# A Tool Architecture for the Next Generation of Uppaal

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

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

**Abstract.** We present the design of the model-checking engine and internal data structures for the next generation of Uppaal. The design is based on a pipeline architecture where each stage represents one independent operation in the verification algorithms. The architecture is based on essentially one shared data structure to reduce redundant computations in state exploration, which unifies the so-called passed and waiting lists of the traditional reachability algorithm. In the implementation, instead of using standard memory management functions from general-purpose operating systems, we have developed a special-purpose storage manager to best utilize sharing in physical storage. We present experimental results supporting these design decisions. It is demonstrated that the new design and implementation improves the efficiency of the current distributed version of Uppaal by about 60% in time and 80% in space.

## 1 Introduction

Based on the theory of timed automata [1], a number of verification tools have been developed for timed systems in the past years [7, 22]. Various efficient algorithms and data structures, e.g. techniques for approximative analysis[21], state space reduction[18], compact data structures[4], clock reduction [10] and other optimisations, for timed automata are available. However, there has been little information on how these techniques fit together into a common efficient architecture.

This paper provides a view of the architecture and some optimisations 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 technology. As such, it is important for the tool to provide a flexible architecture that allows experimentation. It should allow *orthogonal* features to be integrated in an orthogonal manner to evaluate various techniques within a single framework and investigate how they 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

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

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 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.

Furthermore states are not generated independently from each other. This means the same sets of locations will be explored several times with different sets of variables. The same holds for the variable and the symbolic representation of time. We show how to take advantage of this in the *storage* layer of the engine.

We present a flexible architecture in the form of a pipeline. 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. We present results of combining the two main data structures, the waiting list and the passed list, into a single data structure. We show how this improves speed and memory usage. Finally, we show with a storage layer the effect of sharing common data of states, thereby reducing the memory usage by up to 80%. In particular the sharing property holds for the location and variable vectors, and the zones.

*Outline* Section 2 summarises the definition of timed automata, the semantics, and the timed automaton reachability algorithm. In section 3 we present the pipeline 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 state data is discussed in section 5. We present experimental results in section 6. We conclude the paper with a summary of results and related work.

## 2 Notations

In this section we summarise 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 $(L, l_0, E, I)$, where $L$ is a set of locations, $l_0 \in L$ is the initial location, $E \subseteq L \times (B(C) \times 2^C) \times L$ is a set of edges between locations with guards and clocks to be reset, 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 $(L, l_0, E, I)$ over $C$ is defined as a transition system $(S, s_0, \rightarrow)$, where $S = L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation such that:*

- $(l, u) \rightarrow (l, u + d)$ *if* $u \in I(l)$ *and* $u + d \in I(l)$
- $(l, u) \rightarrow (l', u')$ *if there exists* $e = (l, g, r, l') \in E$ *s.t.* $g$ *holds,* $u' = [r \mapsto 0]u$, *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 an 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* $Z_0 = I(l_0) \wedge \bigwedge_{x,y \in C} x = y = 0$ *be the initial zone. The symbolic semantics of a timed automaton* $(L, l_0, E, I)$ *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,* $s_0 = (l_0, Z_0)$ *is the initial state,* $\Rightarrow = \{(s, s') \in S \times S \mid \exists e = (l_1, g, r, l_2), t : s \overset{e}{\Rightarrow} t \overset{\delta}{\Rightarrow} s'\}$ *: is the transition relation, and:*

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

*where* $Z^{\uparrow} = \{u + d \mid u \in Z \wedge d \in \mathbb{R}_{\geq 0}\}$ *(the* future *operation), and* $r(Z) = \{[r \mapsto 0]u \mid u \in Z\}$. *The function norm* : $\mathbf{N} \times B(C) \rightarrow B(C)$ *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, 8]. Given the symbolic semantics it is straightforward to construct the reachability algorithm, shown in Fig. 1.

Note that the above definitions can be extended in the standard way to networks of automata (using a location vector), timed automata with finite data variables (using a variable vector) and to hierarchical timed automata [13].

## 3 Architecture

The seemingly simple algorithm of Fig. 1 turns out to be rather complicated when implemented. 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 and memory usage.

```
waiting = {(l_0, Z_0 ∧ I(l_0))}
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.

The architecture of UPPAAL has changed a lot over time. Some years ago UPPAAL was a more or less straightforward implementation of the timed automaton reachability algorithm annotated with conditional tests on features or options. Although it was simple, it had several disadvantages:

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

The architecture of UPPAAL is constantly restructured in order to facilitate new designs and algorithms, see Fig. 2 for the latest incarnation. The main goals of the design are speed and flexibility. The bottom layer providing the system and symbolic 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 idea of our pipeline architecture comes from computer graphics. In pipeline terms our architecture is composed of the connection of the *filters* and *buffers* components. Intuitively a filter has a *put* method to receive data. The result is then sent to the next component. A buffer is a purely passive component that awaits for data with a *put* method and offers data with a *get* method. A pump, ommitted here for simplicity, pumps data from a buffer and sends it to a serie of connected filters ending on the starting buffer. This is a data pipeline

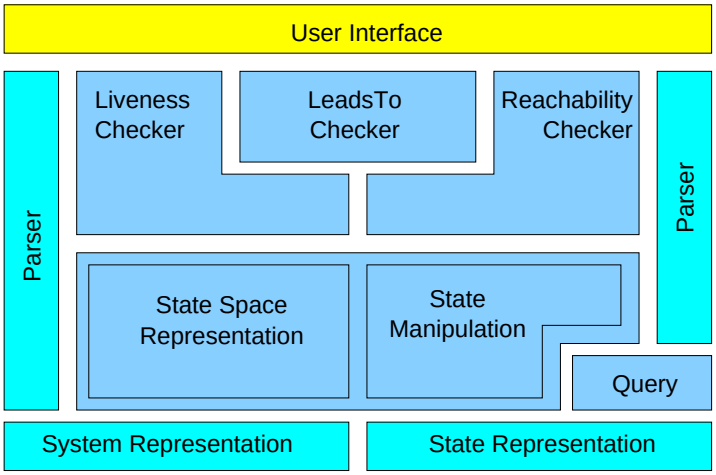


**Fig. 2.** UPPAAL uses a layered architecture. Components for representing the input model and a symbolic state are placed at the bottom. The state space representations are a set of symbolic states and together with the state operations they form the next layer. The various checkers combine these operations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

and there is no concurrency involved in contrast with pipeline designs seen in audio and video processing.

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. It 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 [10], 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 once, 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:*

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

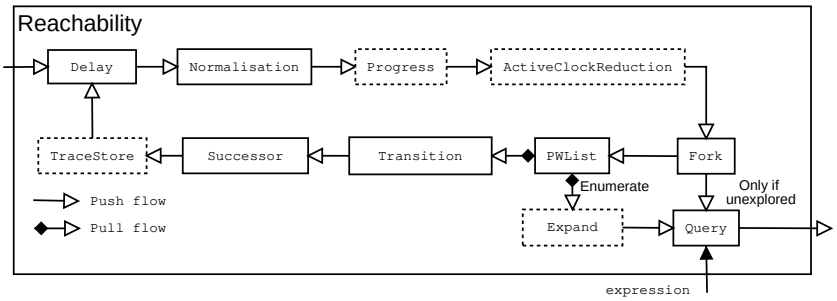


**Fig. 3.** The reachability checker is actually a compound object consisting of a pipeline of filters. Optional elements are dotted.

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

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

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 [18]. 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.[2] Still, this is more effective than starting over.

The number of unnecessary 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

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

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.[3] The overhead of using this array was much higher than the call overhead caused by the pipeline architecture.

## 4 Unifying the Passed list and Waiting List

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

### 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 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 and then moved to the passed list. The main purpose of the passed list is to ensure termination and also 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 being a subset or superset of the state to be added. In this case one of the two states can be discarded [5]. 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: (i) states are looked up in a hash table twice, and (ii) the waiting list might contain a large number of states that have previously been explored though this is not noticed until the state is moved to the passed list thus wasting memory.

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

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 available 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* [3, 2]. The ordering is orthogonal to the storage structure and can be combined with any data representation.

$Q = PW = \{(l_0, Z_0 \wedge I(l_0))\}$
**while** $Q \neq \varnothing$ **do**
  $(l, Z) =$ select state from $Q$
  $Q = Q \setminus \{(l, Z)\}$
  **if** $testProperty(l, Z)$ **then return true**
  $\forall (l', Z') : (l, Z) \Rightarrow (l', Z')$ **do**
    **if** $\forall (l', Y') \in PW : Z' \not\subseteq Y'$ **then**
      $PW = PW \cup \{(l', Z')\}$
      $Q.append(l', Z')$
    **endif**
  **done**
**done**
**return false**

**Fig. 4.** Reachability algorithm using the unified PWList. In the reference implementation (sub-section 4.2) $Q$ only contains references to the entries in $PW$.

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.

### 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 zones 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 independently of the exploration order. This order depends on the waiting queue that keeps state references.

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

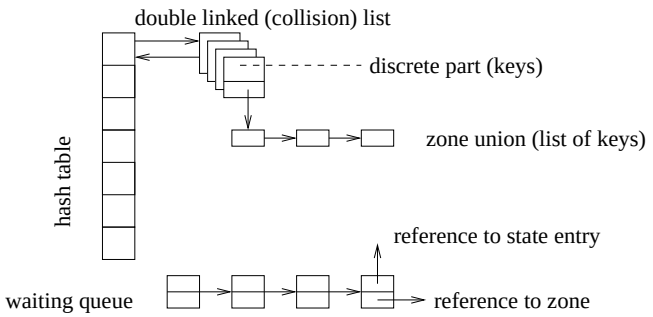


**Fig. 5.** Reference implementation of PWList.

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 [4]. Besides, this representation avoids any discrete state duplicates. The zones share the same discrete parts here. 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. At first glance it seems that the unification would not gain anything from a relatively small waiting list compared to the passed list (in most cases). However the costs of look-ups in small or large hash-tables are about the same, and we need one look-up instead of two.

The *put* 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 *get* operation consists of popping a state reference and checking for its validity (a simple flag).

### 4.3 Experiments

To isolate the impact of the unified list, we instrument the reference implementation. We use the same experiments presented in section 6 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

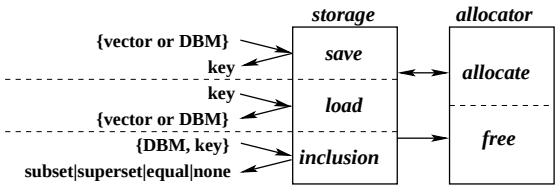


**Fig. 6.** The interface of the storage with the allocator underneath.

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 6.

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

**Table 1.** Percentage of waiting states of the inclusion detections. The new states are compared to the waiting states: they may include those (superset result) or may be included in those (subset result).

## 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 variables used for certain algorithms, i.e. guiding [2]. 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 to avoid 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. We have implemented two variants of this storage, namely one with simple copy and the other one with data sharing. Other particular algorithms aimed at reducing the memory footprint such as *convex hull approximation* [21] or *minimal constraint representation* [18] are possible implementations. These will be ported from the UPPAAL code base.

It is important to notice that UPPAAL implements a minimal constraint representation based on graph reduction. This reduction gives 20-25% gain in memory. It can give even more gain in addition to the shared storage implementation, but it is not implemented 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 saved without their diagonal. The diagonal contains the constraints $x_i - x_i \leq 0$ which do not need to be copied.[4]

### 5.2 Shared Storage

To investigate how data might be shared, 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. These results hold through different examples. This 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 obtained results justified this shared storage implementation.

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

[4] A DBM representing a non empty zone has always its diagonal set to 0. We store only non empty zones, hence we don't need to copy this diagonal.

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.

| 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% |

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

## 6 Experiments

We conduct the experiments on the development version 3.3.24 of UPPAAL without guiding on a Ultra SparcII 400MHz with 4GB of memory. This version incorporates the pipeline and is already twice as fast as the official version due to memory optimization such as reduced number of copies. Here we compare results without and with the PWList structure.

We use an audio protocol [6] (audio), a TDMA protocol [20] (dacapo), an engine gear controller [19] (engine), a combinatorial problem (cups), a field bus communication protocol [14] (different parts BC, master, and slave), and a production plant with three batches [17]. Table 3 shows time and space to generate the whole state space, i.e. the property `A[] true`, except for cups where the reachability property `E<> cups[2] == 4 and y <= 30` is used because the whole state space is too large. Time results under 0.5s are reported as 0.5s in the table. The result $> 4G$ means the verifier crashed because it ran out of memory.

We choose the options `-Ca` to use DBM representation with active clock reduction. Our implementation does not take full advantage of this because dynamic sized-DBM is not supported in the model-checker. Concerning the four last large examples we used the flag `-H273819,273819` to increase manually the size of the hash tables. Default sizes give twice longer verification times.

Depending on the careful chosen options given to UPPAAL our new implementation gives improvements of up to 80% in memory. If we take into account the factor 2 in speed and this improvement we obtain about 60% speed gain.

| | No PWList | PWList - copy | PWList - shared |
|---|---|---|---|
| audio | 0.5s 2M | 0.5s 2M | 0.5s 2M |
| engine | 0.5s 3M | 0.5s 4M | 0.5s 5M |
| dacapo | 3s 7M | 3s 5M | 3s 5M |
| cups | 43s 116M | 37s 107M | 36s 26M |
| BC | 428s 681M | 359s 641M | 345s 165M |
| master | 306s 616M | 277s 558M | 267s 153M |
| slave | 440s 735M | 377s 645M | 359s 151M |
| plant | 19688s $> 4G$ | 9207s 2771M | 8513s 1084M |

**Table 3.** Experimental results.

The memory gain is expected due to the showed sharing property of data. The speed gain comes from only having a single hash table and from the zone union structure: the discrete test is done only once and then inclusion checks is done on all the zones in one union. This is showed by the results of the simple copy version. The plant example has 9 clocks and 28 integer variables. The results show the gain in avoiding discrete duplicates. The amount of shared data is less than in other examples. Slight time improvements of the shared version comes from the smaller memory footprint only since there is a computation overhead. We gain on the page and cache faults.

For small examples the results are identical with all versions (results show allocated memory, less memory is used). The results scale with the size of the models, in particular the sharing property of the data holds.

## 7 Conclusions and Related Work

We have presented a pipeline architecture for the design of a real time model checker based on reachability analysis. The idea of using pipeline is from computer graphics. It is simple, versatile, and easy to maintain. The architechture has been implemented based on a shared data structure unifying the passed and waiting lists adopted in the traditional reachability analysis algorithms for finite state systems. We have also developed a special-purpose memory manager for the architecture to best utilize sharing in physical representation (storage) of logical structures adopted in the verification algorithms.

The work presented in this paper provides a platform for integration of various techniques developed in recent years for efficient analysis of timed systems. It paves the way for a new version of the UPPAAL engine with full support for hierarchical models.

*Related work* The state space storage approach presented in this paper is similar to the one in [11] for hierarhical coloured Petri nets. Both approaches share similarities with BDDs [9] in that common substructures are shared, but avoid the overhead of the fined grained data representation of BDDs. The zone union used in our state representation is a simple list of zones. A more elaborate representation is the CDD [4] that can be used efficiently for analysis. However CDDs pose 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 engine specific data connected to symbolic states. The passed/waiting list unification has been applied to Petri Nets [12] for the purpose of distributed model-checking. Our approach aims at reducing look-ups in the hash table and eliminating waiting states earlier. The particular implementation of the storage that shares data is different from the *state compression* used in Spin [16]. In Spin a *global state descriptor* represents a state and it holds a descriptor for the variables, followed by descriptors for every processes and channels. The user may choose the number of bits for these descriptors, which naturally limits the range of these descriptors. Our representation holds one descriptor for the locations, one for the variables, and one for the zones. The variable sharing is the only similarity. Locations and variables are treated equally as data vectors and are shared as such. It is important to notice that compression is orthogonal and compatible with this representation.

## Acknowledgments

As this work was done on the current UPPAAL engine, we are grateful to all its developers. The questions of unification and memory management had been discussed in our group previously. We would like to thank Johan Bengtsson for discussions and his work in implementing the current version of the UPPAAL engine as well as nice scripts to collect statistics.

## References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] Johan Bengtsson, W.O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. Number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.

[7] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995.

[8] 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.

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

[10] 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.

[11] 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.

[12] Gianfranco F. Ciardo and David M. Nicol. Automated parallelization of discrete state-space generation. In *Journal of Parallel and Distributed Computing*, volume 47, pages 153–167. ACM, 1997.

[13] Alexandre David, Oliver Möller, 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] Alexandre David and Wang Yi. Modeling and analysis of a commercial field bus protocol. In *Proc. of the 12*th *Euromicro Conference on Real Time Systems*, pages 165–172. IEEE Computer Society, June 2000.

[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] Gerard J. Holzmann. The model checker spin. In *IEEE Transactions on Software Engineering*, volume 23, may 1997.

[17] Thomas Hune, Kim G. Larsen, and Paul Pettersson. Guided Synthesis of Control Programs Using UPPAAL. In Ten H. Lai, editor, *Proc. of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation*, pages E15–E22. IEEE Computer Society Press, April 2000.

[18] 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.

[19] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller. In *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 281–297. Springer–Verlag, March 1998.

[20] Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Startup Mechanism. In *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, pages 235–242, December 1997.

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

[22] Sergio Yovine. Kronos: A verification tool for real time systems. In *Int. Journal on Software Tools for Technology Transfer*, pages 134–152, Oct 1997.