Branching Data Structures for Real-Time Model Checking Not As Good As Thought

Gervasio Pérez, Esteban Pavese, and Fernando Schapachnik

Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina [⋆]
{gdperez,epavese,fschapachnik}@dc.uba.ar

Abstract. Clock Difference Diagrams (CDDs), BDD-like data structures for model checking of timed automata, were presented as alternatives for classic DBM representation. However, work on them seems to have stopped, although there are still important open questions. CDD definition required that repeated subtrees were aliased, but no clear algorithm was presented for producing such compact representation, which seems costly to achieve. In this article we describe our implementation of such aliased subtrees and revisit CDDs by comparing their performance against DBMs on current case studies, with and without repeated subtrees. Our experiments show that CDDs still require more time and memory than DBMs, suggesting that eliminating repetitions is still not enough. Thus, this article re-opens issues that previous work on the topic considered closed.

1 Introduction and Previous Work

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 automated 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 is useful.

We focus on exploration of timed automata [1] –an extension of finite automata that models dense time– by on-the-fly forward reachability analysis [2]. Well-known tools in this area, such as Kronos [3] and UPPAAL [4], represent sets of clock valuations by means of *Difference Bound Matrices* (DBMs for short) [5]. DBMs are $(n+1)\times(n+1)$ matrices, where n is the number of clocks in the system, counting 0 as a special clock. Each cell represents the upper bound on the difference between the corresponding clocks. This data structure provides support for a graph reachability-like algorithm (depicted in Fig. 1) which saves the encountered symbolic states in a set called *Visited*.

For both warranting termination and avoiding duplicated exploration, timed automata reachability requires knowing if a newly discovered symbolic state is

[⋆] Partially supported by UBACyT 2008 X634, UBACyT X021, ANPCyT PICT 2007 532, ANPCyT PICT 2005 32440, UBACyT 20020100200103.

```
function ForwardReach(Property $\phi$)
  $Visited \leftarrow \emptyset$
  $Pending \leftarrow \{(\mathsf{l}_0, z_0)\}$
  while $Pending \neq \emptyset$
    $(\mathsf{l}, z) \leftarrow \text{next}(Pending)$
    if $(\mathsf{l}, z) \models \phi$ return YES
    end if
    $\text{Add}((\mathsf{l}_0, z_0), Visited)$
    for $(\mathsf{l}', z') \in suc_{\triangleright}(\mathsf{l}, z)$
      $Z_{\mathsf{l}'} \leftarrow \bigcup_{(\mathsf{l}', z') \in (Visited \cup Pending)} z'$
      if $z' \not\subseteq Z_{\mathsf{l}'}$
        $\text{Add}((\mathsf{l}', z'), Pending)$
        $\text{Add}((\mathsf{l}', z'), Visited)$
      end if
    end for
  end while
  return NO
end function
```

Fig. 1. Forward reachability algorithm.

already covered by the existing ones. This is inefficient when working with DBMs, because they can only represent convex clock valuations. The traditional solution is to employ sets of DBMs, but it is usually very hard to detect superpositions there, leading to repetitions of calculations and other drawbacks.

There have been many attempts to overcome this problem by using BDD-like data structures [6], which are often used in untimed systems. See, for example, Asarin et al. [7] and Strehl et al. [8] among others. Also worth noting is the work by Seshia and Bryant [9], which attempts to solve the problem solely with BDDs. However, the exemplars used in this work are too small to be conclusive. The one with the most potential, however, was introduced by Behrmann et al. in [10], where they presented *Clock Difference Diagrams* (CDDs for short). Their work used DBMs for most of the operations and CDDs for the *Visited* set, obtaining memory savings at the cost of extra time.

We have several reasons to revisit CDDs. First, besides their use as a representation for the *Visited* set, a complete reachability algorithm that did not rely on DBMs was yet to be provided. Also, being a tree-like data structure, they could have repeated subtrees. CDDs require that repeated subtrees are replaced by maximal aliasing (referred to as "maximal sharing" in the original article). Detecting repeated subtrees can be expensive, so we believe the topic requires further analysis (we elaborate on this on Section 4). Finally, original case studies used systems with a maximum of 5 clocks. The verification algorithm's complexity, being $O(n!2^nC^n)$, where n is number of clocks and C is the largest constant appearing in the inequalities [11], is dominated by the number of clocks.

In this work, we evaluate memory and time performance of three implementations for representing non-convex state sets: sets of DBMs (which is the standard representation), (non-convex) CDDs without aliasing (i.e., with repeated subtrees) and (non-convex) CDDs with aliasing (i.e., without repeated subtrees).

Through some well known case studies we show that although CDDs do reduce the number of states found, operating on them imposes an important overhead in terms of both memory and time. The culprit is to be found in the size of non-convex CDDs, and the complexity of operating on their unbounded branching structure, which distinctively separates CDDs from BDDs.

Interestingly, using aliasing to suppress repeated subtrees is not enough. Although our strategy to detect repeated substructures is lightweight, and effectively reduces the memory footprint, time and memory measurements are still considerably above the traditional strategy of using sets of non-convexes. Section 4 deepens on the subject, after presenting the necessary background on timed automata in Section 2 and CDDs in Section 3.

Relevant experimentation is presented and analyzed in Sections 5 and 6, and Section 7 rounds up the article with our conclusions and future road map.

For completeness, CRDs, short for *Clock Restriction Diagrams*, should also be mentioned. They were introduced by Wang in [12] and are similar to CDDs, but represent explicitly only one bound. Although in this work we focus on CDDs, Section 5 does perform a comparison against CRDs.

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 [3], ZEUS [13] or UPPAAL [4]. 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, 3] 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, $I : L \overset{tot}{\to} \Psi_X$ 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 $\mathcal{A} \parallel \mathcal{B}$ of TAs $\mathcal{A}$ and $\mathcal{B}$ is defined using a label-synchronized product of automata [1, 3]. 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 latter case, the system remains in the same location and all clocks increase according to the elapsed time, while still satisfying the location invariant.

It is important to note that, in the timed framework, the existence of real-valued clocks generates an infinite state space (locations plus clocks valuations). Fortunately, this does not imply undecidability of many interesting problems such as state reachability.

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 $\{v : X \overset{tot}{\to} \mathbb{R}^+\}$. 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 conjunctions represents a convex (and infinite) set of points, and is referred to as a *zone*. A data structure called Difference Bound Matrix (DBM) [5] is typically used to manipulate such kind of information. Non-convex sets are represented as unions of convex sets.

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 [14]. 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. When we mention DBMs, we are always referring to this compact version.

Symbolic states are represented by a pair (l, z) where l is a location and z a timed zone. Given a state[1], the timed successor set is computed by the $suc_{\triangleright}$ operator, defined as $suc_{\triangleright}(\mathsf{l}, z) = \{(\mathsf{l}', z') / \langle \mathsf{l}, a, \psi, \alpha, \mathsf{l}' \rangle \in E \wedge z' = suc_{\tau}(reset_{\alpha}(z \cap \psi)) \cap I(\mathsf{l}')\}$, 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. The details can be consulted in [15, Chapt. 2] or [16].

The basic (conceptual) procedure for checking reachability of property ϕ requires a queue of symbolic states to be explored, commonly known as *Pending*, and a set of already visited states, known as *Visited*. The algorithm works like this: insert the initial state in the *Pending* queue and initialize an empty *Visited* set. Then, while *Pending* is not empty, take a state from it and check whether the property ϕ holds for it. If it does, finish with a “YES” result, otherwise, put it in *Visited*, and compute its (timed) successors (one for each outgoing transition from the location). The total number of states is finite [11], but there can be repetitions. To ensure termination, before putting them in *Pending*, it should be checked that they are not *included* in any other state from *Visited*.

[1] From now on states are implicitly symbolic.

3 Clock Difference Diagrams (CDDs)

In this section, we describe our implementation of Clock Difference Diagrams as presented in [10].

3.1 Data Structure Definition

Definition 3 (CDD). *Given a set X of clock variables, $X' = X \cup \{0\}$, a CDD is defined by a tuple $k = [\mathit{diff}(k), \mathit{Ints}(k), S(k)]$, where:*

1. *$\mathit{diff}(k) \in (X \times X') \cup \{\mathit{TRUE}, \mathit{FALSE}\}$ is the clock difference represented by this node. These clock differences are extended with values TRUE and FALSE. Nodes that hold any of these latter values are called* terminal *nodes.*
2. *$\mathit{Ints}(k)$ is a list of outgoing edges. Each edge is labeled with an integral (open or closed) interval. $\mathit{Ints}(k)_n$ denotes the n^{th} list's interval, and $|\mathit{Ints}(k)|$ denotes the interval list's size.*
3. *$S(k)$ is a list of successor nodes. $S(k)_n$ and $|S(k)|$ are defined in a similar way as previously. For each $1 \leq n \leq |S(k)|$, $S(k)_n$ yields the node reached by traversing the edge $\mathit{Ints}(k)_n$.*

A CDD k such that $\mathit{diff}(k)$ is *TRUE* (or *FALSE*) will usually be referred as just *TRUE* (or *FALSE*) for short. Similarly, given an interval I we will write $S(k, I)$ to denote the node (hence, the CDD) obtained by traversing the edge labeled I from k. Formally, $S(k, I) = k' \Leftrightarrow \exists i, 1 \leq i \leq |\mathit{Ints}(k)|$ such that $\mathit{Ints}(k)_i = I \wedge S(k)_i = k'$.

Since CDDs are hierarchical in nature, an order on clock differences is needed, which can be defined as an extension of an order on clock variables.

Definition 4 (Structure invariant). *The following constraints are imposed on every node k of a CDD:*

- *Whenever $\mathit{diff}(k)$ is TRUE or FALSE, the lists $\mathit{Ints}(k)$ and $S(k)$ must be empty.*
- *In the other case, no element of $S(k)$ may be FALSE.*
- *Given a total order on clock differences $<$, it must hold that for any k' in $S(k)$, either $\mathit{diff}(k) < \mathit{diff}(k')$ or k' is TRUE.*
- *Since edges are labeled by intervals, the existence of both differences $x_i - x_j$ and $x_j - x_i$ for any pair of clocks x_i, x_j is redundant. Therefore, if $\mathit{diff}(k)$ is of the form $x_j - x_i$ it must hold that $x_i < x_j$.*
- *Every pair of intervals in $\mathit{diff}(k)$ must be disjoint. Moreover, $\mathit{diff}(k)$ must be sorted; we say that for any two intervals I and J, $I < J \Leftrightarrow \forall i, j$ such that $i \in I$ and $j \in J$, it holds that $i < j$.*
- *For any two nodes k_1 and k_2 of a CDD (potentially the same node), and for any intervals $i_1 \in \mathit{Ints}(k_1)$, $i_2 \in \mathit{Ints}(k_2)$, if it is the case that $S(k_1, i_1)$ is structurally equal to $S(k_2, i_2)$, then it must be that $S(k_1, i_1)$ and $S(k_2, i_2)$ are exactly the same structure instance. That is, we require complete structure aliasing between identical substructures.*

The previous definitions provide the cornerstone for timed automata verification using these hierarchical structures. The following definitions elaborate on the structure's semantics. For this purpose, a mapping between temporal constraints in Ψ_X and CDDs needs to be defined.

Definition 5 (Mapping of temporal constraints to CDDs, $r \rightsquigarrow CDD$). *$\forall \psi_1, \psi_2 \in \Psi_X, x, y \in X, x < y, c \in \mathbb{N}$, we define the mapping $r \rightsquigarrow CDD$ as follows:*

$$\begin{aligned}
r \rightsquigarrow CDD(True) &= [TRUE, \lambda, \lambda] && (1)\\
r \rightsquigarrow CDD(False) &= [FALSE, \lambda, \lambda] && (2)\\
r \rightsquigarrow CDD(c < x) &= [x - 0, \ll (c, \infty) \gg, \ll r \rightsquigarrow CDD(True) \gg] && (3)\\
r \rightsquigarrow CDD(x < c) &= [x - 0, \ll [0, c) \gg, \ll r \rightsquigarrow CDD(True) \gg] && (4)\\
r \rightsquigarrow CDD(x - y < c) &= [x - y, \ll (-\infty, c) \gg, \ll r \rightsquigarrow CDD(True) \gg] && (5)\\
r \rightsquigarrow CDD(\psi_1 \wedge \psi_2) &= Intersection(r \rightsquigarrow CDD(\psi_1), r \rightsquigarrow CDD(\psi_2)) && (6)
\end{aligned}$$

Mapping definitions 3, 4 and 5 have also a corresponding one for $\leq$ with the interval closed to the right.

Theorem 1. *The mapping $r \rightsquigarrow CDD$ previously defined is correct with respect to satisfiability, that is, $\forall \psi \in \Psi_X, v \in \mathcal{V}_X, v \models \psi \Leftrightarrow v \models r \rightsquigarrow CDD(\psi)$. [17]*

3.2 Algorithms

As an integral part of this work, we developed the necessary algorithms to perform full forward reachability based model-checking over CDD structures. The interested reader is referenced to [18] for further discussion of these topics.

In Fig. 2 we present the union algorithm. Remember from Fig. 1 that when a new zone z' is found to be new it should be added to the *Visited* set by means of Add$((\mathsf{l}', z')$, *Visited*). When $Visited_{\mathsf{l}}$[2] is represented as a non-convex CDD, the previous Add() is, conceptually, $Visited_{\mathsf{l}}$ = Union($Visited_{\mathsf{l}}, z'$). For clarity and ease of reading we present a recursive version of the algorithm, although the actual implementation is iterative, stack-based, modifies its parameters instead of returning a new CDD, and is tailored to consider that the second parameter does not contain branching.

4 Compressing Repeated Subtrees

Although the main motivation for CDDs was to obtain an appropriate data structure for non-convex sets, some issues require special attention. We find the requirement for maximum sharing problematic.

Indeed, the original approach to maximum sharing described in [10] was to keep a CDD-node cache as a hash table and try to find an existing node there when a new one was required. The operation was reported as taking constant

[2] $Visited_{\mathsf{l}}$ is the part of *Visited* that corresponds to location l.

```
function Union ($r_A, r_B$ : CDD)$\rightarrow r'$ : CDD
  if $r_A = FALSE$ return $CDD_B$
  else if $r_A = TRUE$ return $TRUE$
  else if $diff(r_A) < diff(r_B)$
      for all $I \in Ints(r_A)$
          $Add(r', I, \text{Union}(S(r_A, I), r_B))$
      end for
  else if $diff(r_A) > diff(r_B)$
      ... follows as previous case.
  else
      for all $I \in Ints(r_A), J \in Ints(r_B)$
          if $\neg Empty(I \cap J)$
              $Add(r', I \cap J, \text{Union}(S(r_A, I), S(r_B, J)))$
          end if
      end for
  end if
  return $r'$
end function
```

Fig. 2. CDD union recursive algorithm.

time. However, for a node to be a suitable replacement of another, their successors have to match. Unlike BDDs, which have fixed binary branching, each CDD node can be branched in any number of (non-overlapping) intervals. The depth is indeed bounded by the square of the number of clocks, as each node involves two of them[3]. It can clearly be seen that it doesn't seem likely to implement the cache in constant time, so we take the term constant as meaning *negligible*, and attribute that to the small sizes of the case studies originally used.

Case studies have grown since the CDD article by Behrmann et al. Current literature examples have at least an order of magnitude more states, and many times the number of clocks (keep in mind that timed automata reachability is exponential in the number of clocks).

To understand the impact, either as a gain or a loss, of compressing CDDs by the use of aliasing, we faced a number of options.

Whatever aliases handling strategy is chosen, it must be implemented in the Union() operation, which takes a zone found to be new and adds it to *Visited* set, as seen in Fig. 2. Remember from Section 2 that the explorations finds states (l, z), where z is a *zone*, that is, a convex CDD. Convex CDDs do not contain branching. Then, (l, z) must be checked for inclusion in $Visited_{\mathsf{l}}$, which contains a non-convex CDD (i.e., a branching CDD). If it is indeed new, then Union($Visited_{\mathsf{l}}, z$) must be performed. This is the only operation that has to modify branched CDDs.

[3] Note that this strategy differs from the *reduce* operation in OBDDs [19]. *Reduce* is called once at the end of operations to recover the structure invariant that states the nonexistence of repeated subgraphs. It works bottom up in linear time to merge isomorphic substructures. Again, the time is linear because the branching is bounded.

4.1 Handling Aliasing in the Union operation

The repetition detection could be done either on-the-fly or off-line. Doing it on-the-fly is hard, because a generated subtree might be modified later on, as the operation returns from the recursion[4], so aliases detected might need to be broken and whole subtrees copied.

The offline approach, which is easier to implement, mimics what is done in BDDs: complete the operation, then make a second pass detecting aliasing and thus compressing the CDD.

We started by the second one, and the results obtained showed no point in also trying the first approach: both approaches will obtain the same level of compaction and, as will be seen in Section 5, memory overhead is still of a couple of orders of magnitude compared to the standard version.

To achieve compression we attached a hash value to each CDD node, computed as a hashing function over its fields combined with the hash values of its descendants. A 64-bit multiplicative hash function was used. A *refcount* and a boolean *already compressed* field were also needed. As aliasing was only going to be detected at the end of the Union() operation, changes to implement compression were somehow localized:

- The Union() function maintains a CDD hash table.
- The compression is an in-order traversal looking each hash code in the table and replacing the appropriate branch by another reference to the existing one found in the table.
- CDD destruction needs to consider aliasing, via the *refcount* field.
- Aliasing is immaterial to the rest of the operations: the inclusion checking can handle it transparently, and the others only operate on zones, which are not aliased because they contain no branching. Conversely, non-convex CDDs are only modified by Union() and the destructor.

The next section shows the experimentation performed. As can be seen, in only little cases was the use of CDDs faster than DBMs. Also, the memory usage increases up to many orders of magnitude. In Section 6 we discuss possible threats to the validity of the method employed.

5 Experimentation

To test the data structures, we incorporated CDDs into the model checker Zeus[5] [13] and ran a series of experiments against well known case studies from the literature.

1. *RCS4* and *RCS5*, the *Railroad Crossing System* inspired by [20] with 4 and 5 trains. The models have 8 and 9 automata, with one clock each.

[4] Our implementation is actually iterative, but the principle is the same.

[5] Although Zeus is a distributed tool, a monoprocessor version was used for this article.

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. 14 components, one clock each.
3. *FDDI4* and *FDDI8*, an extension of the FDDI token ring protocol where the observer monitors the time the token takes to return to a given station. The models use 9 automata with 14 clocks and 17 automata with 26 clocks, respectively.
4. *Conveyor6AB*, *Conveyor Belt* [13] (with 6 stages and 2 objects). 11 components totalizing 10 clocks.
5. *Struct*, Active Structural Control System that limits structural vibration due to earthquakes or strong winds ([21]). 7 automata, one clock each.

Some case studies were also treated by ObsSlice [22], a safe model reducer, and are primed on the tables. Runs are tagged with true or false depending on whether the error state was reachable or not. All the experiments were run on an Intel Xeon 1.6 GHz machine with 4 GB of RAM, running FreeBSD 7.0 Unix in 64-bit mode. Besides the main runs, some extra ones were done to substantiate the claims in Section 6, which we omit for space reasons.

Example	DBM	CDD	Difference
	States found		(Ratio)
FDDI8 true	9272	9272	0%
FDDI8 false	9272	9272	0%
FDDI9 true	27186	27186	0%
FDDI9 false	27186	27186	0%
RCS4 true	3337	2963	-11.2 %
RCS4 false	8274	OOM	
RCS5 true	90560	OOM	
RCS5 false	281547	OOM	
Pipe6' true	52668	OOM	
Pipe6' false	24581	OOM	
Pipe6 true	727694	OOM	
Pipe6 false	80280	OOM	
Struct' true	677	638	-5.7%
Struct' false	6205	5495	-11.4%
Conveyor6AB true	2431	2431	0%
Conveyor6AB false	15481	OOM	

Table 1. States found with and without CDDs.

Table 1 reports the saving on the number of states found when using CDDs. These savings are explained by zones that would be re-explored when *Visited* is a set of zones, because although their state space is already explored, it is fragmentally covered by many other zones and thus not detected. Unfortunately, reduction in number of states does not have a correlation in neither time nor memory.

Example	DBM	CDD	CDD+A	DBM	CDD	CDD+A
	time (elapsed secs)			mem. (KB)		
FDDI8 true	520.23	114.11	121.03	24180	107432	81248
		-78.0%	-76.7%		+344%	+235%
FDDI8 false	520.13	113.81	120.83	24240	107464	82084
		-78.1%	-76.7%		+343%	+ 2.38%
FDDI9 true	2516.22	632.31	680.72	110328	1245684	780380
		-74.8%	-72.9%		+1029%	+607%
FDDI9 false	2515.50	634.45	682.86	110388	1245716	783652
		-74.7%	-72.8%		+1028%	+610%
RCS4 true	1.93	2378.00	647.97	2204	1572572	221084
		+123112%	+33573%		+71250%	+9931%
RCS4 false	7.63	6513.07+	6183.44+	3376	OOM	OOM
		∞	∞			
RCS5 true	47.29	22931.11	14430.81	8428	OOM	OOM
		∞	∞			
RCS5 false	1597.10	22979.94	14220.28	41336	OOM	OOM
		∞	∞			
Conveyor6AB true	2.04	102.93	85.23	3808	398724	123128
		+5044%	+4077%		+10370%	+3233%
Conveyor6AB false	15.56	2189.43	5553.37	11068	OOM	OOM
		∞	∞			
Pipe6' true	118.69	4101.81	6525.58	21400	OOM	OOM
		∞	∞			
Pipe6' false	52.99	3522.60	8397.18	17728	OOM	OOM
		∞	∞			
Pipe6 true	3737.10	2780.36	4663.14	191260	OOM	OOM
		∞	∞			
Pipe6 false	228.64	2978.07	5646.33	48136	OOM	OOM
		∞	∞			
Struct' true	15.56	5.90	5.19	10884	31216	10880
		-62.0%	66.5%		186.8%	-0.0%
Struct' false	3.46	4435.72+	15618.64	2564	OOM	3557300
		∞	+45130%		∞	+138740%

Table 2. Results obtained with all implementations.

Table 2 shows time and memory results for the three implementations considered. *DBM* is the standard version that uses sets of zones for the *Visited* set and expresses DBMs as a packed bit structure of the minimum constraint systems ([15, Chapt. 5]). The version that uses (non-convex) CDDs is labeled *CDD*, and *CDD+A* when aliases detection and compression was used. When the experiment ran out of memory (OOM) time was measured up to memory exhaustion. All percentages are relative to the DBM version.

As can be seen in the tables, most of the cases run out of memory with CDDs, and few of them are saved when compressing by aliasing. Compression, however, does a good job, diminishing significantly both memory and time, but the results are still orders of magnitude higher than using sets of zones.

It is clear from the results that the classical use of sets of zones as a repository outperforms the CDD implementations when it comes to memory use, even when performing full aliasing detection. This at first may seem unintuitive, since the idea behind tree-like structures is the sharing of common substructure and also leveraging on branch prefix sharing. However, some factors influence the CDD repository structure into growing bigger than expected. In the first place, the set based implementation relies strongly on the reduced representation of zones, that is, not all clock differences need to be explicitly represented in order to accurately render a given zone. In the case of tree-like structures, although each added zone is indeed represented in a reduced way, the final product may not preserve this reduction, as many different zones may have various clock differences explicitly represented. However, once they are joined, the structure is prone to converge to represent all clock differences.

Another factor that influences structure growth is *interval atomization*, that is, the phenomenon of constantly breaking up intervals into smaller components. This results from the addition of zones that overlap in a non-constant way over the already existing repository. For instance, refer to Fig. 3, where the spike-shaped area depicts the projection over clocks x and y of the temporal state space at a given point during the verification, while the rectangular area represents the projection of a new zone to be added to the repository. In a CDD representation, the original spike-shaped repository would span three different intervals for the $(y - y_0)$ clock difference. However, when adding the rectangle-shaped zone, these three intervals are further divided into seven. Whereas in the classic set-based implementation the zone addition is achieved just by adding the zone representation to the set, in the tree-like representation the zone may be copied seven times (one for each interval), up to a depth dependent on the actual size of the clock set. Although at first these copies would surely be aliased, further additions may dislodge these aliases from one another, effectively increasing structure size. It is worth mentioning that the depth to which the zones are replicated is sensitive to clock order in the CDD. We speculate atomization-generating clock differences should be pushed further down in the hierarchy as a possible deterrent to structure size explosion.

Indeed, it is well known in BDDs that the ordering of variables can have a important impact in the size of the structure. The same can happen with

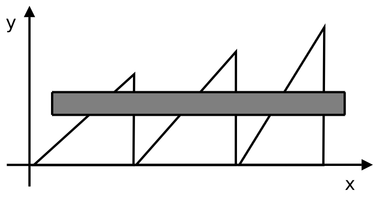

Fig. 3. An addition of a zone to a state space repository generating multiple intervals (projection over two clocks).

CDDs. However, finding a good ordering on clocks is a topic on its own, as in BDDs. We only experimented with a fixed ordering, probably suboptimal. We also performed test runs based on random ordering of the specification clocks. In these preliminary experiments, clock ordering did show a dramatic effect on structure size (drastically reducing or increasing it), although still not being competitive enough compared to the DBM results. There is still much research to perform to be able to tell how a particular clock ordering influences the structure, or whether this effect can be predicted by analyzing the raw system model. The clock ordering issue for CDDs was already mentioned in [23], although the article draws conclusions on the behaviour of CDDs based on a tool that uses CRDs, which encode only one bound. We leave to future work researching the importance of different clock orderings and finding out if there is some generic way of determining an optimal (or quasi-optimal) ordering.

The results we obtained contradict [10] and previous work on CRDs [12]. We were not able to test directly against UPPAAL, as current versions do not include CDDs as a choice of data structure. RED, the tool based on CRDs, does not perform well on our case studies. For example, on *RCS4* true, it takes 5 seconds and 10 MB of memory (more than our DBM implementation but less than CDDs), but also reports a (spurious) counterexample for the unreachable version. The other case studies behaved similarly. Such discrepancies could not be solved while corresponding with the author, within the timeframe available for this article.

6 Threats to Validity

The technique presented is not claimed to be definitive and is under ongoing research. Nevertheless it is important to understand that deciding aliasing based only on hash values can lead to regarding different subtrees as the same. In turn, as non-convex CDDs are used to check if a newly found zone is indeed yet unexplored, it can be the case that it is explored again without need, because a branch was incorrectly chopped off from the CDD. On a pathological bad case this could jeopardize termination. Being aware of the potential problem, we still consider our results valid because:

- In terms of time used, the technique provides a lower bound, compared to what should be used in a "production" setting, which will require comparing complete subtrees or any other more involved method.
- Calculating hashes and storing them incurs by itself in an overhead, which we measured to be around 40% in time and around 10% in memory. That means that resorting to the full comparison method would consume at most 10% less memory, because there might be no need to store the hashes.
- However, if hashes are not used, and get replaced by an in-depth comparison of subtrees, time requirements scale dramatically: all our case studies would run out of time.
- If the model checker does terminate, the number of explored states could be higher than the version without compression, but equally correct.
- Although the model checker ran out of memory in some cases, it did so by the very size of the case study. The states-found counter showed normal values in all cases.
- When it did terminate, the number of states found, and when appropriate, the trace, coincided with the version without compression.
- There is an easy way around to get exact results: use hashes to determine if two subtrees are *possible* matches, and if the hashes coincide, only then perform an in-depth comparison. Note that this approach does not save memory, and the runtime overhead can be between 7% and 20% on the cases where we tried it.
- Nevertheless, we still did all of our experimentation only using hashes, because resorting to in-depth comparisons would only increase time and possibly memory counters, not changing the overall results.
- The data structure used to represent branching CDDs does have provisions to sacrifice a little extra space in order to save time (while adding an extra branching, for instance). As these options are tunable, we turned them off to rule out that the memory overhead of CDDs was not due to them. Results showed that the memory footprint diminished between 10 and 40% with a proportional increase in time. The extra memory requirements for CDDs being many orders of magnitude higher, the potential saving does not make the difference.
- Finally, the sheer difference in our obtained results versus previous work may suggest a bug in our programming. In order to minimize this issue we have performed extensive testing on our tool, and also monitored memory usage through the `valgrind` [24] tool, and detected no bugs that would result in spurious memory usage.

Clock ordering deserves separate mention, as an appropriate one could have high impact in the results. Our preliminary results are non yet conclusive in this regards, as mentioned in page 5. We leave this area pending to future research.

7 Conclusions and Future Work

In this article we revisited Clock Difference Diagrams, a data structure representing non-convex sets of clock valuations for model checking of timed automata.

The original presentation left some questions open, namely experimentation with bigger case studies, and clarification of the repeated subtree detection algorithm.

To approach these questions, we evaluated memory and time performance of three implementations for representing non-convex state sets: sets of DBMs (which is the standard implementation), (non-convex) CDDs without and without aliasing (i.e., with and without repeated subtrees).

Regarding CDDs, as shown in Section 5, if repeated subtrees are not pruned using aliasing, time and memory requirements can grow by many orders of magnitude, even though the number of states diminishes as expected in some cases.

Our experiments confirm that although aliasing can decrease memory consumption significantly, it is not enough to make the CDD version competitive. Even with aliasing, CDDs need orders of magnitude more memory. The original article that introduced CDDs showed memory savings at the price of augmented times. Their case studies, although representative at the time, are orders of magnitude smaller than the ones that tools handle nowadays. We attribute to the number of clocks used and the hundreds of thousands of states dealt with today the difference in outcome, as both factors contribute to more branching in CDDs. A favorable ordering of clocks could also be responsible for the difference.

Said article did not provide details on how aliasing was handled. We found that detecting it is costlier and less useful than expected. Our experiments still place the non-convex, aliased CDD implementation as taking much more time than the standard one based on sets of convexes.

Of course we cannot prove that there is not a better implementation that correctly handles aliasing and is faster than the standard one. But after getting involved with the details of the algorithms we haven't found convincing, detailed evidence that it does exist. Even if it did exist, our current implementation does achieve maximum compression, and on current days case studies, the memory overhead is just too high. It is worth emphasizing that our implementation is based on the same packet bit structure described in [15, Chapt. 5], which is very efficient in space.

As already mentioned, clock ordering also deserves analysis. It is well known that BDD sizes are highly sensitive to the order of variables. The same can be expected from CDDs. The topic of how to order clocks to diminish branching in CDDs is, in our opinion, the path that could turn CDDs useful in the future, if at all. For that to happen, orderings that diminish branching by orders of magnitude need to be found, to make up for the orders of magnitude more memory that branching seems to consume.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science (1994)
2. Bouajjani, A., Tripakis, S., Yovine, S.: On-the-Fly Symbolic Model-Checking for Real-Time Systems. In: RTSS '97. (1997)
3. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems. In: CAV '98. LNCS, Springer (1998)

4. 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 (1995)
5. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Automatic Verification Methods for Finite State Systems. LNCS, Springer (1990)
6. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **35**(8) (1986) 677–691
7. E. Asarin, M. Bozga, A. Kerbrat, O. Maler, M. Pnueli, A. Rasse: Data structures for the verification of timed automata. In: Hybrid and Real-Time Systems. LNCS, Springer (1997)
8. K. Strehl, L. Thiele: Symbolic model checking using Interval Diagram techniques. Technical report, Computer Engineering and Networks Lab (TIK), ETH (1998)
9. Seshia, S., Bryant, R.: Unbounded, fully symbolic model checking of timed automata using boolean methods. In: CAV'03. LNCS. Springer (2003)
10. Behrmann, G., Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Efficient timed reachability analysis using Clock Difference Diagrams. In: CAV'99. (1999) 341–353
11. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. Information and Computation (1993)
12. Wang, F.: Efficient verification of timed automata with BDD-like data-structures. In: VMCAI 2003, London, UK, Springer-Verlag (2003) 189–205
13. 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
14. 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
15. Schapachnik, F.: Timed Automata Model Checking in Monoprocessor and Multiprocessor Architectures. Phd thesis, Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires (2007)
16. Yovine, S.: Model checking timed automata. In Rozenberg, G., Vaandrager, F., eds.: Embedded Systems. Volume 1494 of LNCS., Springer-Verlag (1997)
17. Pavese, E.: A new data structure based on BDDs for the model checking of timed systems. Degree thesis, FCEyN, UBA (2006)
18. Pavese, E., Schapachnik, F.: Relaxed Clock Difference Diagrams for Timed Automata Model Checking. Technical report, FCEyN, UBA (2007)
19. Clarke, E., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press. Cambridge, Massachusetts (1999)
20. Alur, R., Courcoubetis, C., Dill, D., Halbwachs, N., Wong-Toi, H.: An implementation of three algorithms for timing verification based on automata emptiness. In: RTS'92. (1992)
21. Elseaidy, W., Cleaveland, R., Jr., J.B.: Modeling and verifying active structural control systems. Science of Computer Programming **29**(1-2) (1997) 99–122
22. Braberman, V., Garbervetsky, D., Olivero, A.: ObsSlice: A timed automata slicer based on observers. In: CAV '04. LNCS, Springer Verlag (2004)
23. Beyer, D., Noack, A.: Can decision diagrams overcome state space explosion in real-time verification? In König, H., Heiner, M., Wolisz, A., eds.: FORTE. Volume 2767 of Lecture Notes in Computer Science., Springer (2003) 193–208
24. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. In: RV'03. (2003)