Model-Checking with Insufficient Memory Resources

Birgitta Lindström[1] and Paul Pettersson[2]

[1] University of Skövde, Box 408, 541 28 Skövde, Sweden
birgitta.lindstrom@his.se
[2] Uppsala University, Box 337, 751 05 Uppsala, Sweden
paul.pettersson@it.uu.se

Technical Report HS- IKI -TR-06-005
School of Humanities and Informatics
University of Skövde

Abstract. Resource limitations is a major problem in model checking. Space and time requirements of model-checking algorithms grow exponentially with respect to the number of variables and parallel automata of the analyzed model. We present a method that is the result of experiences from a case study. It has enabled us to analyze models with much bigger state-spaces than what was possible without our method.
The basic idea is to build partitions of the state-space of an analyzed system by iterative invocations of a model-checker. In each iteration the partitions are extended to represent a larger part of the state space, and if needed the partitions are further partitioned. Thereby the analysis problem is divided into a set of subproblems that can be analyzed independently of each other.
We present how the method, implemented as a meta algorithm on-top of the UPPAAL tool, has been applied in the case study.

1 Introduction

In the last decades, model checking (e.g., [1] [2]) has established itself as a powerful technique for automatic formal verification of transition systems. Its success has lead to development of several verification tools, including SMV [3], [4] and SPIN [5] for finite state systems, and e.g., UPPAAL [6] and KRONOS [7] for real-time systems modeled as timed automata. These tools have been applied to prove the correctness of several non-trivial industrial systems [8].

A well-known problem when applying model-checking in general is the *state-space explosion problem* [9], i.e., the exponential size of state space w.r.t size of the input model. As a consequence, available memory (or time) can be insufficient for model-checking of complex system models. In such cases, attempts to perform exhaustive model checking, such as verifying a global invariant, are bound to fail. Therefore, a significant amount of work focus on reducing memory usage of model-checking algorithms (e.g., [10], [11], and [12]).

Memory and time remains a bottleneck in model checking. We present a method for mitigating the problem of space limitations in model-checking algorithms based on state-space exploration. The state space is partitioned and a verification algorithm is iteratively invoked to gradually further analyze and partition the existing set of partitions. The analysis problem is thereby divided into subproblems, which can be analyzed independently.

In the presented partitioning method, each partition is represented as the set of states that can be reached, given that a sequence of so-called *partitioning points* are traversed in a given order. Each partition can be generated and extended by guiding the original model, so that it stays within the partition, and further extends and divides the partition, if possible. We show how this can be achieved using an existing model checker by dynamic manipulation of the analyzed model.

The method is applied to analyze a large case study of a real-time application, using the UPPAAL tool [6]. We share experiences from this case study, and report that our method could generate all traces of interest, whereas ordinary model-checking could cover only 26% of them, using the same memory resources.

The rest of this paper is organized as follows: In the remainder of this section we discuss related work. In Section 2 we present preliminary results and give a motivating example of our method. In Section 3 and 4 we present our method in detail, and present how it was applied in a case study. In Section 5 and 6 we discuss how to use our method, conclude, and outline some future work.

Related Work: In order to speed up verification, algorithms for distributed exploration of state-space has been developed [13]. This approach increases speed but the complexity with respect to memory remains and the method requires a lot of communication between the nodes. Our method is based on partitioning the state-space into pieces that we can handle with available memory resources and then explore them one at a time. Our method can easily be subject for distribution and, in such case, the number of messages required is bounded to the number of partitions.

Bounded model-checking uses a SAT solver for finding logical errors or proving their absence in finite-state transition systems [14]. The basic idea is to verify executions of length k and, if no bug is found, iteratively increase k until: (*i*) a bug is found, (*ii*) the problem is intractable, or (*iii*) a defined threshold is reached. Our method is based on the principle of divide an conquer along traces of a model. A technique that enhances our chances to reach deeper into the search space compared with bounded model-checking. We dynamically divide the state-space into partitions. Exploration of the state-space is then performed independently on these partitions.

Partial-order reduction techniques [15–17] are based on the observation that concurrent units may execute independently. Result will be the same disregarding of execution order if two executions are undistinguishable with respect to the specification. Thus, verification can be performed on a reduced state space. The assumption is that removed states are of no importance for results. Our approach cover the original state-space. Hence, our method do not require executions to

be undistinguishable with respect to the specification in order to perform the partitioning.

Several techniques (e.g., symmetry reduction, state space collapsing, etc) for reducing the problem of state space explosion have been applied to the Java PathFinder (JPF) [18]. Here, we focus on their technique to distribute model checking and to create dynamic partitions. Distributed model checking is achieved by partitioning the state space and letting each node handle one partition. When a new state is encountered, it is sent to the node that handle the partition it belongs to. A cache is used to decrease the number of sent messages by remembering if a certain state already have been sent. Our method does not need more than one message per partition since the partitions are independent from each other. Moreover, we can handle more than one partition per node by model checking them one after the other. Dynamic partitioning used in JPF simply means that states that belongs to a certain node, partition, to begin with can be moved to a other node when necessary (e.g., when there is a lack of memory). Our method partitions the state space dynamically, as it is explored.

A final observation is that our method is orthogonal to all methods described here. This means that it is possible to use our method to define a set of partitions and then apply any of the above methods for model checking each of the generated partition.

2 Preliminaries

The theory of timed automata has proven to be useful for specification and verification of real-time systems. In this Section we briefly review the basic definition needed in this paper. We refer the reader to [19] for a more thorough description.

Assume a finite set of real-valued variables $\mathcal{C}$ standing for clocks, and a finite alphabet Σ standing for actions. Let $\mathcal{B}(\mathcal{C})$ denote the set of Boolean combination of clock constraints of the form $x \sim n$ or $x - y \sim n$, where $x, y \in \mathcal{C}$ and n is a natural number.

Definition 1. *A* timed automaton $\mathcal{A}$ *is a tuple* $\langle N, l_0, E, I \rangle$ where*:*

- N *is a finite set of locations,*
- $l_0 \in N$ *is the initial location,*
- $E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times N$ *is the set of edges, and*
- $I : N \to \mathcal{B}(\mathcal{C})$ *assigns invariants to locations.*

Definition 2. *The* semantics of a timed automaton *is a timed transition system with states of the form* $\langle l, u \rangle$*, where* $l \in N$ *and* u *is a clock assignment assigning all clocks in* $\mathcal{C}$ *to a non-negative real-number. Transitions are defined by the two rules:*

- *(discrete transitions)* $\langle l, u \rangle \xrightarrow{a} \langle l', u' \rangle$ *if* $\langle l, g, a, r, l' \rangle \in E$*,* $u \in g$*,* $u' = [r \mapsto 0]u$ *and* $u' \in I(l')$
- *(delay transitions)* $\langle l, u \rangle \xrightarrow{d} \langle l, u \oplus d \rangle$ *if* $u \in I(l)$ *and* $(u \oplus d) \in I(l)$ *for a non-negative real* $d \in \Re_+$

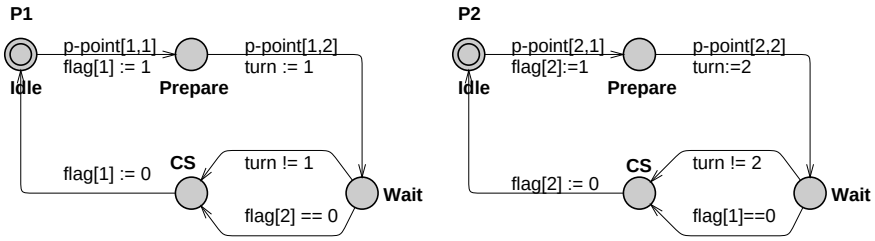


Fig. 1. Tie-breaker algorithm annotated with p-points, $p-point[i,j]$, where i indicates process identity and j is an enumeration of the transition

where $u \oplus d$ denotes the clock assignment which maps each clock x in $\mathcal{C}$ to the value $u(x) + d$, and $[r \mapsto 0]u$ is the clock assignment u with each clock in r reset to zero.

Definition 3. *A* run *of a timed automata* $\mathcal{A} = \langle N, l_0, E, I \rangle$ *with initial state* $\langle l_0, u_0 \rangle$ *over a timed trace* $\xi = (t_1, a_1)(t_2, a_2)(t_3, a_3)...$ *is a sequence of transitions:*

$$\langle l_0, u_0 \rangle \xrightarrow{d_1}\xrightarrow{a_1} \langle l_1, u_1 \rangle \xrightarrow{d_2}\xrightarrow{a_2} \langle l_2, u_2 \rangle \xrightarrow{d_3}\xrightarrow{a_3} \langle l_3, u_3 \rangle ...$$

satisfying the condition $t_1 = d_1$ *and* $t_i = t_{i-1} + d_i$ *for all* $i \geq 1$. *The* timed language $L(\mathcal{A})$ *is the set of all timed traces* ξ *for which there exists a run of* $\mathcal{A}$ *over* ξ.

2.1 A Motivating Example

Consider the automata specifying the tie-breaker algorithm [20] in Figure 1. Our goal is to establish mutual exclusion, i.e., the global invariant $\neg(\mathsf{P1.CS} \wedge \mathsf{P2.CS})$. The generated state space is searched for a state where both P1 and P2 are in their location CS. Suppose that we could identify some transitions in the state space of the model, and that we could make the model checker explore only the part of the state space where these transitions are taken in a given order. That is, the model checker would only explore the partition of the state space containing states reachable when the given global transitions are taken in the specified order. Further, suppose that we could repeat this procedure for all potential orders, we would then cover the original state space of the model. An obvious drawback would, of course, be that we could have as many partitions as there are execution traces through the automata.

Suppose that we instead select a subset of the edges in the automata and use the global order of the transitions derived from these edges to partition the state space. In this way, the number of partitions is decreased since transitions excluded from the subset may be taken in any order.

In the Tie-breaker example we select a subset of the edges as such partitioning points. In Figure 1 they are denoted p-point[i,j] where i is the process identity,

and j is an index in an enumeration of the chosen edges. Thus, we call the partition points *p-points*. Any trace $\xi \in L(\mathcal{A})$ will traverse a subset of these p-points in some order. We call such sequence of traversed p-points a *p-path*. Introducing the p-points in Figure 1 results in a finite set of p-paths if the model-checker is able to detect loops.

Definition 4. *Let pp_ξ be the sequence of p-points that a trace $\xi \in L(\mathcal{A})$ traverses (possibly the empty sequence). Let PP be the set of all p-paths pp of an automaton $\mathcal{A}$. We define $F(\xi, pp)$ to be the predicate such that:*

$$F(\xi, pp) = \begin{cases} true & \text{if } pp_\xi \text{ is a prefix of } pp, \\ false & \text{otherwise.} \end{cases}$$

We say that a trace ξ follows a p-path $pp \in PP$ if $F(\xi, pp)$.

The basic observation is that we can partition the state space with respect to these p-paths. Each p-path have a unique sequence of p-points. Each trace ξ is bound to follow at least one of the p-paths and is therefore included in at least one of the partitions. Instead of verifying a property in the complete state space of a model, we can check the property with respect to the partition defined by one p-path at a time:

FORALL $pp \in PP$ DO
  FORALL ξ such that $F(\xi, pp)$
    verify property

The mechanism we use for verification of mutual exclusion as a global invariant in our Tie-breaker example is a reachability mechanism. With this, we can ask the model checker whether there is a trace to a state where the invariant is falsified. This mechanism is denoted $\exists \Diamond$. In our Tie-breaker example, we have:

FORALL $pp \in PP$ DO #All p-paths generated from model
  FORALL ξ such that $F(\xi, pp)$
    $\exists \Diamond (P1.CS \wedge P2.CS)$

In order to take advantage of the p-paths, we have the following issues: (*i*) selection of p-points (*ii*) limitation of p-path length, (*iii*) identification of all p-paths, and (*iv*) prevention of the model checker searching outside partition. We will discuss these issues further in Section 5. In the next section, we present an algorithm for generating partitions using a standard model checker supporting reachability analysis, and in Section 4 we shall see how the algorithm is applied in a large case study.

3 Partitioning and Model-Checking

In this Section, we describe a method that dynamically generates partitions for a given model (Section 3.1). We also show how model-checking is applied to the generated partitions (Section 3.2).

3.1 Partition Generation

The method for generating partitions presented here uses an extra automaton that will guide the model checker. The basic idea is that the automata of the original model will synchronize with the guiding automaton at each p-point (see Figure 2). The guiding automaton uses a constant array, Ppath, and a constant integer, Length, to store information about the p-path to be exercised. The array Ppath contains the current p-path and the value of Length is the length of the p-path. Synchronization at p-point j as the $(i+1)$th traversed p-point is successful only if the following is true:

1. $\mathsf{Ppath}[i] = j \wedge i < \mathsf{Length}$, *or*
2. $i =$Length,

where $\mathsf{Ppath}[i-1]$ is the ith element of the array Ppath. Intuitively, (1) holds if the execution stays within the partion specified by Ppath, and (2) is true if the end of the partition is reached.

In Figure 2 we see an example of a guiding automaton (the lower of the three automata) that is used to partition the state-space of the Tie-breaker algorithm. Note how the two Tie-breaker process automata synchronizes with the guiding automaton whenever a p-point is reached. Note also that when the end of current p-path is reached (the edge guarded $i =$ Length in the guiding automaton), and hence a potential extension of the p-path is found, the identity of the current p-point extending the p-path is stored in a variable Next. In this way, the guiding automaton will guide the state-space exploration to stay within the partition, by following the p-path, and finally identify how to extend the p-path.

To generate all partitions, represented by p-paths, without modifying the applied model-checker, dynamic manipulation of both the verified safety property and the model files is required. The algorithm for generating partitions is shown in Figure 3. The algorithm uses a stack to store information about the current set of generated partitions. Each stack item is a pair $\langle pp, n\rangle$, where pp is a prefix of a complete p-path, and n is its length. We shall use ϵ to represent the empty path, and $pp :: q$ to represent the result of appending the paths q to pp. The algorithm also uses a model file containing the model of the system, and a property file containing the property to be verified. We use properties of the form $\exists \Diamond \phi$ to specify that a state satisfying ϕ is reachable in the model.

Initially, the stack has one element $\langle pp_0, n_0\rangle$, where $pp_0 = \epsilon$ and $n_0 = 0$, the query in the property file is $\exists \Diamond \mathsf{Next} \neq 0$, and the model file is modified by setting the values for the constant array Ppath to pp_0 and Length to n_0 with values from the stack (i.e. ϵ and 0). The initial property is satisfied as soon as a possible continuation is found, i.e. when a process synchronizes with the guiding automaton at a p-point. From the diagnostic trace generated by the model-checker, we can extract from the value of Next a possible continuation of pp_0. We denote this value pp_1^1. At this point, we do two things:

- $\langle pp_0 :: pp_1^1, n_0 + 1\rangle$, is pushed onto the stack, and
- call the model checker with the extended query $\exists \Diamond (\mathsf{Next} \neq 0 \wedge \mathsf{Next} \neq pp_1^1)$

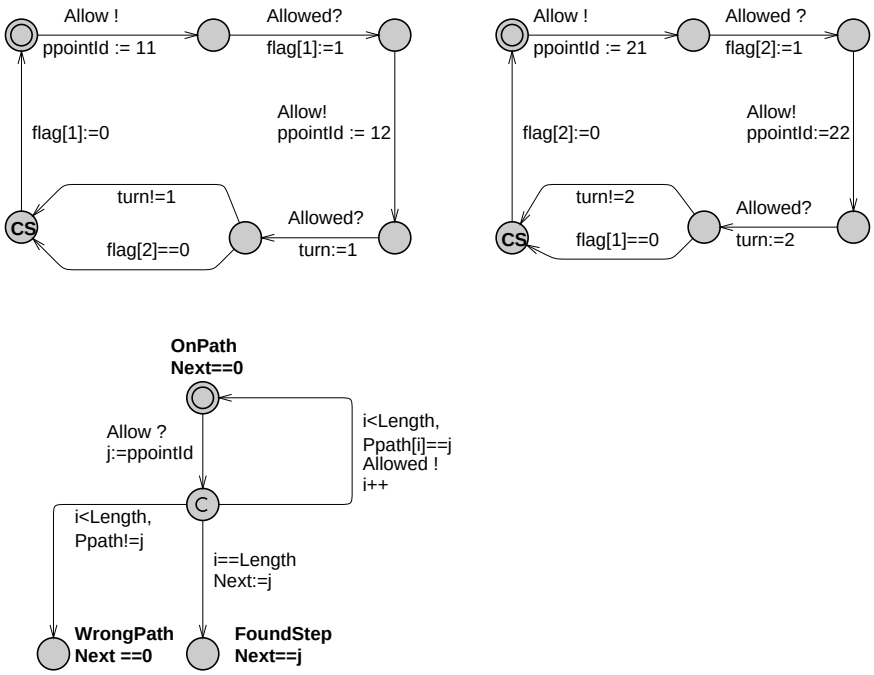


Fig. 2. Tie-breaker model extended with a guiding automaton, which guides the search for next p-point

This is repeated until the query $\exists \Diamond (\mathsf{Next} \neq 0 \wedge \mathsf{Next} \neq pp_1^1 \wedge ... \wedge \mathsf{Next} \neq pp_1^n)$ cannot be satisfied. At this point, there are n p-paths on the stack: $\langle pp_1^1, 1\rangle, ..., \langle pp_1^n, 1\rangle$.

In each iteration, a new pair $\langle pp_i^j, n_i\rangle$ is popped. We use these values to set $\mathsf{Ppath}= pp_i^j$ and $\mathsf{Length}= n_i$ in the model file. The query in the property file is re-initiated to $\exists \Diamond \mathsf{Next} \neq 0$. The above procedure is then repeated until all possible continuations of pp_i^j are pushed on the stack.

The finding of a complete p-path is recognized by the algorithm each time the initial query $\exists \Diamond \mathit{Next} \neq 0$ returns false, meaning that there is no continuation of current p-path. Each time this happens, we have the unique sequence of p-points defining a p-path.

The algorithm terminates when the stack is empty. At this point we have identified all p-paths, and thereby all partitions, of the model. A state that is reachable in the original state space is reachable when following one (or several) of the identified p-paths. We shall see in the next section how to modify the algorithm slightly, so that each partition is model checked for the original property to be verified.

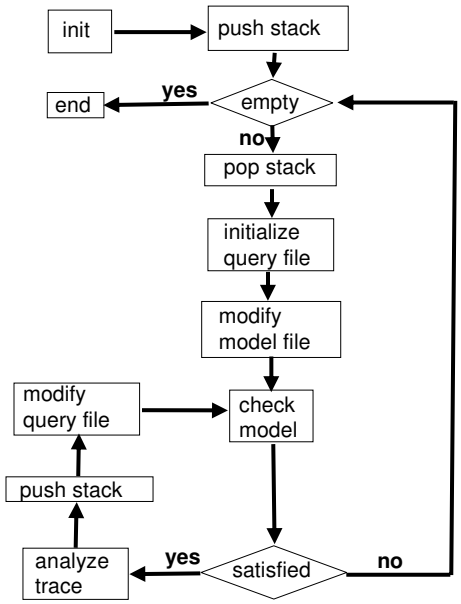


Fig. 3. Algorithm for finding the p-paths

Correctness: The algorithm presented here will in general not terminate for the (only) reason that the iterator i in the guiding automaton (see Figure 2) will in general cause loops of the state space to unfold. We declared variable i as a *hidden* (or *meta*) variable. Such variables are used to annotate a model, but are not considered when states are compared during the state-space generation. This means that states that only differs by value of i will be considered as equivalent by the model-checker during the state-space exploration. The use of a hidden variable guarantees that there is no infinite p-path. Hidden variables can be found in e.g. the tools SPIN[5] and UPPAAL [6]. We also have partial correctness since the algorithm generates the full state space of the original model.

3.2 Model-Checking Partitions

Section 3 describes how to find all partitions of a model. Since we do not want to continue the search for new partitions if the property to be verified is satisfied (or violated) in one the already defined partitions, we extend the algorithm to verify the property each time a complete p-path is found (see Figure 4).

The variable found keeps track of the number of potential continuations of a p-path. The condition found > 1 implies that there exists a continuation. If found $= 0$ and the model-checker returns false, there is no potential continuation

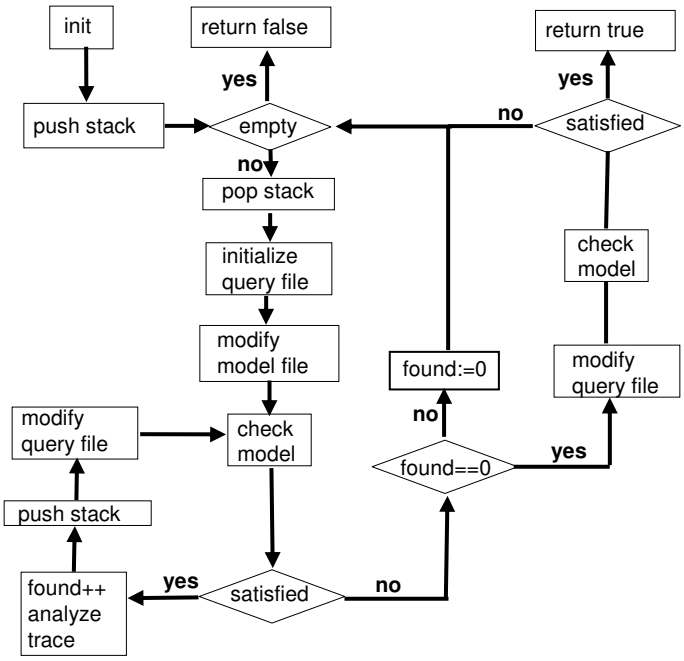


Fig. 4. The partitioning algorithm extended with property verification.

of current p-path. Since the current model contains a complete p-path together with the guiding automaton, we can check the model for the property to be verified. E.g. in the Tie-breaker example, we would check the mutual exclusion property $\exists \Diamond (P1.CS \wedge P2.CS)$ in all completed partitions.

The algorithm terminates when the property is satisfied or the stack is empty, indicating that all partitions have been checked.

4 Case Study: Execution Orders

4.1 Motivation

The motivation for our case study is to investigate the relation between execution environment and testability when testing for timeliness properties. It has previously been shown that testability is inherently low in dynamic, event-triggered real-time systems when compared with corresponding time-triggered systems [21]. Existing theoretical bounds for testability are based on parameters of the execution environment (e.g., potential preemptions) [22]. The goal with our study is to evaluate this theory. We model a dynamic, event-triggered real-

time system with its execution environment. The parameters of the execution environment are varied and the effect on system testability is assessed.

Testability is, however, not possible to measure directly. We, therefore, need another metric with which we can estimate the effect on testability. We have chosen to focus on reproducibility, which is a hard issue when testing event-triggered real-time systems for timeliness. *Reproducibility* is the property that the system repeatedly exhibits identical behavior when stimulated with the same test case. This is hard to achieve in event-triggered systems. The reason is that the actual behavior of a system depends on elements that have not been expressed explicitly as an input to the system (e.g., varying efficiency of hardware acceleration components). This means that what we judge to be a repeated test case might lead to different behaviors due to elements that we do not control. The problem is especially difficult when we try to provoke the system to miss deadlines. In these situations we will enforce bursts of events that causes a high system load and frequent interrupts. This may lead to race conditions and, thereby, variations in the behavior with respect to time and order.

For the reasons given above, we have chosen to use the number of behaviors with respect to *execution orders* (i.e., interleavings among a set of tasks) as a reasonable approximation for testability of dynamic, event-triggered systems when testing of timeliness properties are in focus.

The basic idea is to use sequences of events (identified by type, time of occurrence, and possibly additional parameters) as test cases. For each selected test case, we count the number of potential execution orders. By varying values of the execution environment parameters and studying the effect on the number of execution orders, we can estimate how testability is affected by the execution environment parameters.

4.2 Approach

A real-time system is modeled in timed automata and model checking is used to explore its behavior. We vary the values of selected execution environment parameters and count the number of potential execution orders. The behavior of the system is explored to find all potential execution orders. UPPAAL is used to explore the system behavior.

A picture of the modeled system is shown in Figure 5. Tasks involved in execution orders are part of the application: T_1, ..., T_m, where T_j is a timed automata process corresponding to a task. Controlled environment is restricted to a set of events: e_1, ..., e_n, where e_i is a timed automata process corresponding to an event (e.g., a sensor signal). Execution environment consists of observer, scheduler, and resource handler. The observer, observes events in the controlled environment and communicates with the scheduler. The system is dynamic. Tasks are triggered on event observations. Tasks are scheduled according to their deadlines in an EDF (earliest deadline first) manner. Execution of tasks are conducted in non-preemptive intervals and each time a task is given access to the processor, a p-point is passed. In this specific case, each p-path is an execution order.

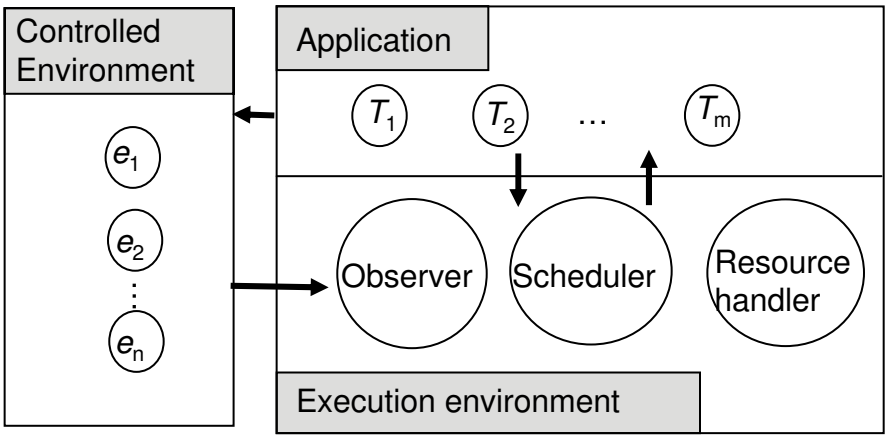


Fig. 5. The modeled system

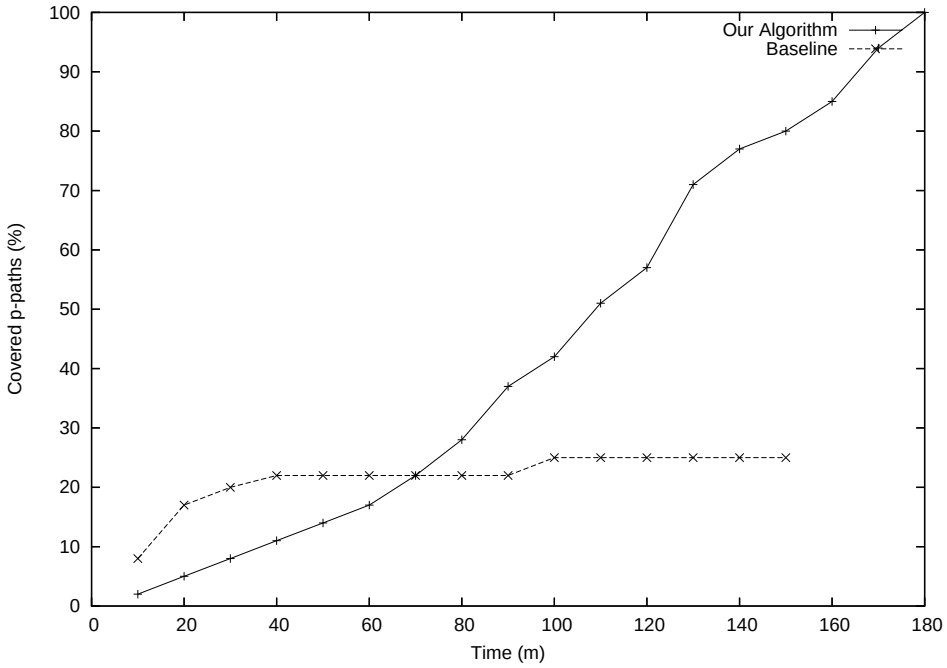
In order to evaluate the performance of our algorithm, we compare it to a base-line algorithm. The basic idea with the base-line algorithm is to ask the model-checker to return complete p-paths, one at a time. In this approach, we need a defined final state S. The initial query asks the model checker if S is reachable, i.e., $\exists \Diamond S$. The generated trace gives us a complete p-path, pp_1. The query is then extended to $\exists \Diamond S \wedge \neg pp_1$. This is repeated until the query $\exists \Diamond S \wedge \neg pp_1 \wedge ... \wedge \neg pp_n$ is falsified and, hence, there are no more p-paths.

The base-line approach does not require any alterations of the model during the search. However, in our case study, the model is too complex for exhaustive search and the more orders we find, the larger is the state space that we need to explore to find new ones. At some point in our search we will, therefore, fail to continue due to memory consumption. Hence, part of state space is not explored and the set of found execution orders may not be complete.

We applied our algorithm and the base-line algorithm on the same model[1]. Table 1 shows that our algorithm managed to cover all p-paths in the model within 3 hours. The base-line algorithm only managed to cover 26% of the p-paths. At that point the model checker stopped with an error message, "out of memory".

[1] Experiment was performed on US-II sparc 12 CPU multiprocessor with 400MHz, 4Mb cache, and 6144Mb memory. The search order was depth-first.

Table 1. Percentage of p-paths covered by our algorithm and the base-line algorithm



4.3 Parallel Verification of Partitions

The proposed algorithm for state-space partitioning described in Section 3 split the search space into partitions along p-paths. In the given description of the algorithm, we find one partition at a time, using a stack to remember where to continue the search. However, it is important to note that each item on the stack contains enough information for an independent continuation of the search for partitions. The fact that all partitions are independent makes this method particularly suitable for parallelization of verification. It is possible to distribute the search over several processors, thereby performing a more time-efficient search.

We ran our experiments on a 12 CPU multiprocessor, thereby increasing efficiency by an order of magnitude. The limitations of this approach is, of course, the size of the partitions. Threading on a multiprocessor implies shared memory and might, hence, invoke the problem of insufficient memory again. In such case, the partitions can easily be distributed over separate processors. The cost for distribution in terms of communication is bounded to the number of partitions since there is only need for one message per p-path.

5 Discussion

In a complex model, we will not know the p-paths beforehand. The algorithm presented in this paper does therefore, collect the p-paths iteratively. We present

an approach to ensure that the model checker does not search outside the current partition. The approach is based on the idea of including the specification of current partition in the model. Hence, states outside partition cannot be generated.

The choice of partitioning points decides how the state space is partitioned. All states will be included in at least one partition and therefore, the choice has no impact on completeness. However, the choice might affect performance. This paper does not provide any general method for selection of partitioning points since we believe that this is application specific. However, we have two general advices that has shown to be a good strategy in our own experiments.

Our first advice concerns the number of partitioning point. Few partitioning points will give few, but large partitions and, hence, the problem of memory consumption might remain. Many points will give a large number of partitions, (i.e., if all transitions are selected as partitioning points, we would get a p-path for each possible trace). It is therefore, necessary to find a balance where the partitions are sufficiently small. Where that balance is depends on the model and resource limitations. Hence, this is a decision that must be left to the user.

Our second advice concerns the placement of the partitioning points. In order to get as little overlap of different partitions as possible it is recommended that the p-points are encountered as close to the start of execution of the model as possible. In a complex and non-deterministic model, it might be a difficult task to identify the optimal partitioning points. However, this is not a critical task for correctness. Even with a less optimal placement of the partitioning points, the original state space is covered by the partitions.

The use of a hidden variable in the presented method prevent it from unfolding of loops in the symbolic state-space of the analyzed model. In other case it is a risk, that states normally treated as equivalent (or included in other states) by an ordinary (symbolic) reachability analysis algorithm can become non-equivalent (or not included). In such cases, the algorithm could theoretically slow down termination. By hiding the information of the progress along the p-path (i.e., the value of the iterator) in a hidden variable we overcome this problem.

6 Conclusions

A major problem when verifying complex timed automata models is memory consumption. We present a method that dynamically divides state space into smaller (with respect to memory requirements) partitions. These partitions are independent and can, therefore, easily be distributed over several nodes.

Our experience from using the method is good. We applied it on a large case study with very promising results. The method enabled us to search the state space generated by a timed automata model of a dynamic real-time system. Using the same memory resources, our method could generate all traces of interest, whereas ordinary model-checking could cover only 26% of them (see table 1).

Since our method is orthogonal to other methods that address the same problem (see Section 1) it is possible to combine our method for state space partitioning with any other method when model checking each such partition. However, further evaluation is needed. One question for further work is how the distribution of p-points affects efficiency of the method.

References

1. E.M.Clarke, A.Emerson: Synthesis of synchronization skeletons for branching time temporal logic. In Logic of Programs: Workshop, Lecture Notes in Computer Science **131** (1981) 52–71
2. Queille, J.P., Sifakis, J.: Specification and verification of concurrent programs in CESAR . In: Proc. 5*th* Int. Symp. on Programming. Number 137, Berlin, Springer–Verlag (1982) 195–220
3. K.L.McMillan: Symbolic Model Checking: An Approach to the State Space Explosion Problem. PhD thesis, Carnegie Mellon University (1992)
4. E.M.Clarke, A.Cimatti, F.Giumchiglia, M.Roveri: NuSMV: A new symbolic model checker. Software Tools for Technology Transfer **2**(4) (2000) 401
5. G.J.Holzmann: The model checker SPIN. IEEE Transactions on Software Engineering **23**(5) (1997) 279–295
6. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Int. Journal on Software Tools for Technology Transfer **1**(1–2) (1997) 134–152
7. Daws, C., Olivero, A., Tripakis, S., Yovine, S.: The tool KRONOS. In: Hybrid Systems III: Verification and Control. Volume 1066 of Lecture Notes in Computer Science., Springer-Verlag (1995)
8. Clarke, E.M., Wing, J.M.: Formal Methods: State of the Art and Future Directions. ACM Computing Surveys **28**(4) (1996) 626–643
9. G.J.Holzmann: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
10. Bengtsson, J., Yi, W.: Reducing memory usage in symbolic state-spac exploration for timed systems. Technical report, Department of Information Technology, Uppsala University (2001)
11. Bengtsson, J., Yi, W.: On clock difference and termination in reachability analysis of timed automata. In: Formal Methods, ICFEM 2003. Volume 2885 of Lecture Notes in Computer Science., Springer-Verlag (2003)
12. Behrmann, G., Larsen, K., Pearson, J., Weise, C., Yi, W.: Efficient timed reachability analysis using clock difference diagrams. In: Eleventh International Conference on Computer Aided Verification. Volume 1633 of Lecture Notes in Computer Science., Springer-Verlag (1999) 341–353
13. G.Behrmann: Distributed reachability analysis in timed automata. International Journal on Software Tools for Technology Transfer, (STTT) **7**(1) (2005) 19–30
14. A.Biere, A.Cimatti, E.M.Clarke, O.Strichman, Y.Zue: Bounded model checking. Advances in Computers **58** (2003)
15. Alur, R., Brayton, R., Henzinger, T., Quadeer, S., Rajmani, S.: Partial order reduction in symbolic state space exploration. In: Proceedings of the Conference on Computer Aided Verification (CAV'97), Haifa, Israel (1997)
16. Godefroid, P.: Using partial orders to improve automatic verification. In: 2nd Workshop on Computer Aided Verification. LNCS 531, New Brunswick, NJ, Springer-Verlag (1990) 176–185

17. Peled, D.: All from one, one for all, on model-checking using representatives. In: 5th Conference on Computer Aided Verification. LNCS, Greece, Springer-Verlag (1993) 409–423
18. Lerda, F., Visser, W.: Addressing dynamic issues of program model checking. Lecture Notes in Computer Science **2057** (2001) 80–102
19. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In Reisig, W., Rozenberg, G., eds.: In Lecture Notes on Concurrency and Petri Nets. Lecture Notes in Computer Science vol 3098, Springer–Verlag (2004)
20. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3) (1981) 115–116
21. Schütz, W.: The Testability of Distributed Real-Time Systems. Kluwer Academic Publishers (1993)
22. Birgisson, R., Mellin, J., Andler, S.: Bounds on Test Effort for Event-Triggered Real-Time Systems. In: The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99). (1999)