New UPPAAL Architecture

Alexandre David[1], Gerd Behrmann[2], Kim G. Larsen[2], and Wang Yi[1]

[1] Department of Information Technology, Uppsala University, Sweden
{adavid,yi}@docs.uu.se
[2] Department of Computer Science, Aalborg University, Denmark
behrmann@cs.auc.dk.

Abstract. We present the design of the new model-checking engine architecture and new internal data structures for the next generation of UPPAAL. Experimental results demonstrate that the new implementation based on these structures improves the efficiency of UPPAAL by about 80% in both time and space. In addition, the new version is built to handle hierarchical models. The challenge in handling hierarchy comes from the very dynamic structure of hierarchical systems: the level of concurrency and the scope of data variables and clocks are not constant.

1 Introduction

The interest in timed automata, first introduced by Alur and Dill[1] in 1990, as a modelling language for timed systems has risen as tools have become more sophisticated and faster. Although the basic model checking algorithms and data structures, including techniques for approximative analysis[18], state space reduction[16], compact data structures[16, 6], clock reduction [11] and other optimisations, for timed automata are well known, there has been little information on how these techniques fit together into a common efficient architecture.

This paper provides a view of the architecture of the real time model checker UPPAAL.[1] The goal of UPPAAL has always been to serve as a platform for research in timed automata techniques. As such, it is crucial for the tool to provide a flexible architecture that allows experimentation, i.e., it must be possible to integrate *orthogonal* features in an orthogonal manner such that it becomes possible to *evaluate* various techniques within a single *framework* and investigate how these influence each other.

The timed automaton reachability algorithm is basically a graph exploration algorithm where the vertices are *symbolic states* and the graph is unfolded on the fly. During exploration, the algorithm maintains two sets of symbolic states: The *waiting list* contains reachable but yet unexplored states, and the *passed list* contains reachable explored states. Maintaining two sets of states does incur some overhead that can be eliminated by unifying them. We show that this results in a significant speedup.

Hierarchical automata are extensions of regular automata where each state can contain another automaton or a number of concurrent automata. The notion

[1] Visit http://www.uppaal.com for more information.

of hierarchy in automata languages was first introduced by Harel as statecharts[14] and later in UML[2]. There has been recent effort in extending UML with real time concepts and, vice versa, extending timed automata with a notion of hierarchy [13]. Although the hierarchical concepts do not invalidate the existing model checking algorithms for timed automata, the varying degree of concurrency and number of variables make it challenging to represent the states of the system in a compact, efficient and orthogonal manner. We present how this can be done in the new architecture of UPPAAL.

Contributions We contribute a flexible and efficient architecture for timed automata model checkers. We show how this architecture makes it possible to implement various algorithms and data structures in an orthogonal manner making it possible to evaluate these techniques within a common framework. Especially, we show how hierarchical timed automata can be supported. We present results of combining the two main data structures, the waiting list and the passed list, into a single data structure. Finally, we show the effect of sharing common elements of states, thereby reducing the memory requirements by up to 80%.

Related work The state space storage approach presented in this paper is similar to the one in[12] for hierarhical coloured petri nets. Both approaches share similarities with BDDs [10] in that common substructures are shared, but avoid the overhead of the fined grained data representation of BDDs.

The zone union used our state representation is a simple list of zones. A more elaborate representation is the CDD [6] that can be used efficiently for analysis. However there are still a number of unresolved problems if we want to use a unified passed and wait structure. Furthermore it is not known how to cope with meta-data needed for individual symbolic states.

Hiearchical state-space verification is not new [2, 7]. However these studies concern theory with algorithm presentations and complexity analysis. They do not address the practical issues for hierarchical searching. In [17] a hybrid approach is used to deal with hierarchical systems but they do not handle hierarchical timed automata. In [3] an ROBDD based approach to hierarchical reachability analysis for untimed systems is presented, but ROBDD based techniques are not directly applicable to timed automata, although a number of attempts have been made.

Outline Section 2 summarises the definition of timed automata, the semantics, and the timed automaton reachability algorithm. In section 3 we present the architecture of UPPAAL and in section 4 we discuss how the passed and waiting list can be combined into a single efficient data structure. The actual representation of the states in this new data structure is discussed in section 5 and section 6 extends this to hierarchical timed automata. Finally, we present preliminary experimental results in section 7.

[2] http://www.uml.org

2 Notations

In this section we summaries the basic definition of a timed automaton, the concrete and symbolic semantics and the reachability algorithm.

Definition 1 (Timed Automaton). *Let C be the set of clocks. Let $B(C)$ be the set of conjunctions over simple conditions on the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. A timed automaton over C is a tuple (L, l_0, E, g, r, I), where L is a set of locations, $l_0 \in L$ is the initial location, $E \in L \times L$ is a set of edges, $g : E \to B(C)$ assigns guards to edges, $r : E \to 2^C$ assigns clocks to be reset to edges, and $I : L \to B(C)$ assigns invariants to locations.*

Intuitively, a timed automaton is a graph annotated with conditions and resets of non-negative real valued clocks.

Definition 2 (TA Semantics). *A clock valuation is a function $u : C \to \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. Let $\mathbb{R}^C$ be the set of all clock valuations. Let $u_0(x) = 0$ for all $x \in C$. We will abuse the notation by considering guards and invariants as sets of clock valuations.*

The semantics of a timed automaton (L, l_0, E, g, r, I) over C is defined as a transition system $(S, s_0, \to)$, where $S = L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\to \subseteq S \times S$ is the transition relation such that:

- $(l, u) \to (l, u + d)$ *if* $u \in I(l)$ *and* $u + d \in I(l)$
- $(l, u) \to (l', u')$ *if there exist* $e = (l, l') \in E$ *s.t.* $u \in g(e)$, $u' = [r(e) \mapsto 0]u$, *and* $u' \in I(l)$

where for $d \in \mathbb{R}$, $u + d$ maps each clock x in C to the value $u(x) + d$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to the value 0 and agrees with u over $C \setminus r$.

The semantics of timed automata results in an uncountable transition system. It is a well known-fact that there exists a exact finite state abstraction based on convex polyhedra in $\mathbb{R}^C$ called zones (a zone can be represented by a conjunction in $B(C)$). This abstraction leads to the following symbolic semantics.

Definition 3 (Symbolic TA Semantics). *Let $Z_0 = I(l_0) \wedge \bigwedge_{x,y \in C} x = y$ be the initial zone. The symbolic semantics of a timed automaton (L, l_0, E, g, r, I) over C is defined as a transition system $(S, s_0, \Rightarrow)$ called the* simulation graph, *where $S = L \times B(C)$ is the set of symbolic states, $s_0 = (l_0, Z_0 \wedge I(l_0))$ is the initial state, $\Rightarrow = \{(s, u) \in S \times S \mid \exists e, t : s \overset{e}{\Rightarrow} t \overset{\delta}{\Rightarrow} u\}$: is the transition relation, and:*

- $(l, Z) \overset{\delta}{\Rightarrow} (l, norm(M, (Z \wedge I(l))^{\uparrow} \wedge I(l)))$
- $(l, Z) \overset{e}{\Rightarrow} (l', r_e(g(e) \wedge Z \wedge I(l)) \wedge I(l'))$ *if* $e = (l, l') \in E$.

where $Z^{\uparrow} = \{u + d \mid u \in Z \wedge d \in \mathbb{R}_{\geq 0}\}$ *(the* future *operation), and* $r_e(Z) = \{[r(e) \mapsto 0]u \mid u \in Z\}$*. The function norm* $: \mathbf{N} \times B(C) \to B(C)$ *normalises the clock constraints with respect to the maximum constant* M *of the timed automaton.*

The relation $\overset{\delta}{\Rightarrow}$ contains the delay transitions and $\overset{e}{\Rightarrow}$ the edge transitions. The classical representation of a zone is the Difference Bound Matrix (DBM). For further details on timed automata see for instance [1, 9]. Given the symbolic semantics it is straight forward to construct the reachability algorithm, shown in Fig. 1. The symbolic semantics can be extended to cover networks of communicating timed automata (resulting in a location vector to be used instead of a location), timed automata with finite data variables (resulting in the addition of a variable vector) and to hierarchical timed automata.

```
waiting = {(l0, Z0 ∧ I(l0))}
passed = ∅
while waiting ≠ ∅ do
  (l, Z) = select state from waiting
  waiting = waiting \ {(l, Z)}
  if testProperty(l, Z) then return true
  if ∀(l, Y) ∈ passed : Z ⊈ Y then
    passed = passed ∪ {(l, Z)}
    ∀(l', Z') : (l, Z) ⇒ (l', Z') do
      if ∀(l', Y') ∈ waiting : Z' ⊈ Y' then
        waiting = waiting ∪ {(l', Z')}
      endif
    done
  endif
done
return false
```

Fig. 1. The timed automaton reachability algorithm. The function *testProperty* evaluates the state property that is being checked for satisfiability. The while loop is refered to as the exploration loop.

3 Architecture

The seemingly simple algorithm of Fig. 1 turns out to be rather complicated when implemented. To make matters worse it has been extended and optimised to reduce the runtime and memory usage of the tool. Most of these optimisations are optional since they involve a tradeoff between speed, memory usage and feature richness.

The architecture of UPPAAL has changed a lot over time. Some years ago UPPAAL was a more or less straightforward implementation of the timed au-

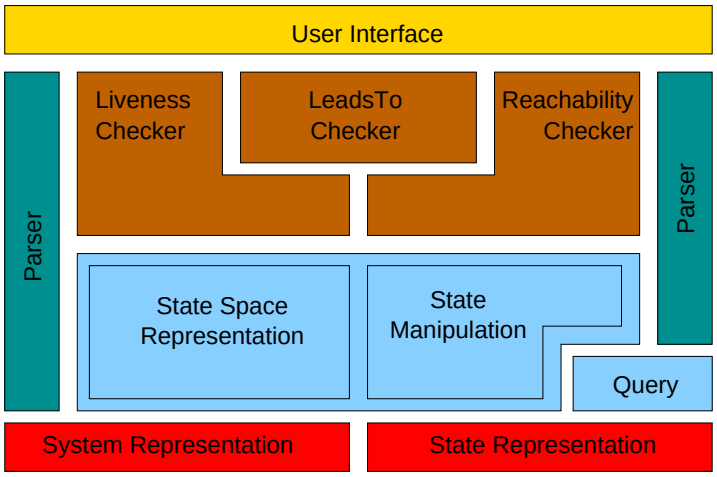

Fig. 2. UPPAAL uses a layered architecture. Components for representing the input model and a symbolic state are placed at the bottom. The state space representations are semantically a set of symbolic states and together with the state operations they form the next layer. The various checkers combine these operations to provide the complex functionality needed. This functionality is made available via either a command line interface or a graphical user interface.

tomaton reachability algorithm annotated with conditional tests on features or options. Although simple it had several disadvantages:

- The core reachability algorithm became more and more complicated as new options where added.
- There was an overhead involved in checking if an option was enabled. This might not seem as much, but when this is done inside the exploration loop the overhead adds up.
- Some experimental designs and extensions required that certain algorithms were replaced with radically different algorithms.

The architecture of UPPAAL is constantly restructured in order to facilitate new designs and algorithms without needing to fork the code base, see Fig. 2 for the latest incarnation. The main goals of the design are speed and flexibility. The bottom layer providing the system and state representations has only seen minimal architectural changes over the years. In fact, the code where most options are implemented are in the *state space manipulation* and *state space representation* components.

The basic building blocks of the three checkers are *filters* and *buffers*. Buffers have the traditional *put* and *get* methods to add and remove elements and a mechanism for enumerating all elements. Filters accept input via a *put* method and provide output by putting data into another filter or a buffer (filters and buffers share the same *sink* interface containing the put method). Several fil-

ters and buffers can be connected to form a pipeline.[3] The various state space representations provide an implementation of the buffer interface and the state manipulations provide an implementation of the filter interface.

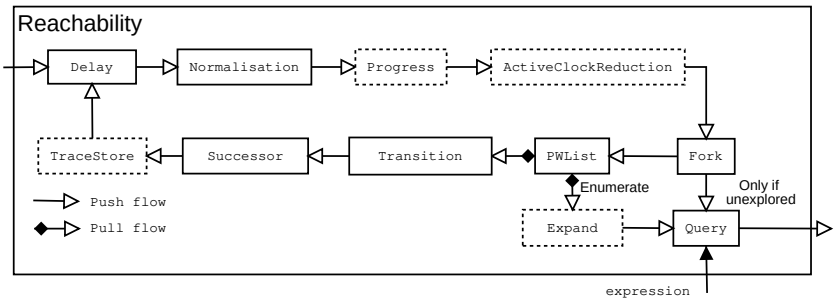

Fig. 3. The reachability checker is actually a compound object consisting of a pipeline of filters. Optional elements are dotted.

The reachability checker is actually a filter that takes the initial state as its input and generates all reachable states satisfying the property. It is implemented by composing a number of other filters into a pipeline, see Fig. 3. The pipeline realises the reachability algorithm of Fig. 1. The pipeline consists of filters computing the edge successors (`Transition` and `Successor`), the delay successors (`Delay` and `Normalisation`), and the unified passed and waiting list buffer (`PWList`). Additional components include a filter for generating progress information (e.g. throughput and number of states explored), a filter implementing active clock reduction [11], and a filter storing information needed to generate diagnostic traces. Notice that some of the components are optional. If disabled a filter can be bypassed completely and does not incur any overhead.

Semantically, the `PWList` acts as a buffer that eliminates duplicate states, i.e. if the same state is added to the buffer several times it can only be retrieved ones, even when the state was retrieved before the state is inserted a second time. To achieve this effect the `PWList` must keep a record of the states seen and thus it provides the functionality of both the passed list and the waiting list.

Definition 4 (PWList). *Formally, a* `PWList` *can be described as a pair* $(P, W) \in 2^S \times 2^S$*, where* S *is the set of symbolic states, and the two functions* $put : 2^S \times 2^S \times S \to 2^S \times 2^S$ *and* $get : 2^S \times 2^S \to 2^S \times 2^S \times S$*, such that:*

- $put(P, W, (l, Z)) = (P \cup \{(l, Z)\}, W')$ *where*

$$W' = \begin{cases} W \cup \{(l, Z)\} & \text{if } \forall (l, Y) \in P : Z \not\subseteq Y \\ W & \text{otherwise} \end{cases}$$

[3] In contrast to pipeline designs seen in audio and video processing software and hardware design there is no concurrency involved.

– $get(P, W) = (P, W \setminus \{(l, Z)\}, (l, Z))$ *for some* $(l, Z) \in W$.

Here P and W play the role of the passed list and waiting list, respectively, but as we will see this definition provides room for alternative implementations. It is possible to loosen the elimination requirement such that some states can be returned several times while still ensuring termination, thus reducing the memory requirements[16]. In this paper we will call such states *transient*. Section 4 will describe various implementations of the `PWList`.

In case multiple properties are verified, it is possible to reuse the previously generated reachable state space by reevaluating the new property on all previously retrieved states. For this purpose, the `PWList` provides a mechanism for enumerating all recorded states. One side effect of transient states is that when reusing the previously generated reachable states space not all states are actually enumerated. In this case it is necessary to explore some of the states using the `Expand` filter.[4] Still, this is more effective than starting over.

The number of unecessary copy operations during exploration has been reduced as much as possible. In fact, a symbolic state is only copied twice during exploration. The first time is when it is inserted into the `PWList`, since the `PWList` might use alternative and more compact representations than the rest of the pipeline. The original state is then used for evaluating the state property using the `Query` filter. This is destructive and the state is discarded after this step. The second is when constructing the successor. In fact, one does not retrieve a state from the `PWList` directly but rather a reference to a state. The discrete and continous parts of the state can then be copied directly from the internal representation used in the `PWList` to the memory reserved for the successor. Since handling the discrete part is much cheaper than handling the continous part, all integer guards are evaluated first. Only then a copy of the zone is made and the clock guards are evaluated.

The benefits of using a common filter and buffer interface are *flexibility*, *code reuse*, and *acceptable efficiency*. Any component can be replaced at runtime with an alternate implementation providing different tradeoffs. Stages in the pipeline can be skipped completely with no overhead. The same components can be used and combined for different purposes. For instance, the `Successor` filter is used by both the reachability checker, the liveness checker, the deadlock checker, the `Expand` filter, and the trace generator. Since the methods on buffers and filters are declared virtual they do incur a measurable call overhead (approximatively 5%). But this is outweighed by the possibility of skipping stages and similar benefits. In fact, the functionality provided by the `Successor` filter was previously provided by a function taking a symbolic state as input and generating the set of successors. This function was called from the exploration loop which then added these successors to the waiting list. The function returned the successors as an array of states.[5] The overhead of using this array was much higher than the call overhead caused by the pipeline architecture.

[4] The `Expand` filter is actually a compound filter containing an instance of the `Successor` and `Transition` filters.

[5] It was actually a `vector` from the C++ *Standard Library*.

4 Unifying the Passed list and Waiting List

In this section we present the concept of the unified passed and waiting list, and we detail a reference implementation of it.

4.1 Unification Concept

The main conceptual difference between the present and previous implementations of the algorithm is the unification of the passed list and waiting list. As described in the previous sections, these two lists are the major data structures of the reachability algorithm. The waiting list holds states that have been found to be reachable but not yet been explored whereas the passed list contains the states that have been explored. Thus a state is first inserted into the waiting list where it is kept until it is explored at which point it is moved to the passed list. The main purpose of the passed list is to ensure termination and secondly to avoid exploring the same state twice. Fig. 1 shows the reachability algorithm based on these lists.

One crucial performance optimisation is to check whether there is already a state in the waiting list that is a subset or superset of the state being added. In this case one of the two states can be discarded [8]. This was implemented by combining the queue or stack structure in the waiting list with a hash table providing a fast method to find duplicate states. Obviously, the same is done for the passed list. This approach has two drawbacks:

- States are looked up in a hash table twice.
- The waiting list might contain a large number of states that have previously been explored, but this is not realised until the state is moved to the passed list thus wasting memory.

The present implementation unifies the two hash tables into one. There is still a collection structure representing the waiting list, but it only contains simple references to entries in the hash table. Furthermore pushing a state to the waiting list is a simple append operation.

A number of options are realisable via different implementations of the `PWList` to approximate the representation of the state-space such as *bitstate hashing*[15], or choose a particular order for state-space exploration such as *breadth first*, *depth first*, *best first* or *random*[5, 4]. The ordering is orthogonal to the storage structure and can be combined with any data representation.

This unified structure implements the `PWList` interface defined in the previous section: From the pipeline point of view new states are pushed and waiting states to be explored are popped. Using this structure allows the reachability algorithm to be simplified to the one given in Fig. 4. In this algorithm the states popped from the queue do not need inclusion checking, only the successors need this.

```
Q = PW = {(l_0, Z_0 ∧ I(l_0))}
while Q ≠ ∅ do
  (l, Z) = select state from Q
  Q = Q \ {(l, Z)}
  if testProperty(l, Z) then return true
  ∀(l', Z') : (l, Z) ⇒ (l', Z') do
    if ∀(l', Y') ∈ PW : Z' ⊈ Y' then
      PW = PW ∪ {(l', Z')}
      Q.append(l', Z')
    endif
  done
done
return false
```

Fig. 4. Reachability algorithm using the unified PWList. In the reference implementation Q only contains references to the entries in PW.

4.2 Reference Implementation

Figure 5 shows the reference implementation of our unified structure. The hash table gives access to the reachable state-space. Every state has a discrete state entry and a union of zone as its symbolic part. The waiting queue is a simple collection of state references (e.g. a linked list).

The first characteristic of this reference implementation is that it builds on top of the storage interface, which allows to change the actual data representation independent of the exploration order. This order may be changed by modifying only where the state reference is added in the waiting queue.

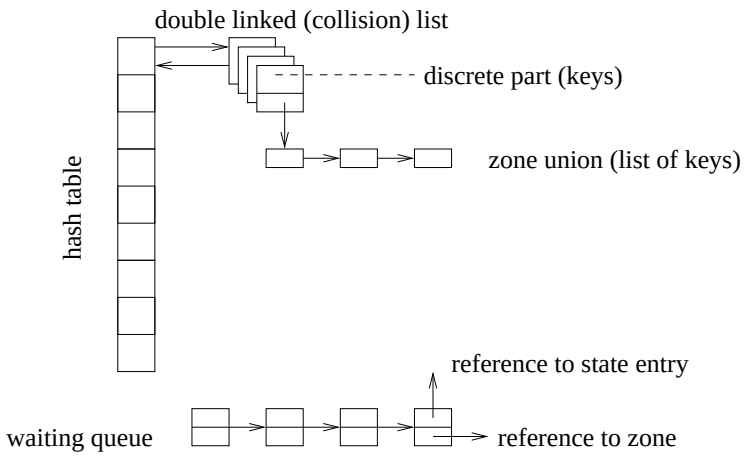

Fig. 5. Reference implementation of PWList.

The second characteristic comes from its state-space representation: the main structure is a hash table giving access to states. The states have a unique entry for a given discrete part, i.e. locations and variables. The symbolic part is a union of zones, or more precisely of zone keys handled by the storage structure. As a first implementation this union is a list of keys, but we plan for future experiments a CDD representation that is well-suited for such union of zones[6]. Besides, this representation avoids any discrete state duplicates. There is sharing of discrete data at this level. The storage underneath may implement sharing of all data between different discrete states and zones of different unions of zones: this is at a lower level and it is described in section 5.

The third characteristic is the limited use of double-linked lists. The discrete state list (collision list of the hash table) is double-linked because we need to be able to remove transient states when they are popped of the waiting list. The waiting queue is single-linked because its length is rather small and it is efficient to decide on a validity bit if a popped state should be explored or thrown away. In this case we postpone the removal of states. The same applies for the zones in the zone union. Proper removal of states involves a simple flag manipulation. It is an implementation detail, not to be discussed here.

The push operation is described as follows: hash the discrete part of the state to get access to the zone union. Check for inclusion, remove included zones, add this new zone, or refuse the zone. Finally add a reference to the waiting queue. The pop operation consists of popping a state reference and checking for its validity (a simple flag).

4.3 Experiments

We instrumented the reference implementation to isolate the impact of the unified list. We use the same experiments presented in section 7 with the addition of dacapo, a TDMA protocol. We count in the inclusion checking the number of (symbolic) states that are included in the new state and the number of new states rejected because they are already explored. Among these states that are on the passed and the waiting list, we count those that are marked "waiting", i.e. not yet explored. Table 1 shows how often an inclusion is detected with a waiting state. The figures are highly dependent on the model and which states are generated. Compared with the traditional 2-lists approach, we avoid to push states to the passed list or the waiting list. However the exploration is still the same since a waiting state that is going to be explored is guaranteed not to be in the passed or the waiting list in both approaches. In addition to this, if we consider the length of the waiting list compared with the passed list, we expect a performance improvement, but not critical. This is confirmed in the experiments of section 7.

5 Storage Structure

The storage structure is the lower layer whose role is to store simple data. It is in charge of storing location vectors, integer variables, zone data, and other meta

Model	superset	subset
Cups	97%	86%
Bus Coupler	17%	60%
Dacapo	86%	81%

Table 1. Percentage of waiting states of the inclusion detections. Waiting states may be superset (strict) or subset (non-strict) of the new state we are testing.

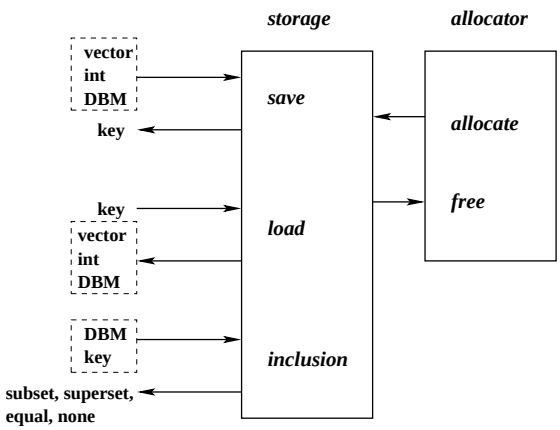

Fig. 6. The interface of the storage with the allocator underneath.

variables used for certain algorithms, i.e. guiding[4]. This structure is based on keys: data is sent to the storage that returns a key to be able to retrieve the data later. In addition to this the storage is able to perform simple operations such as equality testing of vectors and inclusion checking of zones, without the intermediate step of reading the data first with the associated key. The different storage implementations are built on top of a specialised data allocator. This allocator allocates memory by big chunks and is optimised to deliver many small memory blocks of limited different types. This means that the memory allocation has very little overhead and is very efficient for allocating and deallocating many memory blocks of the same size. This is justified by the nature of the data we are storing: there are few types of vectors and data structures stored but their number is huge. Figure 6 illustrates the main functions of the interface with the allocator underneath.

The storage structure is orthogonal to a particular choice of data representation and the pwlist structure. Particular algorithms aimed at reducing the memory footprint such as *convex hull approximation*[18] or *minimal constraint representation*[16] are possible implementations. These will be ported from the UPPAAL code base. We have implemented two variants of this storage, namely one with simple copy and the other one with data sharing.

It is important to notice that UPPAAL implements a minimal constraint representation based on graph reduction and we do not compare this reduction.

This reduction gives 20-25% gain in memory. This reduction can give more gain in addition to the shared storage implementation. This is not discussed here.

5.1 Simple Storage

The simple storage copies data in memory allocated by the allocator and is able to restore original data. This is similar to the default implementation of UPPAAL with the difference that DBM matrices are save without their diagonal. The diagonal contains the constraints $x_i - x_i \leq 0$ which do not need to be copied. This implementation is the reference implementation for comparison against UPPAAL and against other implementations of this storage structure.

5.2 Shared Storage

Before implementing such a structure we instrumented the current implementation of UPPAAL to see how much of the data was shared. We put a printout code at the stage where a state is stored after having tested it for inclusion. The printing was processed through a perl script to analyse it. Table 2 shows consistent results concerning storage of location vectors, integer variables, and DBM data. This property holds through different examples and can be explained by the way the reachability works: when computing the next state all the possibilities are tried, so for a given location many variable settings exist. The same holds in the other direction: a given variable set will exist in many location configurations. The differences in the results are consistent: *audio* and *dacapo* are middle sized models, *fischer* is the well-known Fischer's protocol for mutual exclusion which behaves badly with respect to timing constraints, and *bus coupler* is a very big example. The bigger the model, the more combinations, and the more sharing we get. The *audio* model is more oriented on control locations. The pre-experiment justified this shared storage implementation.

Table 2. Results from instrumented UPPAAL. The smaller the numbers are, the more copies there are.

Model	Unique locations	Unique variables	Unique DBMs
Audio	52.7%	25.2%	17.2%
Dacapo	4.3%	26.4%	12.7%
Fischer4	9.9%	0.6%	64.4%
Bus coupler	7.2%	8.7%	1.3%

The shared storage has a hash table internally to be able to find previously saved data quickly. This requires to compute a hash value for every saved data. However we need to compute hash values anyway to retrieve the discrete part of a state so this is done only once. Another possible overhead is the lookup in collision lists. By a careful choice of the hash function collisions are rare and

besides this matches are found in 80% of the cases because of the high sharing property of stored data.

A particular choice has been made concerning the deletion of stored data for this implementation (the interface is free on this point). Only zone data, i.e. DBMs here, are really deallocated. We justify this by the high expected sharing of the discrete part of the states, that is not going to be removed from the passed list. When testing for zone inclusion, we may have to remove zones (this is implemented), but the discrete part is equal. The only case where this does not hold is for transient states because they are stored only in the waiting list and never in the passed list. This will give a set of locations that could be freed from memory. However removing data requires double linked lists, and the locations and variables are saved the same way. For this implementation we adopted this compromise.

6 Ready for the Future: Hierarchy

Hierarchy is commonly used to handle complexity of models. Well known hierarchical statecharts are those used in STATE-MATE and Rhapsody. UPPAAL will support hierarchy natively in the future on the base of hierarchical timed automata. The main challenge in hierarchical exploration is to handle the dynamic of data, in particular when changing scope and level of parallelism. The pipeline of UPPAAL was modified with this in mind and the storage is built to deal with such dynamic data.

Concerning the discrete data, they are simple vectors that need to be handled at the beginning of the successor state computation. It is there that we get the information on which variables and clocks we need for the next state. Then we need to handle the symbolic part based on this information. We have to carry operations on the smallest possible zones because such operations are cubic and quadratic. To do so the storage structure has built-in addition of clocks in the load function and the zones have a shrink method. The future pipeline for successor computation using this is depicted in Fig. 7. This figure shows the different steps needed to compute a successor of a state with the smallest number of useless copies. From a given state with a clock x in its scope, we have to copy the discrete part of the state to evaluate the guards of a particular transition. It it is false other transitions from the same state may be tried, otherwise the next location may be computed. Then the symbolic part is copied and new clocks may be added and removed. In our case the state is changed with a new scope with a clock y: we have to add constraints for y. Further operations can be computed like delays, resets, and assignments to obtain the symbolic successor state. Invariant evaluation occurs after the resets. The main difference with the non-hierarchical version is the operation add/remove clocks. This operation will be skipped in most cases since transitions that change scope and level of parallelism are in minority. So we expect no negative impact on performance.

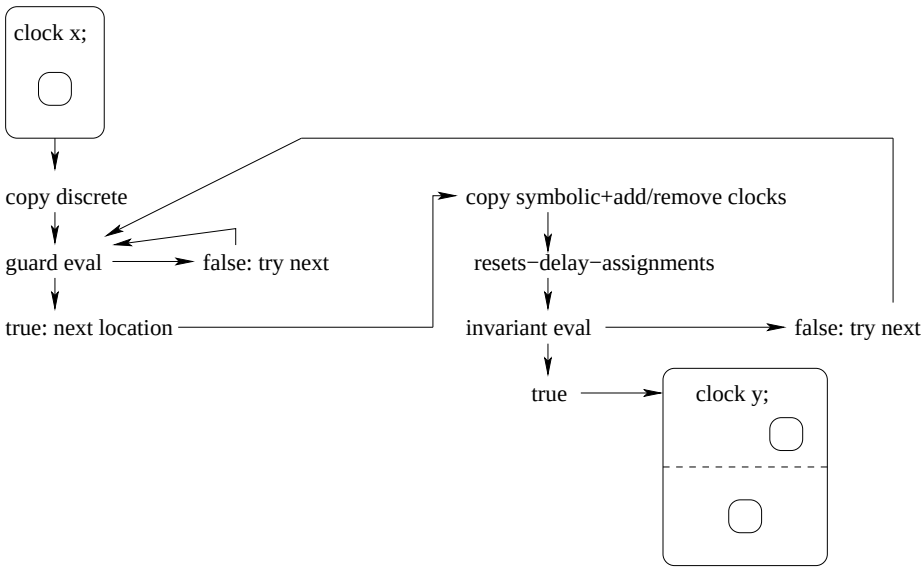

Fig. 7. Future hierarchical pipeline for successor computation.

7 Experiments

We conduct the experiments on the development version of UPPAAL. We compare results before and after the modifications presented in this paper. It is worth noticing that comparing the official version is too unfair because the development version is already twice as fast as the official one due to memory management optimizations, better hash functions and other optimisations.

We use two examples for these tests. Other examples such as dacapo are too small and they are irrelevant here though the results are still consistent. We focus on a *cup* example (a combinatorial problem) and the *bus coupler* example (a communication protocol). Table 3 shows the obtained results. These numbers correspond to the whole state-space construction for the "bus coupler" example (property `A[] true`), and to the reachability property `E<> cups[2] == 4 and y <= 30` for the cups example because the whole state-space is too large.

Table 3. Experimental results.

Version	Bus coupler	Cups
UPPAAL	617s - 695M	52s - 116M
New - copy	422s - 655M	46s - 113M
New - shared	350s - 167M	44s - 27M

We chose the best appropriate options to have fair comparisons: we used `-H273819,273819` to have large passed and waiting lists for the UPPAAL run.

These tables are not resized dynamically as in the new versions and a good large size have a significant impact on speed (2x). Furthermore we used the options `-Ca`. The `C` says not to use the compact data structure, so that zones are manipulated and saved as DBMs. The `a` enables active clock reduction, which reduces memory consumption. Our new implementation does not take advantage of this yet because this information is not transmitted to the storage.

Depending on the careful options given to UPPAAL our new implementation gives improvements of up to 80% in memory and improves speed significantly. The memory gain is expected due to the showed sharing property of data. The speed gain (in spite of the overheads) comes from only having a single hash table and from the zone union structure: the discrete test is done only once, then comes only inclusion checks on all the zones in one union. This is showed by the results of the simple copy version.

8 Conclusions

In this paper we have presented a pipeline architecture. The idea borrowed from the computer graphics domain is appropriate for model-checking engines. Furthermore we have unified the traditional passed and waiting lists used in reachability algorithms, we have changed the representation of states towards a more symbolic one with the union of zones, and finally we have showed the sharing property of data composing a state and taken advantage of it.

This work paves the way for a new version of the UPPAAL engine with full support for hierarchy. These new structures support it. Furthermore these new structures allow new optimization experiments and they are flexible enough to support all the previous list representations currently being ported.

Future work is now to continue to develop UPPAAL with hierarchy support and to combine different zone representations with the sharing implementation we have, in particular the minimal constraint representation.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. In *Foundations of Software Engineering*, pages 175–188, 1998.

[3] G. Behrmann, K. G. Larsen, H. R. Andersen, H. Hulgaard, and J. Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. *Formal Methods in System Design*, 21(2):225–244, 2002.

[4] Gerd Behrmann, Ansgar Fehnker, Thomas S. Hune, Kim Larsen, Paul Petterson, and Judi Romijn. Efficient guiding towards cost-optimality in uppaal. In *Proc. of TACAS'2001*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[5] Gerd Behrmann, Thomas Hune, and Frits Vaandrager. Distributed timed model checking - How the search order matters. In *Proc. of 12th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Chicago, Juli 2000. Springer-Verlag.

[6] Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *Proceedings of the 12th Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[7] Michael Benedikt, Patrice Godefroid, and Thomas W. Reps. Model checking of unrestricted hierarchical state machines. In *Automata, Languages and Programming*, pages 652–666, 2001.

[8] Johan Bengtsson. Reducing memory usage in symbolic state-space exploration for timed systems. Technical Report 2001-009, Uppsala University, Department of Information Technology, May 2001.

[9] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Are timed automata updatable? In *Proceedings of the 12th Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[10] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. In *Transactions on Computers*, volume C-35 no. 8 of *IEEE*, August 1986.

[11] C.Daws and S.Yovine. Reducing the number of clock variables of timed automata. In *Proceedings of the 1996 IEEE Real-Time Systems Symposium, RTSS'96*. IEEE Computer Society Press, 1996.

[12] S. Christensen and L.M. Kristensen. State space analysis of hierarchical coloured petri nets. In B. Farwer, D.Moldt, and M-O. Stehr, editors, *Proceedings of Workshop on Petri Nets in System Engineering (PNSE'97) Modelling, Verification, and Validation*, number 205, pages 32–43, Hamburg, Germany, 1997. Universität Hamburg, Fachbereich Informatik.

[13] Alexandre David, Oliver Müller, and Wang Yi. Formal verification uml statecharts with real time extensions. In *Proceedings of FASE 2002 (ETAPS 2002)*, volume 2306 of *Lecture Notes in Computer Science*, pages 218–232. Springer-Verlag, 2002.

[14] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[15] Gerard J. Holzmann. On limits and possibilities of automated protocol analysis. In *Proc. 7th IFIP WG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification*, pages 137–161, 1987.

[16] Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.

[17] Sofiene Tahar, Paul Curzon, and Iskander Kort. Hierarchical formal verification using a mdg-hol hybrid tool. In *IFIP CHARME 2001*, volume 2144 of *Lecture Notes in Computer Science*, pages 244–258. Springer-Verlag, September 2001.

[18] Howard Wong-Toi. *Symbolic Approximations for Verifying Real-Time Systems*. PhD thesis, Standford University, 1995.