Distributing Timed Model Checking — How the Search Order Matters[⋆]

Gerd Behrmann[1], Thomas Hune[2], and Frits Vaandrager[3]

[1] Basic Research in Computer Science, Aalborg University
Frederik Bajersvej 7E, 9220 Aalborg East, Denmark
`behrmann@cs.auc.dk`

[2] Basic Research in Computer Science, Aarhus University
Ny Munkegade, Bygning 540, 8000 Århus C, Denmark
`baris@brics.dk`

[3] Computing Science Institute, University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
`fvaan@cs.kun.nl`

Abstract. In this paper we address the problem of distributing model checking of timed automata. We demonstrate through four real life examples that the combined processing and memory resources of multiprocessor computers can be effectively utilized. The approach assumes a distributed memory model and is applied to both a network of workstations and a symmetric multiprocessor machine. However, certain unexpected phenomena have to be taken into account. We show how in the timed case the search order of the state space is crucial for the effectiveness and scalability of the exploration. An effective heuristic to counter the effect of the search order is provided. Some of the results open up for improvements in the single processor case.

1 Introduction

The technical challenge in model checking is in devising algorithms and data structures that allow one to handle large state spaces. Over the last two decades numerous approaches have been developed that address this problem: symbolic methods such as BDDs, methods that exploit symmetry, partial order reduction techniques, etc [4]. One obvious approach that has been applied successfully by a number of researchers is to parallelize (or distribute) the state space search [1, 15]. Distributed reachability analysis and state-space generation has also been investigated in the related field of performance analysis in the context of stochastic Petri nets [3, 8] (see the second paper for further references). Since the state-of-the-art in model checking and performance analysis is still progressing very fast, it does not make sense to develop parallel or distributed tools from scratch. Rather, the goal should be to view parallelization as an orthogonal feature, which can always be easily added when the appropriate hardware is available.

[⋆] Research supported by Esprit Project 26270, Verification of Hybrid Systems (VHS).

To some extend this goal has been achieved in the work of [3, 15, 8], all with very similar solutions. Stern and Dill [15], for example, present a simple but elegant approach to parallelize the Murφ tool [5] using the message passing paradigm. In parallel Murφ, the state table, which stores all reached protocol states, is partitioned over the nodes of the parallel machine. Each node maintains a work queue of unexplored states. When a node generates a new state, the *owning* node for this state is calculated with a hash function and the state is sent to this node; this policy implements randomized load balancing. In the case of Murφ, the algorithm of Stern and Dill achieves close to linear speedup. We applied the approach of Stern and Dill to parallelize UPPAAL[11], a model checker for networks of extended timed automata. We experimented with parallel UPPAAL using four existing case studies: DACAPO [13], communication [7] and power-down [6] protocols used in B&O audio/video equipment, and a model of a buscoupler.

In the case of timed automata, the state space is uncountably infinite, and therefore one is forced to work with *symbolic* states, which are finite representations of possibly infinite sets of concrete states. A key problem we had to face in our work is that the number of symbolic states that has to be explored depends on the order in which the state exploration proceeds. In particular, the number of states tends to grow if state space exploration is parallelized. The main contribution of this paper consists an effective heuristic which takes care that the growth of the number of states remains within acceptable bounds. As a result we manage to obtain close to double linear speedups for the B&O protocols and the buscoupler. For the DACAPO example the speedup is not so good, probably because the state space is so small that only a few nodes are involved in the computation at a time. Some of the results open up for improvements in the single processor case.

The rest of this paper is structured as follows: Section 2 reviews the notion of timed automata. Section 3 describes our approach to distributed timed model checking, Section 4 presents experimental results, and Section 5 summarizes some of the conclusions.

2 Model Checking Timed Automata

In this section we briefly review the notion of timed automata that underlies the UPPAAL tool. For a more extensive introduction we refer to [2, 10]. For reasons of simplicity and clarity in presentation we have chosen to only give the semantics and exploration algorithms for timed automata. The techniques described in this paper extend easily to networks of timed automata, even when extended with shared variables as is the case in UPPAAL.

Timed automata are finite automata extended with real-valued clocks. Figure 1 depicts a simple two node timed automaton. As can be seen both the locations and edges are labeled with constraints on the clocks. Given a set of clocks C, we use $\mathcal{B}(C)$ to stand for the set of formulas that are conjunctions of atomic constraints of the form $x \bowtie n$ and $x - y \bowtie n$ for $x, y \in C$, $\bowtie \in \{<, \leq, =, \geq, >\}$

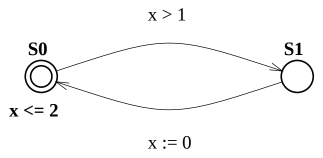

Fig. 1. A simple two state timed automaton with a single clock x.

and n being a natural number. Elements of $\mathcal{B}(C)$ are called clock constraints over C. $P(C)$ denotes the power set of C.

Definition 1. *A timed automaton A over clocks C is a tuple (L, l_0, E, I) where L is a finite set of locations, l_0 is the initial location, $E \subseteq L \times \mathcal{B}(C) \times \mathcal{P}(C) \times L$ is the set of edges, and $I : L \to \mathcal{B}(C)$ assigns invariants to locations. In the case of $(l, g, r, l') \in E$, we write $l \xrightarrow{g,r} l'$.*

Formally, clock values are represented as functions called clock assignments from C to the non-negative reals $\mathbf{R}_{\geq 0}$. We denote by $\mathbf{R}^C$ the set of clock assignments for C. The state space of an automaton A is $L \times \mathbf{R}^C$. The semantics of a timed automaton A is defined as a transition system:

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

where for $d \in \mathbf{R}$, $u + d$ maps each clock x in C to the value $u(x) + d$, and $[r \mapsto 0]u$ denotes the assignment for C which maps each clock in r to the value 0 and agrees with u over $C \backslash r$. In short, the first rule describes *delay* and the second *edge* transitions. It is easy to see that the state space is uncountable. However, it is a well-known fact that timed automata have a finite-state symbolic semantics [2] based on countable symbolic states of the form (l, D), where $D \in \mathcal{B}(C)$:

- $(l, D) \to (l, \mathrm{norm}(M, (D \wedge I(l))^{\uparrow} \wedge I(l)))$
- $(l, D) \to (l', r(g \wedge D \wedge I(l)) \wedge I(l'))$ if $l \xrightarrow{g,r} l'$.

where $D^{\uparrow} = \{u + d \mid u \in D \wedge d \in \mathbf{R}_{\geq 0}\}$ (the *future* operation), and $r(D) = \{[r \mapsto 0]u \mid u \in D\}$. The function norm : $\mathbf{N} \times \mathcal{B}(C) \to \mathcal{B}(C)$ normalizes the clock constraints with respect to the maximum constant M of the timed automaton. Normalizing the clock constraints guarantees a finite state space. We refer to [2, 10] for an in-depth treatment of the subject.

The state space exploration algorithm is shown in Fig. 2. Central to the algorithm are two data structures: the waiting list, which contains unexplored but reachable symbolic states, and the passed list, which contains all explored symbolic states. An important but in the literature often ignored optimization is to check for state coverage in both lists. Instead of only checking whether a

```
PASSED := $\emptyset$
WAITING := $\{(l_0, D_0)\}$
repeat
    get $(l, D)$ from WAITING
    if $D \not\subseteq D'$ for all $(l, D') \in$ PASSED then
        add $(l, D)$ to PASSED
        SUCC := $\{(l', D') : (l, D) \rightarrow (l', D') \wedge D' \neq \emptyset\}$
        for all $(l', D') \in$ SUCC do
            put $(l', D')$ in WAITING
        od
    end if
until WAITING = $\emptyset$
```

Fig. 2. Sequential symbolic state space exploration for timed automata.

symbolic state is already included in the list, UPPAAL searches for states in the list that either cover the new state or is covered by it. In the first case the new state is discarded and in the latter case it replaces the existing state covered by it. We will return to this matter in Section 3.

3 Distributed Model Checking of Timed Automata

The approach we have used for distributing the exploration algorithm is similar to the one presented in [3, 15, 8]. Each node executes the same algorithm (see Fig. 3) which is a variant of the sequential algorithm shown in Fig. 2. Since we assume a distributed memory model, all variables are local. Each node is assigned a part of the state space according to a distribution function mapping symbolic states to nodes. Whenever a new symbolic state is encountered it is sent to the node responsible for exploring and storing that particular state. Each time a state has been explored and its successors have been sent, all states waiting to be received are received and put into the waiting list. If there are no states in the waiting list, the node waits until a state arrives. Although all nodes run the same algorithm, each node knows its own id and one node is the *master* node. This node is responsible for calculating the initial state and sending it to the owning node, and for deciding when the verification has finished. The verification terminates when there are no more states waiting to be explored in the waiting lists and there are no messages in transit. When the master finds out that the verification is finished it sends a *termination* signal to all the nodes.

3.1 Nondeterminism and Search Orders

When exploring a state space using UPPAAL, one can choose between breadth-first or depth-first search order corresponding to a queue or a stack implementation of the waiting list, respectively. In a distributed search one must still choose

```
PASSED := $\emptyset$
WAITING := $\emptyset$
repeat
    receive states and place them in WAITING
    get $(l, D)$ from WAITING
    if $D \not\subseteq D'$ for all $(l, D') \in$ PASSED then
        add $(l, D)$ to PASSED
        SUCC := $\{(l', D') : (l, D) \rightarrow (l', D') \land D' \neq \emptyset\}$
        for all $(l', D') \in$ SUCC do
            send $(l', D')$ to $h(l')$
        od
    end if
until not terminate
```

Fig. 3. The distributed state space exploration algorithm.

whether each node uses a queue or a stack; we will call this "distributed breadth-first" and "distributed depth-first" order, respectively. This only tells in what mode the single nodes run. In general the search order will be nondeterministic and may change from execution to execution.

In a distributed breadth-first search the states are explored in order of arrival at each node. However, the order in which states arrive at a node (enter the waiting list) will differ between executions. Some reasons for this are varying communication delays, and different workloads on the nodes. This means that in general states will not be searched in breadth-first order.

The main difference between a depth-first search and a distributed depth-first search is that in the single processor case only one path is explored at a time while in the distributed case more paths are explored at the same time. This is because all successors of a state are generated and sent to their owning nodes, where the search is continued in parallel. When the waiting list is implemented as a stack small changes of the order in which states arrive may significantly change the search order. Assume two states α and β arrive at a node while it is exploring a state with α arriving last (so α will be on the top of the stack). The successors of α are generated and sent to their owning nodes. One or more of these may go to the node itself which means that they are explored before β (because states are received before a new state is popped from the waiting list), and the same for their successors and so on. It may thus occur that β has to wait a long time before it is explored even though it has arrived at almost the same time as α. Hence small changes in the order of arrival of states may change the search order drastically.

3.2 Why the Search Order Matters

In a distributed state space search the number of states explored (and thereby the work done) may differ from run to run. This is because whether a state is explored or not depends on the states encountered before. As an example, consider two states (l, D) and (l, D') with same location vector l but different time zones satisfying $D \subset D'$. If (l, D) arrives first and is explored before arrival of (l, D'), then (l, D') will also be explored since it is not covered by any state in the passed list (assuming that there are no other states covering it). Since the successors of (l, D') are very likely to have larger time zones than the successors of (l, D) these will also be explored later. However, if (l, D') arrives first and is explored before (l, D) arrives, then (l, D) will not be explored because it is covered by a state in the passed list. This also means that no successors of (l, D) will be generated or explored.

Earlier experiments with the sequential version of UPPAAL showed that breadth-first search is often much faster than depth-first search when generating the complete state space. This comes from the fact that depth-first search order causes higher degree of fragmentation of the zones that breadth-first order, resulting in a higher number of symbolic states being generated.

As noted above, the distributed algorithm neither realizes a strict breadth-first nor depth-first search. When using a queue on each node, the algorithm approximates breadth-first search. In fact, on a single node the search order will be breadth-first. As we increase the number of nodes, chances increase that the nondeterministic nature of the communication causes the ordering within the queue to be such that some states with a large depth (distance from the initial state) are explored before other states with a smaller depth. In cases where breadth-first is actually the optimal search order, increasing the number of nodes is bound to increase the number of symbolic states explored.

Since it seems that breadth-first order in most cases is the optimal search order we propose a heuristic for making a distributed breadth-first order closer to breadth-first order. The heuristic keeps the states in each waiting list ordered by depth, for example by using a priority queue. This guarantees that the state in the waiting list with the smallest depth is explored first. In Section 4, we will demonstrate that this heuristic drastically reduces the rate at which the number of symbolic states increases when the number of nodes grows. In some cases it actually decreases the number of states explored.

3.3 Distribution Functions and Locality

On one hand, a good distribution function should guarantee a uniform work load for the nodes, on the other hand it should reduce communication between nodes. Since these objectives in most cases contradict each other, one has to find a suitable tradeoff. We therefore considered several distribution functions.

As in [15], most of our results are based on using a hash function as the distribution function. However, to make the inclusion checks of the time zones in the waiting and the passed lists possible, states with the same location vector must be mapped to the same node. The hash value of a symbolic state is therefore only based on the location vector and not on the complete state.

One possible hashing function is the one already implemented in UPPAAL and used when states are stored in the passed list. It uniquely maps each state to an integer modulo the size of the hash table. Experiments have shown that it distributes location vectors uniformly. Trying to increase locality of the distribution function, it should be possible to use the fact that transitions only change a small part of the location vector and only some transitions change the integer variables. If we consider a state α and a successor β, we can expect most locations and integer variables in β to be the same as in α. Section 4 reports on experiments where the distribution function only hashes on part of the location vector or only on the integer variables.

Some experiments with model specific distribution functions were done, but it was extremely difficult to even approach the performance of the generic distribution functions. Finding effective model specific distribution functions requires much work and a thorough understanding of the given model.

Within UPPAAL the techniques described in [12] for reducing memory consumption by only storing loop entry points in the passed list are quite important for verifying large models. The idea is to keep a single state from every static loop (which are simple to compute). This guarantees termination while giving considerable reductions in memory consumption for some models. UPPAAL implements two variations of this techniques. The most aggressive one is described in [12] which only stores loop entry points. While reducing memory consumption this technique may increase the number of states explored, since certain states are explored more than once. A less aggressive approach is to also store all non-committed states (in which no automaton is in a committed location) in the passed list. Experiments show that this is a good compromise between space and speed.

We propose using this technique to increase locality in the exploration. Since non loop entry points are not stored on the passed list they might as well be explored by the node which computed the state in the first place instead of sending it to another node, thereby increasing locality. Consider, for example, a state α and its successor β. If β is not a loop entry point and therefore is not going to be stored on the passed list, we may as well explore β on the same node as α. Section 4 reports on experiments with this technique.

3.4 Generating Shortest Traces

An important feature of a model checker is its ability to provide good debugging information in case a certain property is not satisfied. For a failed invariant property this is commonly a trace to the state violating the invariant. Providing a short trace increases the value of a trace. One of the features of UPPAAL is that when the algorithm from Fig. 2 is used with a breadth-first search order, the trace to the error state is the shortest possible, since all states that can be reached with a shorter trace have been explored before. It would be nice to have this feature also in a distributed version of the tool. However, as described above, the order of a distributed state space search is non-deterministic, and this may lead to non-minimal traces. Fortunately, with little extra computational effort a shortest trace can be found regardless of the search order. The idea is to record for each symbolic state its "depth", i.e., the length of the shortest trace leading to this state. When a violating state is found the algorithm does not stop, but instead continues to search for violating states that can be reached with a shorter trace. We need to make sure that the inclusion checks performed on the waiting and passed lists do not discard potential violating states. When a new state (l, D) is added to the waiting or passed list, we normally compare it to every state (l, D') on the list, and if an inclusion exists we keep the larger of the two states. In order not to discard potential traces, we add the restriction that a state is only replaced/discarded if it does not have a smaller depth than the state it is compared to. The same idea is used for the decision whether or not to explore a state when looking it up in the passed list: we only decide not to explore a state if its clock constraints are included in the clock constraints of another state with the same location vector *and* at the same time does not have a smaller depth than the state it is included in. The corresponding line in the algorithm changes to:

```
if $D \not\subseteq D'$ or $\text{depth}(l,\ D) < \text{depth}(l,\ D')$ for all $(l, D') \in \text{PASSED}$ then
```

With these changes the algorithm in Fig. 3 can find shortest traces independently of the ordering used on the waiting list. As described above we have implemented a heuristic which approximates breadth-first search. In Section 4, we demonstrate that when using this heuristic the extra cost for finding the shortest trace is minor and we keep good speedups.

4 Experimental Results

For implementing communication between nodes we have used the *Message Passing Interface* (MPI) [14]. This facilitates porting and running the program on different kinds of machines and architectures. We have conducted experiments on a Sun Enterprise 10000 with 24 333Mhz processors, which has a shared memory architecture, and on a Linux Beowulf cluster of 10 450Mhz Pentium III CPUs.

4.1 Nondeterminism and Search Orders

One of the first examples the distributed UPPAAL was tried on, was a model of a batch plant [9] constructed to verify schedulability of a production process. The verification, which on one processor took several hours, surprisingly took less than five minutes on 16 nodes. This super linear speedup came as a surprise to us. Verifying schedulability in this model means searching for a state where all batches have been processed. For this particular model we had previously identified depth-first search as the fastest strategy on one node, and therefore we used a distributed depth-first search. In this particular model, the verification benefited from the nondeterministic search order. The distributed depth-first search did not find the same state as the verification on a single processor, and in fact the number of states searched was not the same in the two cases. It should be possible to achieve a similar effect with the sequential algorithm by introducing randomness into the search order. First experiments with using a kind of random depth-first search have been promising.

Because of this property of checking for a particular state (or a set of states), we have in the remaining experiments chosen to generate the complete state space of the given system using a distributed breadth-first search. Generating the complete state space reduces the impact of the nondeterministic search order because one cannot find a "lucky" path which finds the state searched for quickly. This makes the results from different runs comparable.

4.2 Speedup Gained

We have chosen to focus our experiments around four UPPAAL examples: the start-up algorithm of the DACAPO [13] protocol which is quite small but had some interesting behavior as will be discussed later; a communication protocol used in B&O audio/video equipment (CP)[7]; a power-down protocol also used in B&O audio/video equipment (PD)[6]; and a model of a buscoupler (which thus far has not been published). The reason not to look further at the model of the batch plant is that the state space was too big to be generated completely. All other known UPPAAL examples were also tried, but these were so small that the complete state space can be generated in a matter of seconds using a few processors, and were therefore considered too small to be of interest.

The examples were run on the Sun Enterprise on 1, 2, 3, 4, 5, 8, 11, 14, 17, 20 and 23 nodes; and on the Beowulf on 1 to 10 nodes to the extend it was possible (only the DACAPO model could be run on a single node because of memory usage). Since the search order (and thereby the work done) is non-deterministic we repeated one experiment several times. The observed running times[1] and number of states generated varied less than 3%. Running the experiments only once therefore seemed reasonable.

When generating the complete state space for a number of examples using distributed breadth-first search a general pattern occurred. In most cases the number of states generated increased with the number of nodes, and in all cases the smallest number of states was generated using one node. It therefore seems that in most cases breadth-first is *close to* the optimal search order for generating the complete state space. In most cases the increase in the number of states was minor (less than 10%), but for a few examples the increase was substantial. In the DACAPO example the number of states more than doubled — from 45000 states to more than 110000 states using 17 nodes (see Table 1).

To counter this effect, we applied the heuristic described in Section 3.2 and used a priority queue to order the states waiting on each node such that the states with the shortest path to the initial state is searched first. Not only did this counter the increase in the number of states, it actually decreased the number of states generated in some cases. This shows that there is still room for improvement with respect to the search order even when using a single processor. Table 1 and 2 show the effect of applying the heuristic to our examples. As

[1] When talking about the running time we always consider the time of the slowest node.

Table 1. States generated with (Priority) and without (FIFO) use of heuristic on Sun Enterprise.

#	States							
	DACAPO		CP		Buscoupler		PD	
	FIFO	Priority	FIFO	Priority	FIFO	Priority	FIFO	Priority
1	45001	44925	3466548	3010244	6502 804	6436543	7992048	7992098
2	45754	44863	5505161	3027728	8042882	6199274	8004165	8003477
3	69141	45267	5472878	3070491	8064519	6243785	8001670	7997859
4	62541	45177	5454067	3086016	8123748	6171125	8004717	8004439
5	78008	45667	5583368	3077890	8651090	6481067	8002412	7998607
8	77396	46510	5452888	3113378	8359647	6185288	8004898	8004898
11	84598	46318	5642463	3059169	8968257	6184329	8004888	8004892
14	108344	49741	5653134	3102709	8914300	6278855	8004888	8004888
17	110634	52247	5270822	3082967	9049252	6243571	8001813	7996979
20	98266	47573	5449055	3111333	9271401	6251283	8004881	8004880
23	104945	52457	5535724	3065916	9146026	6103629	8004714	8004651

Table 2. States generated with (Priority) and without (FIFO) use of heuristic on Beowulf.

#	States							
	DACAPO		IR		Buscoupler		PD	
	noorder	order	noorder	order	noorder	order	noorder	order
1	45858	45748	N/A	N/A	N/A	N/A	N/A	N/A
2	48441	46899	N/A	3028368	N/A	N/A	N/A	N/A
3	74882	47671	5605882	3053837	N/A	N/A	N/A	N/A
4	62398	47640	5533159	3058230	15832617	12794520	9473496	9409935
5	79899	47678	5454676	3060070	16637609	13603603	9432828	9287527
6	92678	49438	5684749	3133769	20443824	13896789	9511548	9482742
7	97065	49739	5702856	3074131	20329057	13797531	9513477	9441041
8	97662	50477	5358514	3106414	22430748	14442925	9527173	9488775
9	92642	49284	5449403	3071827	21086691	14455201	9535657	9515920
10	92400	48821	5532205	3060705	20704595	15507978	9526732	9500000

can be seen from the tables, the heuristic performs well in three of the four examples and in the PD example it has no effect. We also tried to use a distributed depth-first search order, and to 'reverse' the heuristic to first explore the states with the longest path to the initial state during distributed depth-first search. In both cases the number of states generated was increased substantially. Therefore these search orders were discarded for the remaining experiments.

An important question is of course how well the distribution of the search scales in terms of number of nodes. Tables 3 and 4 show the running times in seconds for the different examples on the Sun Enterprise and the Beowulf, respectively. When running on the Sun Enterprise we were able to generate the complete state space on a single node for all the examples. We can therefore calculate the speedup with respect to running on a single node. The speedups we have calculated are normalized with respect to the number of states explored, to clarify the effect of the distribution. The speedup for i nodes is calculated as

$$\frac{\textit{time on one node/states on one node}}{\textit{time on } i \textit{ nodes/states on } i \textit{ nodes}}$$

where *time on one node* is the time for generating the complete state space using the distributed version running on one node, and *time on i nodes* is the time of the slowest node when running on i nodes.

Table 3. Run time with (Priority) and without (FIFO) use of heuristic on Sun Enterprise.

#	Run time							
	DACAPO		CP		Buscoupler		PD	
	FIFO	Priority	FIFO	Priority	FIFO	Priority	FIFO	Priority
1	8.6	9.0	804.0	732.0	2338.6	2213.8	3362.8	3195.4
2	5.2	5.0	725.8	351.6	1506.5	861.4	1507.1	1101.2
3	5.4	3.7	446.4	238.6	773.0	559.4	943.0	649.8
4	3.9	2.9	317.9	175.2	596.4	413.4	713.4	467.6
5	4.0	2.5	266.9	142.0	501.2	342.5	453.5	373.1
8	2.8	2.1	152.8	86.8	283.0	202.3	231.6	226.9
11	2.6	1.9	121.7	65.3	221.4	148.0	159.9	161.4
14	2.7	2.0	95.5	53.9	172.2	118.0	127.3	133.4
17	2.7	2.1	74.2	43.1	145.2	97.7	106.9	102.4
20	2.4	2.3	66.5	38.8	127.6	83.6	93.0	92.1
23	2.2	2.4	60.2	34.3	112.4	72.7	76.9	79.6

For the DACAPO example the speedup decreases from being linear already in the case of 5 nodes. However, it only takes 2.5 seconds to generate the complete states space using 5 nodes. Since the states space is small not all nodes can be kept busy and relatively much time is spent to start and close down the exploration. Therefore, a poor speedup was to be expected. For the CP example

Table 4. Run time with (Priority) and without (FIFO) use of heuristic on Beowulf.

#	Run time							
	DACAPO		CP		Buscoupler		PD	
	FIFO	Priority	FIFO	Priority	FIFO	Priority	FIFO	Priority
1	3.88	4.15	N/A	N/A	N/A	N/A	N/A	N/A
2	3.20	3.16	N/A	682.57	N/A	N/A	N/A	N/A
3	3.49	2.37	934.44	349.86	N/A	N/A	N/A	N/A
4	2.88	2.02	540.19	218.94	1060.09	799.69	616.85	541.89
5	2.71	1.64	390.02	169.93	836.09	646.02	413.45	401.30
6	2.62	1.52	337.79	144.50	1796.23	563.08	453.20	377.39
7	2.55	2.47	285.30	124.69	811.78	476.69	343.49	315.69
8	2.51	1.39	200.50	97.84	782.28	440.87	283.07	274.41
9	2.23	1.38	178.75	87.38	619.84	394.91	244.72	242.16
10	2.00	1.19	173.07	82.44	536.74	387.27	214.03	217.98

the speedup is close to linear. However, for the buscoupler and the PD examples the speedup is super linear, which is surprising since the speedup has been normalized with respect to the total number of states. Figure 4 shows the graphs for the speedups of the CP and buscoupler examples. We are not sure about the reason for these super linear speedups. For the Sun Enterprise machine, accessing main memory is considered to be a bottleneck. When the number of nodes used in an exploration increases so does the amount of cache available (each node has 4Mb of cache). Since UPPAAL spends much time looking up states in the passed and waiting lists, faster access to larger parts of these lists may increase the speed substantially. This conjecture is supported by the fact that the examples with the largest number of states (and therefore most accesses to the passed and waiting lists) gain the largest speedup. The same kind of super linear speedups were not encountered by Stern and Dill [15]. As mentioned in their paper, Murφ has implemented a wide range of techniques for minimizing the state space. This means that, compared to UPPAAL, Murφ spends less time on looking up states and accessing memory, and therefore Murφ does not gain the same speedup from the larger cache.

On the Beowulf it was in most cases not possible to generate the complete state space using only one processor. We have therefore chosen to present the amount of work done, where work for i nodes is defined as *the time on i nodes* times i divided by *the number of states on i nodes*, to normalize with respect to the number of node generated. A horizontal then corresponds to a linear speedup. As expected the line for the DACAPO example increases, so we do not have a linear speedup. The speedup looks better for the CP on the Beowulf example but since we do not have the time on one node (this could not complete due to memory shortage) it is hard to judge whether the work is approaching the work in one node or really is decreasing below that. The same is the case for the buscoupler and the PD example. Figure 5 shows the work for the CP and buscoupler examples. One interesting point to notice is that for six nodes without the heuristic the buscoupler performs very poorly (we have no explanation for this behavior).

The explanations we suggest for the super linear speedups we encounter on the Beowulf are the same as for the Sun Enterprise: access to a larger amount of local (cache) memory.

4.3 Distribution Functions and Locality

In most of the experiments, states are distributed evenly among nodes using the hash function from UPPAAL. However, for small models we observed that some nodes explore twice as many states as others because some location vectors have more reachable symbolic states than others, which means that some nodes have more states allocated than others. Counting the number of different location vectors on the different nodes, the distribution again looks uniform. This effect does not show up in larger models.

We ran experiments for different distribution functions: a function hashing on the discrete part of a state (D0), a function hashing on the complete state (D1), a function hashing on the integer variables (D2), and a function hashing on every second location (D3). We also ran experiments for different settings of the state space reduction technique described in Section 3.3, where only states that are actually stored in the passed list are mapped to different nodes: storing all states (S0), storing non-committed or loop entry points (S1), and storing only loop entry points (S2). Table 5 shows for the buscoupler and the power-down models the percentage of states explored on the same node they were generated on. These experiments were run on the Sun Enterprise with 8 CPUs, but similar results were obtained using the Beowulf cluster.

Table 5. Percent of locally explored states for different distribution and storage policies for the buscoupler model (left) and the power-down protocol (right) when verified on a Sun Enterprise using 8 nodes.

Bus	D0	D1	D2	D3
S0	14%	n/a	52%	42%
S1	36%	n/a	60%	58%
S2	55%	n/a	62%	62%

PD	D0	D1	D2	D3
S0	4%	n/a	76%	22%
S1	34%	n/a	76%	48%
S2	60%	n/a	78%	86%

For the buscoupler with D0 and S0 we almost obtain the expected uniform distribution ($100\%/8 = 12.5\%$). This was not the case for the power-down model although the total load on the nodes was uniform. None of the D1 experiments terminated within a reasonable time frame. This was expected since much fewer inclusion checks can succeed with this distribution function and hence a much higher number of symbolic states will be generated. Both S1/S2 and D2/D3 improve locality. What cannot be seen is that both S1 and S2 increase the number of states generated (for the buscoupler to such an extend that S2 is actually slower than S0). D2 is surprisingly uniform while increasing locality, but the load distribution of D3 was observed to be highly non-uniform, resulting in poor performance. For the buscoupler D2 and S1 turned out to be the fastest combination. For the power-down model D2 and S2 turned out to be the fastest combination.

4.4 Generating Shortest Traces

For the buscoupler system we tried the version finding the shortest trace on four different properties (finding a particular state not generating the complete state space) on the Sun Enterprise. The speedups are displayed in Fig. 6. As for the DACAPO system the speedup for properties one and two suffer from too few states being explored. The speedup for properties three and four are much better but here more states are searched to find the state satisfying the property. So we can conclude that also the version finding shortest trace scales quite well, as long as sufficiently many states need to be generated.

5 Conclusions

This paper demonstrates the feasibility of distributed model checking of timed automata. A side effect of the distribution was an altered search order, which in turn increased the number of symbolic states generated when exploring the reachable state space. We have proposed explicit ordering of the states in the waiting list as an effective heuristic to improve the scalability of the approach. In addition we propose an algorithm for finding shortest traces that performs well in a distributed model checker.

In several cases we obtained super linear speedups. We have suggested some explanations, but more work is needed to clarify the observed phenomena. Importantly, some of our results suggests possible improvements to the sequential state space exploration algorithm for timed automata.

References

1. S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. In *Discrete Event Systems: Models and Applications. IIASA Conference*, pages 40–56, 1987.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
3. S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for GSPN models. In G. De Michelis and M. Diaz, editors, *Proceedings 16th Int. Conf. on Application and Theory of Petri Nets,* Turin, Italy, volume 935 of *Lecture Notes in Computer Science*, pages 181–200. Springer-Verlag, June 1995.
4. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Cambridge, Massachusetts, 1999.
5. D. L. Dill. The murφ verification system. In *Conference on Computer-Aided Verification*, LNCS, pages 390–393. Springer-Verlag, July 1996.

6. K. Havelund, K. Larsen, and A. Skou. Formal verification of a power controller using the real-time model checker UPPAAL. In Joost-Pieter Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS'99*, volume 1601 of *Lecture Notes in Computer Science*, pages 277–298. Springer-Verlag, 1999.

7. K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 2–13, December 1997. San Francisco, California, USA.

8. B.R. Haverkort, A. Bell, and H.C. Bohnenkamp. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models (PNPM'99),* Zaragoza, Spain, pages 12–21. IEEE Computer Society Press, 1999.

9. T. Hune, K. G. Larsen, and P. Pettersson. Guided synthesis of control programs using UPPAAL. In *Proc. of the International Workshop on Distributed Systems, Verification and Validation*, April 2000. Taipei, Taiwan.

10. K.J. Kristoffersen, F. Laroussinie, K.G. Larsen, P. Pettersson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97: Theory and Practice of Software Development,* Lille, France, volume 1214 of *Lecture Notes in Computer Science*, pages 565–579. Springer-Verlag, April 1997.

11. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

12. 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.

13. H. Lönn and P. 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.

14. M. Snir, S.W. Otto, S. Huss-Lederman, D. Walker, and J.J. Dongarra. *MPI:The Complete Reference.* MIT Press, Cambridge, Massachusetts, 1996.

15. U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference*, volume 1254 of *LNCS*, pages 256–67. Springer-Verlag, June 1997. Haifa, Isreal, June 22-25.

A Results

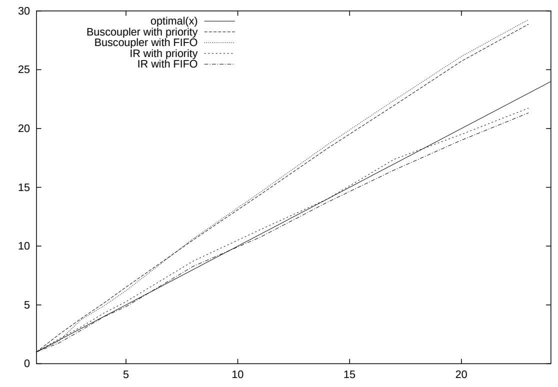

Fig. 4. Speedup for CP and buscoupler on Sun Enterprise

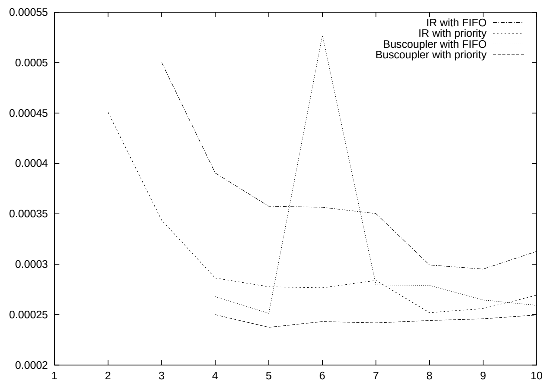

Fig. 5. Work for CP and buscoupler on Beowulf

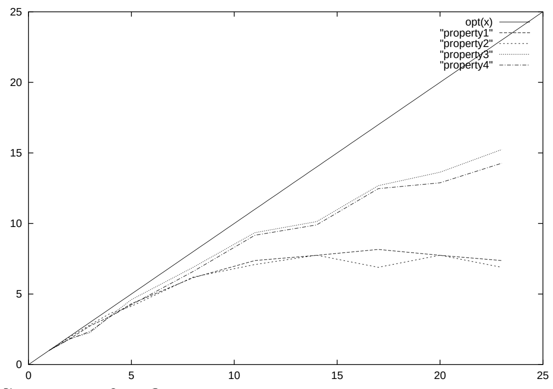

Fig. 6. Speedup for finding shortest trace in Buscoupler model.