z') includes one (l', z'') in *Visited* (contrary to *Pending* which is checked in line 14 of Fig. 1), because in case it is, (l', z'') still can't be removed, as it might be part of a trace to the target state.

Section 4 shows the time and memory results for the implementation of the above mentioned techniques in the model checker Zeus. Before that, in Section 3.4, we explore the question of whether it is also worth sorting *Pending*.

3.4 Sorting *Pending*. Worth it?

At first sight the idea of sorting *Pending* by decreasing *HVol* sounds appealing: suppose that both z and z' are in *Pending*, and that $hypervolume(z) > hypervolume(z')$. As suc_τ is monotonic, it makes sense to compute $\hat{z} = suc_\tau(z)$ before $\hat{z'} = suc_\tau(z')$ because $\hat{z}$ is bigger that $\hat{z'}$. Chances are that $\hat{z'}$ might be included in $\hat{z}$, avoiding the exploration of a new zone.

An interesting aspect of sorting *Pending* that way is that it *seems* conservative. It *seems* that in case it didn't improve things, they will not get worse, i.e., no more zones will be generated.

To test these ideas, we changed the *Pending* FIFO queue into a priority queue sorted by *HVol*, and implemented it also as a van Emde Boas tree. Unfortunately, results were not positive, leading to more zones found (and thus more total time

and memory) for many case studies, even ones that generated the complete state space.

The problem is that when locations are considered into the equation, things get more complicated than the intuition presented in previous paragraphs. Suppose there are (l_1, z_1) and (l_2, z_2) in *Pending* (in that order), with z_1 being $x < 10$ and z_2 being $x < 12$. Also, assume that both l_1 and l_2 have a transition to l_3 (which has a *true* invariant), but the second with an $x > 5$ guard while the first imposes no restriction.

In a FIFO exploration (l_1, z_1) will be explored first, leading to the discovery of (l_3, z_3), with z_3 being $x < \infty$. When (l_2, z_2) is expanded, the new state, with zone $z_3' = 5 < x < \infty$, will already be included. On the other hand, if *Pending* was ordered by hypervolume, (l_2, z_2) would be explored first, leading to the discovery of (l_3, z_3'), which in turn will be put into *Pending* and explored before (l_1, z_1). When this state gets its turn, (l_3, z_3) will be generated, but the inclusion check will fail, thus creating a new zone to be explored.

Next section shows the experimental evidence that backs the claims made in Sections 3.1 and 3.3.

4 Experimental Results

To validate the proposed techniques, we incorporated them into a monoprocessor version of the ZEUS distributed model checker [17] and ran a series of experiments against well known case studies from the literature. For some of them, a reduced version (created with OBSSLICE [8], a safe model reducer) was also used and is primed in the tables. Each of them comprises two versions: the one where the property is reached (true) and the unreachable one (false). The difference is usually a changed delay. The examples used are:

1. *RCS5*, the *Railroad Crossing System* inspired in [18] with 5 trains.
2. *Pipe6*, end-to-end signal propagation in a pipe-line of sporadic processes that forward a signal emitted by a quasi periodic source, with 6 stages.
3. *FDDI8*, an extension of the FDDI token Model ring protocol where the observer monitors the time the token takes to return to a given station.
4. *Conveyor6ABC*, *Conveyor Belt* [17] (with 6 stages and 3 objects)
5. *MinePump*, a design of a fault-detection mechanism for a distributed mine-drainage controller [19].

Table 1 summarizes the sizes of the examples used in this article. The experiments were run on a Intel Pentium IV 3.0 GHz processor with 2 GB of RAM, running the Linux 2.6 operating system.

Table 2 shows the results obtained with each method independently and Table 3 the combination of both. The first columns report the time, memory and number of inclusion checks for the standard version, and then the time and number of inclusion checks for each optimization, with the percentage in parenthesis (negative values for saving, positive for increase). Memory is not reported for the first optimization because the overhead was always less than 2%

Example	Components	Clocks	Reachable locations
MinePump	10	10	1428
RCS5	8	8	1617
Conveyor6ABC	11	12	31443
FDDI8	15	23	4608

Table 1. Examples sizes.

(consistently with the reasoning already mentioned in Section 3.1). The overhead of the second comes from the van Emde Boas tree, which requires many pointers.

It should be noted that in the combined method, although not direct check is saved by *HVol*, many inverse ones (see line 14 of Fig. 1) can still be avoided.

Example	Standard			With *HVol*		*Visited* priority queue		
	Time (secs)	Mem (MB)	#checks ($\times 10^6$)	Time (secs)	#checks ($\times 10^6$)	Time (secs)	Mem (MB)	#checks ($\times 10^6$)
MinePump'	46	7	10.5	43	8.9	43	7.8	9.2
true				(-7%)	(-15%)	(-7%)	(+11%)	(-12%)
MinePump'	527	19	206.0	475	173.8	465	21.2	180.6
false				(-10%)	(-16%)	(-12%)	(+12%)	(-12%)
MinePump	2807	56	1124.3	2542	969.7	2444	62.1	959.3
true				(-9%)	(-14%)	(-13%)	(+11%)	(-15%)
MinePump	20580	152	4348.9	18353	3051.1	17425	163.2	3030.2
false				(-11%)	(-30%)	(-15%)	(+7%)	(-30%)
RCS5	31	7	10.2	23	5.5	28	7.7	9.1
true				(-26%)	(-46%)	(-10%)	(+10%)	(-11%)
RCS5	1039	34	341.3	952	286.4	955	34.6	311.7
false				(-8%)	(-16%)	(-8%)	(+2%)	(-9%)
Conveyor6ABC	1385	183	163.6	1280	104.8	1338	198.0	138.5
true				(-8%)	(-36%)	(-4%)	(+8%)	(-15%)
Conveyor6ABC	4142	280	998.8	3368	567.1	3600	306.4	762.7
false				(-19%)	(-43%)	(-13%)	(+9%)	(-23%)
FDDI8	488	20	0.1	488	0.1	486	21.0	0.1
true				(0%)	(0%)	(-<1%)[5]	(+5%)	(-<1%)
FDDI8	7642	120	18.2	7645	18.2	7629	122.0	18.2
false				(+<1%)	(0%)	(-<1%)	(+2%)	(-<1%)

Table 2. Results obtained for each method.

As can be seen in Table 2, *HVol* features time savings of up to 19% (not counting *RCS5* true, because it already takes a very short time and is only presented for completeness), and sorting *Visited* of up to 15%. The first one has neglectable memory overhead, while the second uses around a 10-12%. There

[5] Marginal improvements, not seen in figures because of rounding.

Example	Both methods		
	Time (secs)	Mem (MB)	#checks ($\times 10^6$)
MinePump' true	43 (-7%)	7.8 (+11%)	8.3 (-21%)
MinePump' false	465 (-12%)	21.2 (+12%)	181.3 (-12%)
MinePump true	2436 (-13%)	62.1 (+11%)	955.7 (-15%)
MinePump false	17388 (-16%)	163.2 (+7%)	2348.4 (-46%)
RCS5 true	23 (-26%)	7.5 (+10%)	8.0 (-22%)
RCS5 false	926 (-11%)	34.6 (+2%)	286.7 (-16%)
Conveyor6ABC true	1295 (-6%)	198.0 (+8%)	115.9 (-29%)
Conveyor6ABC false	3470 (-16%)	306.4 (+9%)	569.3 (-43%)
FDDI8 true	487 (-<1%)	21.0 (+5%)	0.1 (-<1%)
FDDI8 false	7652 (+<1%)	122.0 (+2%)	18.2 (-<1%)

Table 3. Results obtained with both methods combined.

is no direct relationship between the number of checks saved and the speedup. This is because each case studies has different sized matrices and for each check saved, the number of cell matrices to be compared differs.

FDDI8 is an interesting case study because, as no inclusion check could be avoided with *HVol*, the difference in times measures pure overhead, showing that the method is very light.

Table 2 also shows that different case studies benefit the most from different techniques. If memory is a premium, clearly *HVol* is convenient, but if some memory can be spent in order to obtain earlier results, then a decision should be made among the two. Also, they can be combined. As shown in Table 3, with the exception of *FDDI8*, the combined method is never worse than the worse of the optimizations (see for instance *Conveyor6ABC*). Sometimes it is as good as the best of them (*MinePump'*, *MinePump* true, *RCS5* true) and sometimes better (*MinePump* false, *RCS5* false).

Both the second and the combined method have indeed a memory overhead that seems, in terms of percentages, on the same order of magnitude as the time saved. However, in cases like *MinePump* false on Table 3, it can be seen that the 16% saving of time translates to almost one hour of almost six, at the cost of 7% more memory, which only amounts to less than 12 extra MB of RAM.

5 Conclusions and Future Work

In this article we presented two techniques that contribute to speed up forward reachability. They are based on approximating the hypervolume of the polyhedra that represents the valuations of clocks in timed automata model checking. Although the hypervolume is approximated, both techniques give exact answers.

The first is based on avoiding inclusion checks when the approximate hypervolumes makes the inclusion impossible. The other, in sorting the already-visited-states set according to the approximate hypervolumes, avoiding to traverse some

parts of it while checking for included zones. The second is only possible thanks to a very optimized implementation of a van Emde Boas tree [16], but the first is quite simple.

These techniques can be used independently and obtain interesting speedups. For instance, in cases like *MinePump* false in Table 3 it can be seen that the 16% saving of time translates to almost one hour of almost six, at the cost of 7% more memory, which only amounts to less than 12 extra MB of RAM. According to our experiments, some models benefit more with one of them, and some with the others, with acceleration of up to 19 and 15% respectively. The first one has neglectable memory overhead, the second, a moderate one (10-12%).

They can also be combined, although the speedups are not additive, because there is some mutual cancellation. The good news is that there is no need to speculate on which one to use, because using both is generally as good as using the best of them.

It should be noted that the techniques are very unobtrusive, in the sense that they are orthogonal to many other optimizations such as [3–6, 8, 9], allowing to use all of them together.

Also, we showed that applying the same ideas to the *Pending* queue –where the states that remain unexplored are kept– can have negative impact. In order to reverse that, *Pending* should be separated by location, and then sorted, but that will increase the cost of adding newly discovered zones to it. Experimentation with different data structures towards that end is a yet-to-be-explored area.

Although we chose a van Emde Boas tree to sort the *Visited* queue, some less modular yet simpler implementations –based on linked lists of states, with pointers marking insertion places– are possible. This trade-off should be revisited to see if some of the overhead can be avoided.

To pursue further in this line of research, it would be interesting to analyze which topological characteristics of the model influence in each method's performance. A consequence of this could be an on-the-fly detection method, that switches between them.

Acknowledgements We would like to thanks the insightful comments of the anonymous referees that helped us improve the paper.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2) (1994) 183–235
2. Bouajjani, A., Tripakis, S., Yovine, S.: On-the-Fly Symbolic Model-Checking for Real-Time Systems. In: 18^{th} IEEE Real-Time Systems Symposium (RTSS '97), San Francisco, USA, IEEE Computer Scienty Press (1997)
3. Behrmann, G., Bouyer, P., Larsen, K., Pelnek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Proceedings of the 10th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04). Volume 2988 of LNCS., Springer Verlag (2004) 312–326

4. Behrmann, G., David, A., Larsen, K.G., Yi, W.: Unification & sharing in timed automata verification. In: SPIN Workshop 03. Volume 2648 of LNCS. (2003) 225–229
5. Bengtsson, J.: Reducing memory usage in symbolic state-space exploration for timed systems. Technical Report 2001-009, Department of Information Technology, Uppsala University (2001)
6. Daws, C., Yovine, S.: Reducing the number of clock variables of timed automata. Proceedings IEEE Real-Time Systems Symposium (RTSS '96) (1996) 73–81
7. Wang, F.: Efficient verification of timed automata with BDD-like data-structures. In: VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation, London, UK, Springer-Verlag (2003) 189–205
8. Braberman, V., Garbervetsky, D., Olivero, A.: ObsSlice: A timed automata slicer based on observers. In: Proc. of the 16th Intl. Conf. CAV '04. LNCS, Springer Verlag (2004)
9. Braberman, V., Olivero, A., Schapachnik, F.: Optimizing timed automata model checking via clock reordering. In: 27^{th} IEEE International Real-Time Systems Symposium, Work in Progress Session, IEEE (2006)
10. Balarin, F.: Approximate reachability analysis of timed automata. In: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), Washington, DC, USA, IEEE Computer Society (1996) 52
11. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems. In: Proc. of the 10th Intl. Conf. CAV '98. Volume 1427 of LNCS., Springer-Verlag (1998) 546–550
12. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL - a tool suite for automatic verification of real-time systems. In: Hybrid Systems, Springer-Verlag (1995) 232–243
13. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: International Workshop of Automatic Verification Methods for Finite State Systems. Volume 407 of LNCS., Grenoble, France, Springer-Verlag (1990) 197–212
14. Yovine, S.: Model checking timed automata. In Rozenberg, G., Vaandrager, F., eds.: Embedded Systems. Volume 1494 of LNCS., Springer-Verlag (1997)
15. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Compact data structures and state-space reduction for model-checking real-time systems. Real-Time Syst. **25**(2-3) (2003) 255–275
16. van Emde Boas, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Mathematical Systems Theory **10** (1977) 99–127
17. Braberman, V., Olivero, A., Schapachnik, F.: Issues in Distributed Model-Checking of Timed Automata: building ZEUS. International Journal of Software Tools for Technology Transfer **7** (2005) 4–18
18. Alur, R., Courcoubetis, C., Dill, D., Halbwachs, N., Wong-Toi, H.: An implementation of three algorithms for timing verification based on automata emptiness. In: Proceedings of the 13th IEEE Real-time Systems Symposium, Phoenix, Arizona (1992) 157–166
19. Braberman, V.: Modeling and Checking Real-Time Systems Designs. Ph d. thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires (2000)