Unification & Sharing in Timed Automata Verification

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

[1] Department of Information Technology, Uppsala University, Sweden
[2] Department of Computer Science, Aalborg University, Denmark

Abstract. We present work on unifying the two main data structures involved during reachability analysis of timed automata. We also present result on sharing common elements between states. The experimental evaluations show speedups of up to 60% and memory savings of up to 80% compared to previous implementations.

1 Introduction

Timed automata (TA) is a popular formalism for modelling real-time aspects. The distinctive feature of TA is the use of clocks. Clocks are non-negative real valued variables, that can be compared and reset, and which increase at identical rates during delay transitions. A number of verification tools for TA exist. Like all verification tools, they suffer from the state explosion problem. In addition, they must deal with the infinite state-space of TA (due to the real valued clocks). Most tools use a pseudo-explicit state-space exploration algorithm based on *zones*. Zones describe infinite sets of clock valuations, and the state-space is represented by pairs (l, Z) called *symbolic states* containing the current *location* and the zone (more generally, the state-space of a network of TA extended with bounded integer variables might be represented by triples $(\boldsymbol{l}, \boldsymbol{\nu}, Z)$ containing a location vector $\boldsymbol{l}$, a variable vector $\boldsymbol{\nu}$ and a zone Z).

During state-space exploration, it is for reasons of termination and efficiency necessary to keep track of both which symbolic states have been explored as well as which still need to be explored. In this paper we present results on unifying these two data structures into a common structure. We also present orthogonal results on sharing common location vectors, variable vectors and zones between symbolic states. We will motivate these decisions and evaluate them through experiments in a prototype implementation based on the real-time verification tool UPPAAL.

Related work The sharing approach presented in this paper is similar to the one in [1] for hierarhical coloured Petri nets. Both approaches share similarities with BDDs in that common substructures are shared, but avoid the overhead of the fine grained data representation of BDDs. The unification has been applied to Petri Nets [2] for the purpose of distributed model-checking. Our approach aims at reducing look-ups in the hash table and eliminating waiting states earlier. To our knowledge, there has been no work on these issues in the context of TA.

2 Unification and Sharing

During state-space exploration, there is a fundamental need to maintain two sets of symbolic states: States that need to be explored (the waiting list) and states that have been explored (the passed list). States are taken from the waiting list, compared to the states in the passed list, and if unexplored added to the passed list while successors are added to the waiting list.

Since symbolic states are sets of concrete states, it makes sence to define inclusion between states having the same location and variable vector, *i.e.*, $(\boldsymbol{l}, \boldsymbol{\nu}, Z) \subseteq (\boldsymbol{l}, \boldsymbol{\nu}, Z')$ iff $Z \subseteq Z'$. We say that $(\boldsymbol{l}, \boldsymbol{\nu}, Z)$ is covered by $(\boldsymbol{l}, \boldsymbol{\nu}, Z')$. Three observations are essential for good performance:

– When determining whether a state s has already been explored by comparing it to the passed list, rather than searching for states identical to s, we might as well look for states covering s.
– When adding a state s to the waiting list, there is no need to add s if it is covered by an existing state.
– When adding a state s to the passed list or the waiting list, all states covered by s can be removed.

The traditional approach to implementing these operations is to use a hash table and define the hash function on the location vector and variable vector, but not on the zone. Thus, it is easy to find states with the same location vector and variable vector. This approach has a major drawback: It is often the case that many states in the waiting list are covered by states in the passed list, but this is not realized until the states are moved from the waiting list to the passed list. This enlarges the waiting list, wasting memory and increasing the cost of adding new states. One solution would be to move the passed list lookup s.t. states are added to the passed list and the waiting list at the same time, see Fig. 1. Then the waiting list is guaranteed to only contain unexplored states. However, this solution is undesirable: First, states in the waiting list are duplicated. Second, although it is not apparent from Fig. 1, adding a state s to the passed and waiting lists still requires a partial traversal of those data structures in order to eliminate states covered by s.

```
$WL = PL = \{s_0\}$
while $WL \neq \varnothing$ do
  $s$ = select and remove state from $WL$
  if $s \models \varphi$ then return true
  $\forall s' : s \Rightarrow s'$ do
    if $\forall s'' \in PL : s' \nsubseteq s''$ then
      $PL = PL \cup \{s\}$
      $WL = WL \cup \{s\}$
    endif
  done
done
return false
```

Fig. 1. Explicit state reachability algorithm. States are inserted into both the passed list and the waiting list.

Instead we propose to unify the two data structures into a single data structure – for lack of a better name we call it the *unified list*. When adding a state s to the unified list, it is compared to existing states: If s is covered by any of the existing states, then s is not added. Otherwise, all states covered by s are removed and s is added. Internally, it is still necessary to keep track of which states have been explored and when retrieving a state from the list, it is marked

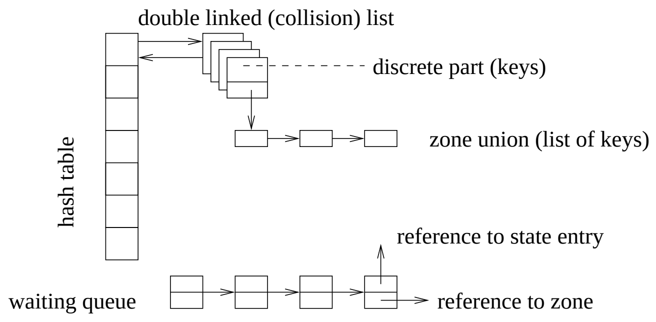

Fig. 2. Implementation of a unified passed and waiting list.

as explored, but not actually removed. Figure 2 shows one possible implementation of the unified list data structure. A hash table provides fast access to a linked list of zones sharing the same location vector and variable vector. At the same time, a list (ordered either in FIFO or LIFO depending on the desired exploration order) of references to unexplored states is maintained.

Unifying the passed and waiting lists reduces the number of needless states, *i.e.* states covered by previously explored states, stored on the waiting list. It does not reduce the amount of memory needed to store each symbolic state. Our second proposal is to share common location vectors, variable vectors, and zones among states. This is motivated by the results shown in Tab. 1. This can be implemented by storing location vectors, variable vectors, and zones in different hash tables. The unified list then only maintains references (keys) to the elements in those hash tables.

3 Experiments

We conduct our experiments on development version 3.3.24 of Uppaal on an Ultra Sparc II 400MHz with 4GB of memory. This version incorporates a new architecture and is already twice as fast as the official 3.2 version. Here we compare results without and with the unified list structure for an audio protocol (Audio), a TDMA protocol (Dacapo), an engine gear controller (Engine), a combinatorial problem (Cups), a field bus communication protocol (different parts BC, Master,

Table 1. The number of unique location vector, variable vectors and zones, measured in percent for four different examples. The lower the number, the more copies of the same data there are.

Model	Unique locations	Unique variables	Unique zones
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. Experimental results for 8 examples without unification, with unification, and with unification and sharing.

Model	Before		Unification		Unication & Sharing	
Audio	$\leq 0.5s$	2M	$\leq 0.5s$	2M	$\leq 0.5s$	2M
Engine	$\leq 0.5s$	3M	$\leq 0.5s$	4M	$\leq 0.5s$	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

and Slave), and a production plant with three batches. We refer to the UPPAAL web-site for references to these examples.

Table 2 shows time and space requirements to generate the whole state-space, except for cups where a reachability property is used because the whole state space is too large. The result $> 4G$ means the verifier crashed because it ran out of memory. In all examples, zones were represented using the DBM data structure, and active clock reduction was enabled. In the four last examples, the hash table size of the passed list and waiting list was enlarged to 273819 (using the default size doubles the verification time). The unified list implementation automatically resizes the hash table and does not suffer from this problem.

Focusing on the experiments with unification and without sharing, we see an increase in speed and a slight reduction in memory usage. This is due to not wasting space on storing states in the waiting list, that are not going to be explored anyway and due to keeping a list of zones having the same location and variable vector. The latter is also responsible for the speedup. This is in particular the case in the plant example, which has 9 clocks and 28 integer variables. Focusing on the experiments with sharing, we see a significant reduction in memory usage of up to 80%. We also observe a slight speedup, which we expect is due to the smaller memory footprint - this seems to compensate for the computational overhead caused by maintaining extra hash tables. For small examples, the results are identical for all versions. The results scale with the size of the models, in particular the sharing property of the data holds. In total, we observe a speedup of up to 60%.

Acknowledgments go to Ahiable Antoinette and Johan Bengtsson.

References

[1] 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 PNSE'97*, number 205, pages 32–43, Hamburg, Germany, 1997.

[2] 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.