On Memory-Block Traversal Problems in Model-Checking Timed Systems

Fredrik Larsson[1] *Paul Pettersson*[2] *Wang Yi*[1]

[1] Department of Computer Systems, Uppsala University, Sweden.
E-mail: {fredrikl,yi}@docs.uu.se.
[2] BRICS⋆⋆⋆, Department of Computer Science, Aalborg University, Denmark.
E-mail: paupet@cs.auc.dk.

Abstract. A major problem in model-checking timed systems is the huge memory requirement. In this paper, we study the memory-block traversal problems of using standard operating systems in exploring the state-space of timed automata. We report a case study which demonstrates that deallocating memory blocks (i.e. memory-block traversal) using standard memory management routines is extremely time-consuming. The phenomenon is demonstrated in a number of experiments by installing the UPPAAL tool on Windows95, SunOS 5 and Linux. It seems that the problem should be solved by implementing a memory manager for the model-checker, which is a troublesome task as it is involved in the underlining hardware and operating system. We present an alternative technique that allows the model-checker to control the memory-block traversal strategies of the operating systems without implementing an independent memory manager. The technique is implemented in the UPPAAL model-checker. Our experiments demonstrate that it results in significant improvement on the performance of UPPAAL. For example, it reduces the memory deallocation time in checking a start-up synchronisation protocol on Linux from 7 days to about 1 hour. We show that the technique can also be applied in speeding up re-traversals of explored state-space.

1 Introduction

During the past few years, a number of verification tools have been developed for real-time systems in the framework of timed automata (e.g. KRONOS and UPPAAL [HH95,DOTY95,LPY97,BLL$^+$98]). One of the major problems in applying these tools to industrial-size systems is the huge memory-usage (e.g. [BGK$^+$96]) for the exploration of the state-space of a network (or product) of timed automata. The main reason is that the model-checkers must store a large number of symbolic states each of which contains information not only

⋆⋆⋆ **B**asic **R**esearch **I**n **C**omputer **S**cience, Centre of the Danish National Research Foundation.

on the control structure of the automata but also the clock values specified by clock constraints. To reduce memory usage, the model-checker must throw away parts of the state-space explored (resulting in memory deallocation), that are not needed for further analysis or re-traverse parts of the state-space explored and stored in (virtual) memory blocks to check a new property. In both cases, the underling operating system must traverse the memory blocks storing the state-space explored.

Unfortunately, using the standard memory management service for memory-block traversals e.g. memory deallocation is surprisingly time-consuming in particular when swapping is involved during state-space exploration. A problem we discovered in a very recent case study when Uppaal was applied to check two correctness properties of the start-up algorithm for a time division multiple access protocol [LP97]. The first property was verified using 5 hours of CPU time and 335MB of memory[1] but the memory deallocation process, after verifying the first property, did not terminate until 7 days!

The phenomenon described above is caused by the so-called *thrashing*, which occurs occasionally in common-purpose operating systems, but much more often in the context of state-space exploration due to the large memory consumption. Unfortunately, this is a phenomenon not only occurring on Linux, but most of the existing operating systems. The fact has been demonstrated by our experiments on Uppaal installed on Linux, Windows 95 and SunOS 5. Furthermore, we notice that as Uppaal is based on the so-called symbolic reachability analysis which is the basis for several other model-checkers e.g. Kronos and HyTech, this should be a common problem for verification tools in the domain of real-time systems.

More intuitively, the problem can be described as follows: When throwing away parts of the state-space, the states are deallocated one by one. Note that the size of a state could be a number of bytes. To deallocate the amount of memory for a particular state, the memory page containing that state must be in the main memory. When swapping is involved, this means that the particular page must be loaded from disc. If the next state we want to throw away is in another page, and memory is almost full, the newly loaded page must be swapped out, even if it needs to be swapped in later when another state shall be removed. If the deallocation order is independent of how the allocated states are mapped to memory, unnecessary swapping will occur. Therefore, it is crucial to store information on the allocation order of memory blocks, but this will introduce extra overhead for the model-checker. It is not obvious how much information that should be collected during the verification process and used later for deallocating. The more information collected, the more overhead in the verification but the better the deallocation performance obtained. We need to find the best trade-off.

[1] The experiment was performed on a 200 MHz Pentium Pro equipped with 256MB of primary memory running Red Hat Linux 5.

As our first experiment, we have simulated the allocation order of memory blocks in Uppaal and experimented with three different deallocation orders. The first simply traverses the allocated structure without taking into account how blocks were allocated. This was the one used in Uppaal when the start-up protocol was verified. The second strategy deallocates memory blocks in the same order as they were allocated. The third one deallocates them in the reverse allocation order. According to our experiments, the last strategy is clearly the best choice, which has been implemented in Uppaal. It results in significant performance improvements on Uppaal. For example, it reduces the memory deallocation time on Linux from 7 days to about 1 hour for the start-up protocol. The technique is also implemented to speed up re-traversing of the explored state-space to check new properties when the model-checker is used in an interactive manner. Our experiments demonstrate similar performance improvement.

The rest of the paper is organised as follows: In Section 2, we briefly introduce the notion of timed automata and symbolic reachability analysis for networks of timed automata. In Section 3, we describe and study the memory deallocation problem in more details. Several experiments are presented to illustrate that it is a common phenomenon for all the common-purpose operation systems. In Section 4, we present a solution to the problem and experimental results showing that our solution does result in a significant performance improvement for the Uppaal tool. Section 5 concludes the paper by summarising our contributions and future work.

2 Preliminaries

2.1 Timed Automata

Timed automata was first introduced in [AD90] and has since then established itself as a standard model for real–time systems. For the reader not familiar with the notion of timed automata we give a short informal description.

A timed automaton is a standard finite–state automaton extended with a finite collection C of real–valued clocks ranged over by x, y etc. A clock constraint is a conjunction of atomic constraints of the form: $x \sim n$ or $x - y \sim n$ for $x, y \in C$, $\sim \in \{\leq, <, \geq\}$ and n being a natural number. We shall use $\mathcal{B}(C)$ ranged over by g (and later by D) to stand for the set of clock constraints.

Definition 1. *A timed automaton A over clocks C is a tuple $\langle N, l_0, E, I\rangle$ where N is a finite set of nodes (control-nodes), l_0 is the initial node, $E \subseteq N \times \mathcal{B}(C) \times 2^C \times N$ corresponds to the set of edges, and finally, $I : N \rightarrow \mathcal{B}(C)$ assigns invariants to nodes. In the case, $\langle l, g, r, l'\rangle \in E$, we write $l \xrightarrow{g,r} l'$.* □

Formally, we represent the values of clocks as functions (called clock assignments) from C to the non–negative reals $\mathbf{R}$. We denote by $\mathbf{R}^C$ the set of clock assignments for C. A semantical *state* of an automaton A is now a pair (l, u), where l is a node of A and u is a clock assignment for C, and the semantics of A is given by a transition system with the following two types of transitions (corresponding to delay–transitions and edge–transitions):

- $(l, u) \rightarrow (l, u + d)$ if $I(u)$ and $I(u + d)$
- $(l, u) \rightarrow (l', u')$ if there exist g and r such that $l \xrightarrow{g,r} l'$, $u \in g$ and $u' = [r \rightarrow 0]u$

where for $d \in \mathbf{R}$, $u + d$ denotes the time assignment which maps each clock x in C to the value $u(x) + d$, and for $r \subseteq C$, $[r \mapsto 0]u$ denotes the assignment for C which maps each clock in r to the value 0 and agrees with u over $C \backslash r$. By $u \in g$ we denote that the clock assignment u satisfies the constraint g (in the obvious manner).

Clearly, the semantics of a timed automaton yields an infinite transition system, and is thus not an appropriate basis for decision algorithms. However, efficient algorithms may be obtained using a *symbolic* semantics based on *symbolic states* of the form (l, D), where $D \in \mathcal{B}(C)$ [HNSY94,YPD94]. The symbolic counterpart to the standard semantics is given by the following two (fairly obvious) types of symbolic transitions:

- $(l, D) \rightsquigarrow \left(l, (D \wedge I(l))^{\uparrow} \wedge I(l)\right)$
- $(l, D) \rightsquigarrow \left(l', r(g \wedge D)\right)$ if $l \xrightarrow{g,r} l'$

where $D^{\uparrow} = \{u + d \mid u \in D \wedge d \in \mathbf{R}\}$ and $r(D) = \{[r \rightarrow 0]u \mid u \in D\}$. It may be shown that $\mathcal{B}(C)$ (the set of constraint systems) is closed under these two operations ensuring that the semantics is well–defined. Moreover, the symbolic semantics corresponds closely to the standard semantics in the sense that, whenever $u \in D$ and $(l, D) \rightsquigarrow (l', D')$ then $(l, u) \rightarrow (l', u')$ for some $u' \in D'$.

It should be noticed that the symbolic semantics above is by no means finite because clock values are unbounded. However, the following reachability problem can be solved in terms of a finite symbolic semantics based on the so-called k-normalisation on clock constraints [Pet99,Rok93].

2.2 Reachability Analysis

Given an automaton with initial symbolic state (l_0, D_0), we say that a symbolic state (l, D) is reachable if $(l_0, D_0) \rightsquigarrow^* (l_n, D_n)$ and $D_n \wedge D \neq \emptyset$. The problem can be solved by a standard graph reachability algorithm; but termination may not be guaranteed because the number of clock constraints generated may be infinite. The standard solution to this problem is by introducing a k-normalised version

```
Passed:= $\{\}$
Waiting:= $\{(l_0, D_0)\}$
repeat
        begin
        get $(l, D)$ from Waiting
        if $(l, D) \models \varphi$ then return "YES"
        else if $D \not\subseteq D'$ for all $(l, D') \in$ Passed then
                begin
                add $(l, D)$ to Passed
                Succ:=$\{(l_s, D_s) : (l, D) \leadsto_k (l_s, D_s) \land D_s \neq \emptyset\}$
                for all $(l_{s'}, D_{s'})$ in Succ do
                        put $(l_{s'}, D_{s'})$ to Waiting
                end
        end
until Waiting=$\{\}$
return "NO"
```

Fig. 1. An Algorithm for Symbolic Reachability Analysis.

of the infinite symbolic semantics. The idea is to utilise the maximal constants appearing in the clock constraints of the automaton under analysis and D of the final symbolic state to develop a finite symbolic transition system. For details we refer the reader to [Pet99]. The main fact about the k-normalisation is as follows:

Assume that k is the maximal constant appearing in an automaton A with initial state (l_0, D_0). Then (l, D) is reachable from (l_0, D_0) iff there exists a sequence of k-normalised transitions: $(l_0, D'_0) \leadsto_k (L_1, D'_1)...(l_{n-1}, D'_{n-1}) \leadsto_k (l_n, D'_n)$ such that $D \land D'_n \neq \emptyset$ where D'_i is the so-called normalised constraints with all constants being less than k.

Figure 1 shows the pseudo-code of a reachability algorithm to check if the automaton satisfies a given reachability formula e.g. a final symbolic state of the form (l, D)[2]. It is basically a standard graph-searching algorithm. The algorithm use two important data structures: Waiting and Passed. Waiting contains the state-space awaiting to be explored. If this data structure is a queue the search order is breath-first; if it is organised as a stack, the search becomes depth-first. At start, the initial state is placed in the Waiting structure. Passed contains the parts of the state-space explored so far. It is implemented as a hash table so that it can be searched and updated efficiently. Initially, it is empty. Due to the size of state-space, these structures may consume a huge amount of main memory.

[2] We define that $(l', D') \models (l, D)$ if $l' = l$ and $D' \land D \neq \emptyset$.

Memory	Disc	Operation
{s2,s4}	{s1,s3}	deallocReq(s1)
		SWAP
{s1,s3}	{s2,s4}	dealloc(s1)
{-,s3}	{s2,s4}	
{-,s3}	{s2,s4}	deallocReq(s2)
		SWAP
{s2,s4}	{-,s3}	dealloc(s2)
{-,s4}	{-,s3}	
{-,s4}	{-,s3}	deallocReq(s3)
		SWAP
{-,s3}	{-,s4}	dealloc(s3)
{-,-}	{-,s4}	
{-,-}	{-,s4}	deallocReq(s4)
		SWAP
{-,s4}	{-,-}	dealloc(s4)
{-,-}	{-,-}	

Table 1. Memory Deallocation Example

3 The Problem and Solutions

The algorithm (or its equivalent) presented in the previous section has been implemented in several verification tools e.g. Uppaal for timed systems. Such tools are either used in an *interactive* manner, when the users interactively enters reachability properties given as symbolic states, or in a *non-interactive* manner, where the sequence of properties are known a priori.

When used interactively, the tool may in the worst case construct a huge date structure Passed (storing the explored state-space) for each symbolic state when it contains a different maximal clock constant. Therefore, before each check, the model-checker must traverse and deallocate states (i.e. memory blocks) used for previous checks. Note that this is not the only reason why memory deallocation is required during the verification process. For example, for each separate reachability check, parts of the explored state-space may be thrown away when they are not needed for further analysis, which also requires memory deallocation. In the special case where the whole state-space must be deallocated, and this is known before the actual verification starts, it is possible to avoid traversing memory blocks by creating a separate process that does the verification. It is then possible to deallocate all states just by "killing" the dedicated process and have the operating system reclaiming all pages at once. However, this is not applicable when only parts of the state-space are deallocated.

When the tool is used non-interactively, the maximal constant of the whole sequence may be determined before the first property is checked, as all symbolic states to be checked are known. Thus, the Passed structure does not have to be deallocated between two consecutive checks. In fact, the state-space generated in the previous checks is often reused to avoid unnecessary re-computation. A new check then amounts to determine if the symbolic state is already in the previously generated state-space and, if necessary, continue to generate new symbolic states. Note that, independent of how the tool is used, each check requires the previously generated state-space to be accessed, either during memory deallocation or when reusing the state-space[3]. Both cases result in memory-block traversals.

Surprisingly, the time spent on traversing states in PASSED consumes a very large part of the execution time. The reason is that standard operating-system services for memory management requires that the page containing the state to access resides in main memory. This is ensured by swapping out other memory pages to disc; pages that later may have to be swapped in again because they contain other states to access. It is clear that when swapping is involved, it is important how the memory is accessed, i.e. in what order the states are accessed. Ideally, we would like to localise memory accesses for states as much as possible.

To improve the presentation, the remainder of this section focuses on techniques for more efficient memory deallocation when swapping is involved. However, the presented techniques apply also to the case when a large portion of the memory is accessed, as when the state-space is reused when several properties are checked. We shall study the case of reuse further in the next section.

3.1 An Example

To illustrate the problem we study an example where memory is deallocated. We assume two memory pages, each containing two states. Initially one page is in main memory and one is in a part of the virtual memory currently on disc. Tables 1 and 2 show the page layout in main memory and on disc together with the operations an operating system may perform when the application requests deallocation of the states. The strategies illustrated is deallocation of the states when they are traversed in an order independent of memory layout and reverse allocation order respectively.

In Table 1 the allocation order is s1, s3, s2, s4 and the deallocation order is s1, s2, s3, s4. SWAP is a very expensive operation and the deallocation strategy in Table 1 requires four such operations in order to deallocate all states. In Table 2 the allocation order is the same as in Table 1 but the deallocation order is different; s4, s2, s3, s1 i.e. reverse allocation order. By using this deallocation strategy the number of SWAP operations can be reduced to one. The dealloc() can be performed immediately after the request in most cases.

[3] In the latter case the search may terminate before the whole state-space has been accessed.

Memory	Disc	Operation
{s2,s4}	{s1,s3}	deallocReq(s4)
{s2,s4}	{s1,s3}	dealloc(s4)
{s2,-}	{s1,s3}	
{s2,-}	{s1,s3}	deallocReq(s2)
{s2,-}	{s1,s3}	dealloc(s2)
{-,-}	{s1,s3}	
{-,-}	{s1,s3}	deallocReq(s3)
		SWAP
{s1,s3}	{-,-}	dealloc(s3)
{s1,-}	{-,-}	
{s1,-}	{-,-}	deallocReq(s1)
{s1,-}	{-,-}	dealloc(s1)
{-,-}	{-,-}	

Table 2. Memory Deallocation Example

Blocks	Linux	Solaris	Windows
32 768	169	845	469
65 536	387	1 795	1 038
131 072	1 029	4 272	2 487
262 144	2 709	9 779	6 250
524 288	7 691	25 193	12 288
1 048 576	27 790	22 082	43 227

Table 3. Deallocation time (in seconds) for hashtable order.

3.2 Deallocation Strategies

A common way to represent state-spaces is to use data structures based on hash tables for efficient analysis. A convenient way to deallocate such data structures is to go through the table in consecutive hash value order and deallocate the symbolic states one by one. This is not by far the most efficient strategy even if it is convenient to implement. Table 3 shows deallocation times when blocks are deallocated in a hash-value order, an order totally ignoring how blocks are layed-out on pages and whether requested pages are on disc or in main memory. To further emphasise the fact that deallocation order affects the amount of swapping see example 3.1. The example in Table 1 and Table 2 illustrates the operations involved when deallocating memory according to two different strategies.

A much better strategy would be to first deallocate blocks on pages already in main memory and when a page is swapped in from disc deallocate all blocks on that page before swapping it out. This strategy would suit most common memory-management strategies used in operating systems. However this type of low-level information is generally not available to an application program like

Blocks	Linux	Solaris	Windows
32 768	122	124	179
65 536	124	125	193
131 072	127	135	200
262 144	128	151	240
524 288	145	198	198
1 048 576	176	242	300

Table 4. Deallocation time (in seconds) for allocation order.

the Uppaal model-checker. Most standard programming languages and portable operating system libraries only allow the application programs to request deallocation of a previously allocated block. It is up to the application program to perform the requests in a suitable order. Information that an application program may maintain is in what order memory blocks have been allocated.

It is also possible to collect information on how often a memory block is accessed. While this may give some hints on whether a block resides on a page in main memory or on a page on disc, it is not enough to decide what blocks reside on the same page thus leading to the same bad performance with heavy swapping.

To test if a successful deallocation strategy could be based only on information about allocation order, we had an experiment in which 32MB of memory were allocated in a number of equally sized blocks on three machines with 32MB of physical memory running the operating systems Linux, SunOS 5 and Windows 95. The blocks were placed randomly in a hash table with place for each allocated block. The blocks were then deallocated according to three different strategies: We call the first one hash table order. It is used to illustrate a commonly used order, easy to implement but independent of memory layout. The second is deallocation in the same order as allocation. The third order is deallocation in reverse allocation order.

Table 3, 4 and 5 show the deallocation times for the three chosen strategies implemented on the three operating systems: Linux, SunOS 5 and Windows 95. The experimental results clearly indicate that memory deallocation time really matters when swapping is involved. Both strategies that utilise the information about allocation order are superior to the first one i.e. the table order [4]. Note that the strategy using reverse allocation order demonstrates the best performance on all three operating systems. The reason may be that newly allocated blocks are used more recently and hence are more likely to reside in main memory.

[4] Note that this may be the most common strategy adopted by the existing verification tools e.g. Uppaal.

Blocks	Linux	Solaris	Windows
32 768	26	74	33
65 536	18	110	13
131 072	21	119	19
262 144	31	130	38
524 288	44	153	44
1 048 576	77	204	99

Table 5. Deallocation time (in seconds) for reverse allocation order.

4 Implementation and Performance

The experimental results presented in the previous section indicate that the deallocation strategy currently implemented in UPPAAL, which corresponds to hash table order, should be modified to optimise the time-performance. Note that the problem we want to solve here is how to find a suitable traversal strategy that for example let us control memory deallocation efficiently, by localising memory accesses as much as possible, without writing our own memory manager. Thus, the question is how to keep track of the allocation order of memory blocks without getting involved in low-level operations. Certainly, it is not a good idea to keep track of the allocation order of all memory blocks, as this might be as hard as writing a completely new memory manager.

Our solution is based on the observation that memory deallocation is mainly performed in two different situations: between consecutive reachability checks performed on the same system description, and just before the program terminates. In these situations deallocating memory corresponds to throwing away parts of the symbolic state-space that are not needed for the next reachability check. Thus, to utilise the presented deallocation strategies we need to keep track of the allocation order of the symbolic states. This is realised by extending every symbolic state with two pointers that are used to store the symbolic states in a doubly-linked list, sorted in allocation order. The list structure is easy to maintain and allows the symbolic state-space to be traversed in allocation order and reverse allocation order, as required by the presented memory deallocation strategies, in linear time. It also enables deallocation of symbolic states close to each other in memory to occur close in time while a page is in main memory, i.e. to keep the deallocation as local as possible.

In fact, the solution is an approximation to the exact allocation order for the symbolic states. This is because some operations performed by the reachability algorithm change parts of a symbolic state and it cannot be guaranteed that all data belonging to a given symbolic state is allocated consecutively. Further, all data for a state may not fit together on a single page. These facts make the assumption that states allocated consecutively will have all its data collected on the same page weaker.

4.1 Performance of Deallocation Strategies

The presented deallocation strategies have been implemented and integrated in a new version of Uppaal. In this section we present the results of an experiment where the new Uppaal version was installed on Linux, Windows 95 and SunOS 5, and applied to verify three examples from the literature:

Bang and Olufsen Audio/Video Protocol (B&O) This example is a protocol developed by Bang and Olufsen that is highly dependent on real-time [HSLL97]. It is used in their audio and video equipments to exchange control information between components communicating on a single bus. In the experiment we have verified the correctness criteria of the protocol. For details we refer to section 5.1 of [HSLL97].
The verification was performed using Uppaal installed on a Pentium 75MHz PC machine equipped with 8MB of physical memory running Linux.

Dacapo start-up Algorithm (Dacapo) The Dacapo protocol is a time division multiple access (TDMA) based bus protocol [LP97]. It is intended for physically small safety-critical distributed real-time systems limited to tens of meters and less than 40 nodes, e.g. operating in modern vehicles. In the experiment we verify that the start-up algorithm of the protocol is correct in the sense that the protocol becomes operational within a certain time bound. To vary the amount of needed memory in the verifications it is possible to adjust the number of communicating nodes of the protocol.
Four versions of the protocol were verified on four machines: the Pentium 75MHz described above, a Pentium MMX 150MHz with 32MB of physical memory running both Linux and Windows 95, a Pentium Pro 200MHz equipped with 256MB of memory running Linux, and a Sun SPARC Station 4 with 32MB of memory running SunOS 5.

Fischer's Mutual Exclusion Protocol (Fischer) This is the well-known Fischer's protocol previously studied in many experiments, e.g. [AL92,KLL$^+$97]. It is to guarantee mutual exclusion among several processes competing for a critical section. In the experiment we verify that the protocol satisfies the mutual exclusion property, i.e. that there is never more than one process in the critical section. Two versions of the protocol were verified using Uppaal installed on the Pentium 75MHz PC.

Table 6 presents the memory usage together with the verification time (check) and the time needed to deallocate the required memory (dealloc) in seconds. Each example is verified with Uppaal versions deallocating memory using the original strategy, i.e. the hash table order, and the two new strategies, namely allocation order and reverse allocation order.

As shown in Table 6, memory deallocation in reverse allocation order outperforms both hash table order and allocation order in the tested examples. In Uppaal, the reverse allocation order saves 82% to 99% of the deallocation time compared with the originally used hash table order. It can also be observed that

Memory Usage		Machine		Hash table		Allocation		Reverse	
Example	MB	OS	MB	check	dealloc	check	dealloc	check	dealloc
B&O	13	Linux	8	1 400	31 978	1486	1127	1497	1067
Fischer	8	Linux	8	126	1 118	132	207	133	197
	9	Linux	8	135	1 995	138	290	143	245
Dacapo	16	Linux	8	4 654	37 363	5 031	8 095	5 046	1 999
	38	Linux	32	621	6 013	689	812	690	597
	38	Solaris	32	3 406	3 780	3 740	304	3 704	279
	38	Windows	32	754	11 850	797	1035	823	995
	56	Linux	32	4 413	164 328	4 743	2 781	4 819	2 647
	56	Solaris	32	8 764	5 969	10 271	384	10 333	375
	336	Linux	256	21 189	602 354	24 741	6 754	23 390	5 307

Table 6. Deallocation times (in seconds).

the overhead during verification associated with keeping track of the allocation order is relatively small, which varies between 6% and 19% in the experiment. Moreover, the space overhead, which is not shown in the table, is insignificant.

4.2 Performance of State-space Traversals

Memory Usage		Hash table		Allocation		Reverse	
Example	MB	check	re-use	check	re-use	check	re-use
Dacapo	42	652	1 169	772	106	781	107
Fischer	43	532	498	540	94	546	99

Table 7. Verification times (in seconds).

In section 3 it was mentioned that properties were often verified interactively, and that changes in the maximal constants may require deallocation of the whole state-space before verification of a new property. If the properties are known a priori the maximal constant for all properties can be determined thus eliminating the need to destroy the Passed and Waiting structures for that reason. Another advantage with such an approach is that we can search through the state-space generated so far and check if their already exist states satisfying our reachability property and only generate successors of states on Waiting if no states exist in Passed.

This approach would obviously increase the memory consumption and increase the possibility of swapping during traversal of the generated state-space since Passedand Waitingwill not be deallocated. In fact the same reasoning in finding a better deallocation order of states may be used here. Assume that we want to verify n reachability properties $p_1...p_n$. If we traverse the state-space in a manner that keeps accesses to states as local as possible we might reduce swapping and the verification time for properties p_2 to p_n.

Table 7 compares the verification times of traversing the state-space in the three different orders described earlier. All of them were implemented in UPPAAL and tested on a 150 MHz Pentium running Linux. To guarantee that the same number of symbolic states were search through by all the different strategies we only verify properties not satisfied by the system. In this way the whole generated state-space is traversed in all the three cases. As shown in Table 7, we obtain reductions in time-usage in traversing the state-space for up to 80%.

In order to perform experiments involving swapping we have to use examples that consume more physical memory than what is available on the given hardware architecture. Also, we are forced to use existing configurations of processors, amount of physical memory and the possibilities to install the different operating systems. It turned out that most of our case-studies did not meet the imposed requirements. They were either too small or too large. This explains the rich variation of used hardware architectures and why the same examples were verified multiple times. We still think that the results are significant since the behaviour of all three heuristics was consistent for all examples.

5 Conclusion and Future Work

We have studied memory-block traversal behaviour of verification tools for real time systems. We discovered that deallocating memory blocks during state-space exploration using standard memory management routines in the existing operating systems is extremely time-consuming when swapping is involved. This common phenomenon is demonstrated by experiments on three common-purpose operating systems, Windows 95, SunOS 5 and Linux. It seems that the problem should be solved by implementing a memory manager for the model-checker. However this may be a troublesome task as it is involved in internal details of the underlining operating system.

As the second contribution of this paper, we present a technique that allows the model-checker to control how the operating system deallocates memory blocks without implementing an independent memory manager. The technique is implemented in the UPPAAL model-checker. Our experiments show that it results in significant improvements of the performance of UPPAAL. For example, it reduces the memory deallocation time on Linux from 7 days to about 1 hour for a start-up synchronisation protocol published in the literature. The proposed solution introduces very little overhead during the reachability analysis, and it guarantees that examples not involving swapping still perform well. The technique has been applied to speed up re-traversals (i.e. re-use) of the explored state-space in reachability analysis for timed automata when checking a sequence of properties with the same maximal clock constant.

We should point out that even though most of the experiments presented here focus on memory-block deallocation in model-checking timed systems, our results are applicable to any problem involving traversals of large amounts of memory in model-checking not only for timed systems, but concurrent systems in general. For other work in the context of memory management for automated verification, see [Boe93,Wil92,SD98]. As future work, we plan to develop a special-purpose memory manager for verification tools, that keeps total control over the allocation order and memory layout.

References

[AD90] Rajeev Alur and David Dill. Automata for Modelling Real-Time Systems. In *Proc. of Int. Colloquium on Algorithms, Languages and Programming*, number 443 in Lecture Notes in Computer Science, pages 322–335, July 1990.

[AL92] Martin Abadi and Leslie Lamport. An Old-Fashioned Recipe for Real Time. In *Proc. of REX Workshop "Real-Time: Theory in Practice"*, number 600 in Lecture Notes in Computer Science, 1992.

[BGK$^+$96] 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. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of the 8th Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.

[BLL$^+$98] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, Wang Yi, and Carsten Weise. New Generation of UPPAAL. In *Int. Workshop on Software Tools for Technology Transfer*, June 1998.

[Boe93] Hans-J. Boehm. Space Efficient Conservative Garbage Collection. In *Proc. of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 197–206, 1993.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 208–219. Springer–Verlag, October 1995.

[HH95] Thomas A. Henzinger and Pei-Hsin Ho. HyTech: The Cornell HYbrid TECHnology Tool. In *Proc. of TACAS, Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1995. BRICS report series NS–95–2.

[HNSY94] Thomas. A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994.

[HSLL97] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In *Proc. of the 18th IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1997.

[KLL$^+$97] Kåre J. Kristoffersen, Francois Laroussinie, Kim G. Larsen, Paul Pettersson, and Wang Yi. A Compositional Proof of a Real-Time Mutual Exclusion

Protocol. In *Proc. of the 7th Int. Joint Conf. on the Theory and Practice of Software Development*, April 1997.

[LP97] 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.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[Pet99] Paul Pettersson. *Modelling and Analysis of Real-Time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Department of Computer Systems, Uppsala University, February 1999.

[Rok93] Tomas Gerhard Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford University, 1993.

[SD98] Ulrich Stern and David L. Dill. Using Magnetic Disk instead of Main Memory in the Murphi Verifier. In *Proc. of the 10th Int. Conf. on Computer Aided Verification*, Lecture Notes in Computer Science. Springer–Verlag, June 1998.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proc. of the International Workshop on Memory Management*, number 637 in LNCS. Springer–Verlag, 1992.

[YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the 7th Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.