Generating Trace-Sets for Model-based Testing

Birgitta Lindström
University of Skövde, P.O. Box 408
SE-541 28 Skövde, Sweden
E-mail: birgitta.lindstrom@his.se

Paul Pettersson
Mälardalen University, P.O. Box 883
SE-721 23, Västerås, Sweden
E-mail: paul.pettersson@mdh.se

Jeff Offutt
George Mason University
Fairfax, VA 22030, USA
E-mail: offutt@gmu.edu

Abstract

Model-checkers are powerful tools that can find individual traces through models to satisfy desired properties. These traces provide solutions to a number of problems. Instead of individual traces, software testing needs sets of traces that satisfy coverage criteria.

Finding a trace set in a large model is difficult because model checkers generate single traces and use a lot of memory. Space and time requirements of model-checking algorithms grow exponentially with respect to the number of variables and parallel automata of the model being analyzed.

We present a method that generates a **set** *of traces by iteratively invoking a model checker. The method mitigates the memory consumption problem by dynamically building partitions along the traces. This method was applied to a testability case study, and it generated the complete trace set, while ordinary model-checking could only generate 26%.*

1. Introduction

In the last decade, model checking [7, 17] has developed into a powerful technique for automatic formal verification of transition systems. Researchers have developed several verification tools, including SMV [15, 10] and SPIN [12] for finite state systems, and UPPAAL [14] and KRONOS [9] for real-time systems modeled as timed automata. These tools have been applied to several non-trivial industrial systems [8].

A model checker can accept a state-based model and a property, and find a trace through the model that satisfies (or contradicts) that property if such a trace exists. Common properties are to prove global invariants or to show that some state can be reached. Model checking can also be used for job scheduling; for example, to find a job schedule that gives high throughput and sufficient product quality. These applications need individual traces, one for each property. Test case generation, however, needs **sets of traces** that collectively satisfy a test criterion. In concurrent systems faults are sometimes related to the ordering of certain events such as synchronization or access to CPU or shared data. A test suite covering different ordering of such events can therefore, be useful. In real-time systems, covering execution orders has been used to support structured testing [16, 19]. A key insight to this research is to generate sets of traces by iteratively invoking the model checker, where each new trace must differ from the previous traces with respect to these orders. Unfortunately, real-time models tend to be complex, with many states, and the *state-space explosion problem* of model checkers [13] means it is difficult to exhaustively analyze the models. The state-space explosion problem refers to the exponential size of state space w.r.t. the size of the input model. Several attempts have been made to reduce the memory usage of model-checking algorithms [2, 3, 4]. However, memory and time still remains a bottleneck in model checking.

This paper presents a new method for generating sets of traces. The method mitigates the space and memory limitations by dynamically partitioning the state space along the traces as they are generated. This limits the state-space that must be searched for individual invocations of the model checker.

In this method, a trace through a timed automaton is a sequence of edges that are traversed in a given order. The final set of traces gives all unique sequences with

which these selected edges can be traversed in the given model. Each trace is generated and extended step-wise by guiding the original model, so that the search for an extension stays within the partition where the extension can be found. We show how this can be done using an existing model checker and dynamic manipulation of the analyzed model.

The method uses the UPPAAL tool [14], and is applied to a large case study of a real-time application. The case study investigates the relationship between testability and properties of the execution environment. One goal is to study the effect on the number of *execution orders* (interleavings among a set of tasks) when varying parameters. We provide data from this case study, and report that our method could generate all traces of interest (in this case all execution orders) whereas ordinary model-checking was only able to cover 26% of them, using the same memory resources.

The remainder of this section discusses related work. Section 2 presents preliminary results and give a motivating example of our method. Sections 3 and 4 present our method in detail, and explain how it was applied in a case study. Sections 5 and 6 discuss how to use our method and provide conclusions.

Related Work: The idea to use a model checker to generate test cases is not new. Whenever a test criterion can be expressed as a property verifiable by a model checker, such techniques for test case generation are useful. For example, Visser et al. [20] presented a framework built on top of the Java PathFinder (JPF) to automatically generate test inputs for Java programs. The basic idea is that the test criterion is expressed as a safety property ϕ and symbolic execution of a path that satisfies ϕ generates a set of constraints. A constraint solver then gives the input that fulfills the constraints, i.e., executes the path where ϕ is satisfied. Garganti and Heitmeyer [11] present a method for obtaining a test input sequence from a system property and a Software Cost Reduction requirements specification. Ammann et al. [1] present a mutation analysis approach for test case generation with a model checker.

Schütz described the relation between execution orders and testability for distributed real-time systems [18]. Thane and Hansson [19] present a method to generate all execution orders for static real-time systems, arguing that there are not enough test methods for concurrent programs. They also discuss testability related to the number of execution orders in the sense that each order is similar to a sequential program. Hence, test methods for sequential programs can be applied to concurrent programs by testing each execution order as a sequential program. Nilsson et al. [16] generate test cases based on execution orders.

2. Preliminaries

Engineers commonly use timed automata to specify and verify real-time systems. This section reviews definition used in this paper. Bengtsson and Yi [5] have more details on these concepts.

Clocks are represented by a finite set of real-valued variables $\mathcal{C}$ and *actions* are represented by a finite alphabet Σ. Let $\mathcal{B}(\mathcal{C})$ denote the set of Boolean combinations of clock constraints of the form $x \sim n$ or $x - y \sim n$, where $x, y \in \mathcal{C}$, n is a natural number and $\sim$ represents one of the relational operators $\{>, \leq, =, \geq, >\}$.

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*
- $I : N \rightarrow \mathcal{B}(\mathcal{C})$ *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 of all clocks in* $\mathcal{C}$ *to non-negative real-numbers. 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* $d \in \Re_+$

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 automaton* $\mathcal{A} = \langle N, l_0, E, I \rangle$ *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*.*

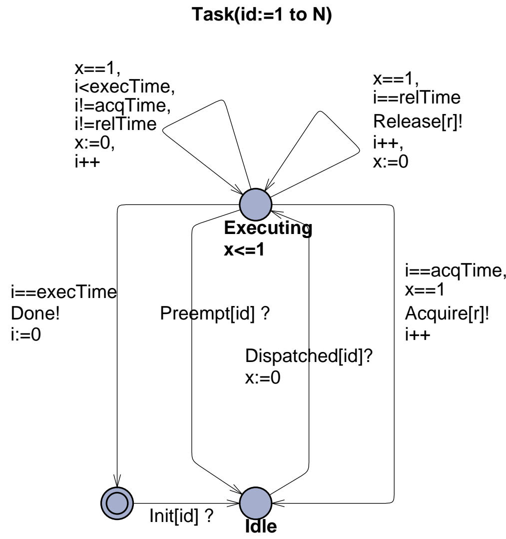


Figure 1. A simplified task model

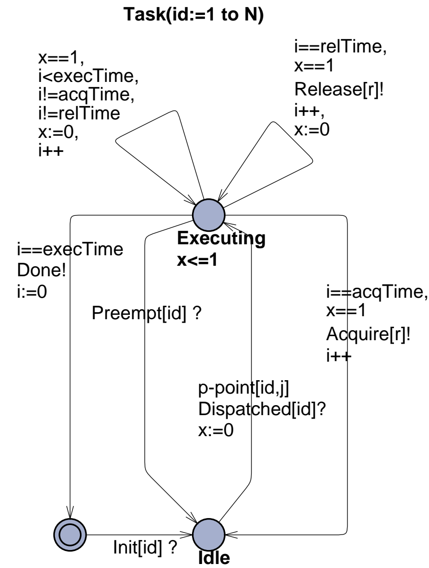


Figure 2. The simplified task model annotated with a p-point, *p-point[id,j]*, where id is the process identity, and j is an enumeration of the p-point

2.1. Example

The automaton in Figure 1 specifies a simple model of a task. The task is executed by a process that is part of a concurrent system with *N* processes executing similar tasks. When initiated, the task alters between the states Idle and Executing until it has been in state Executing for *execTime* time steps. At time *acqTime* a request for resource *r* is made and at time *relTime* the resource is released. We assume that there is a scheduler with which the automaton synchronizes via the channels Dispatch, Preempt and Done. The automaton synchronizes with a resource handler via the channels Acquire and Release.

For testing, we want to select a subset of the edges in the automaton and use the global order of the transitions derived from these edges to define categories of test cases. Which edges to choose depends on the test strategy. If the strategy focuses on execution orders, we choose the edges where the tasks synchronize with the scheduler, labeled *Dispatched[id]*? in Figure 1. By generating all global orderings with that can traverse these edges, we get all execution orders. If, on the other hand, the strategy focuses on shared resources, we choose the edges where the tasks synchronize with the resource handler instead. These are labeled *Release[r]!* and *Acquire[r]!* in Figure 1.

3. Partitioning and Model-Checking

This section describes a method that dynamically generates all "interesting" traces in a given model. In this situation, "interesting" means unique with respect to the order in which a subset of transitions are taken.

We mark edges that dispatch tasks with special markers called *p-points*. A *p-point* is denoted p-point[id,j], where id is the process identity, and j is a number associated with that p-point. This is shown in Figure 2. Any trace $\xi \in L(\mathcal{A})$ will traverse a subset of the p-points in some order. We call the sequence of p-points traversed a *p-path*. Figure 2 contains a p-point on the edge labeled *Dispatch[id]*?.

If the model checker can detect loops[1], then it can generate a finite set of p-paths through the model in Figure 2. The set of p-paths in this example describes the set of execution orders since the *p-point* is placed on the edge where the task synchronizes with the scheduler.

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 $F(\xi, pp)$.

Collecting all p-paths results in several problems (*i*) limiting p-path length, (*ii*) identifying all p-paths, and (*iii*)

[1] A reasonable requirement since many model checkers, including UPPAAL [14] and SPIN [12], do detect loops.

memory consumption. These issues are discussed in Section 5. Next section presents an algorithm for generating p-paths while mitigating the state space explosion problem using a standard model checker that supports reachability analysis, and Section 4 shows how the algorithm is applied in a large case study.

We generate p-paths by iteratively invoking the model checker with queries that extend an existing p-path prefix by one step. An extra, *guiding automaton*, is used to guide the model checker in its search for the one-step extensions. The basic idea is that the original automaton will synchronize with the guiding automaton at each p-point (as shown in Figure 3). The guiding automaton uses an array, Ppath, to store current p-path prefix and an integer, Length, to store the length of the p-path prefix. Synchronization at p-point j as the $(i+1)$th traversed p-point is successful only if the following is true:

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

where $Ppath[i-1]$ is the ith element of the array *Ppath*. Intuitively, (1) holds if the execution follows the p-path prefix specified by *Ppath*, and (2) holds if an extension to the p-path prefix is reached.

Figure 3 shows a guiding automaton (on the right). The left automaton is the simple task model extended with two new edges (labeled *Allow!* and *Allowed?*), which implement the p-point. Note how the left automaton synchronizes with the guiding automaton whenever a p-point is reached (i.e., when the left automaton is given access to the processor). 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 and the system deadlocks. Thus, the guiding automaton will prevent the state-space exploration from exploring states where extensions cannot be found. Any attempt to abandon the current p-path during exploration is bound to fail. The approach thereby mitigates the problem of memory consumption. By following the p-path, the process will find an extension if one exists. Using p-points and a guiding automaton allow the current p-path to be extended if possible. It also mitigates the state-space explosion problem since the search follows the p-path. To generate all complete p-paths without modifying the applied model checker, it is necessary to dynamically manipulate both the verified safety property and the model files. The algorithm for generating p-paths is shown in Figure 4.

The algorithm uses a stack, *S*, to store information about the current set of generated p-paths. Each stack item is a pair $\langle pp, n\rangle$, where pp is a p-path prefix and n is its length. We use ϵ to represent the empty p-path, and $pp \cdot q$ to represent the result of appending the path q to pp. The algorithm also uses files that contain the model of the system, *M*, and the property to be verified, *Q*. We use properties of the form $\exists\Diamond\phi$ to specify that a state satisfying ϕ is reachable in the model.

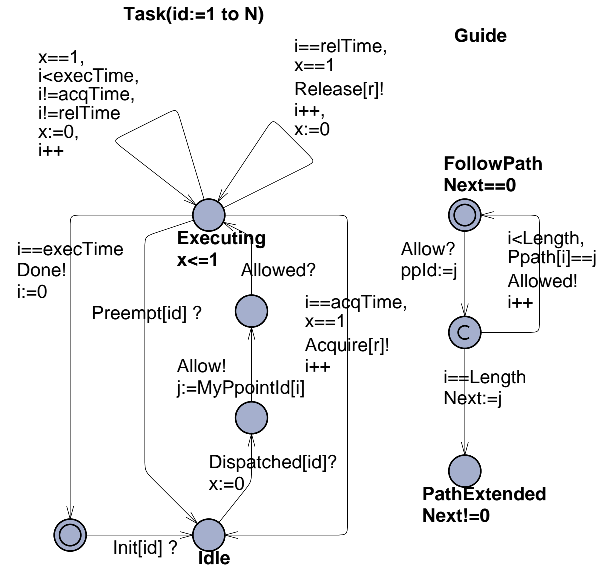


Figure 3. The simplified task model extended with a guiding automaton, which guides the search for next p-point

The stack starts with one element $\langle pp_0, n_0\rangle$, where $pp_0 = \epsilon$ and $n_0 = 0$, the query in the property file is initialized to $\exists\Diamond\, Next \neq 0$, 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:

- Push $\langle pp_0 \cdot pp_1^1, n_0 + 1\rangle$, onto the stack, and
- Call the model checker with the extended query $\exists\Diamond\, Next \neq 0 \wedge Next \neq pp_1^1$

This is repeated until the query $\exists\Diamond\,(Next \neq 0 \wedge Next \neq pp_1^1 \wedge ... \wedge Next \neq pp_1^n)$ cannot be satisfied. At this point, there are n p-paths on the stack: $\langle pp_1^1, 1\rangle, ..., \langle pp_1^n, 1\rangle$, each with length one. A new pair $\langle pp_i^j, n_i\rangle$ is popped in each iteration. We use these val-

while S **DO** $\rightarrow$
    pop S for pp and n
    Q = "$\exists\Diamond Next \neq 0$"
    copy original model file to M
    find and replace in M $\rightarrow$
        $Ppath[0] = \{\}$ with $Ppath[n] = \{pp\}$
        $Length = 0$ with $Length = n$
    $satisfied = true$
    $alternatives = 0$
    while satisfied **DO** $\rightarrow$
        invoke model checker with M and Q
        if $satisfied$ **DO** $\rightarrow$
          $alternatives++$
          find pp_i^j in generated trace
          push $< pp \cdot pp_i^j, n+1 >$ onto S
          $Q = Q \cdot$ "$\wedge Next \neq pp_i^j$"
        else if $alternatives == 0$ DO $\rightarrow$
          save pp # This is a complete p-path
        end
    end # All alternatives found
end # All p-paths found

Figure 4. P-path generating algorithm. pp **is the current sub p-path,** n **is its length,** S **is the stack,** Q **is the current query, and** M **is the file containing the model**

ues to set Ppath= pp_i^j and Length= n_i in the model file. The query in the property file is re-initiated to $\exists\Diamond$ Next $\neq 0$. The above procedure is then repeated until all possible continuations of pp_i^j with length $n_i + 1$ are pushed on the stack.

The algorithm finds a complete p-path when the initial query $\exists\Diamond Next \neq 0$ returns false, meaning that there is no continuation of current p-path. The result is a 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.

Termination: Introducing the iterator i in the guiding automaton (see Figure 3) might cause loops to be unfolded in the state space. The reason is that the model checker cannot recognize a previously explored state (and thereby avoid exploring it again) if i is incremented. We therefore, declare 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 to be equivalent by the model checker. Using a hidden variable guarantees that p-paths terminate. Hidden variables are common in model checking tools like SPIN [12] and UPPAAL [14].

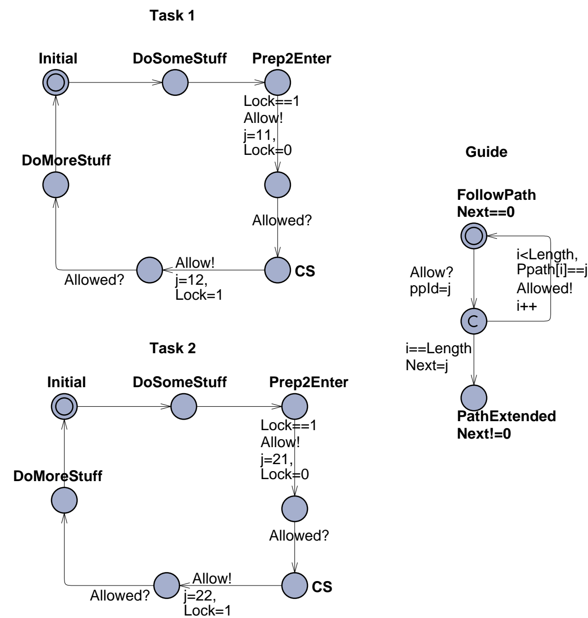


Figure 5. A simplified model using a binary semaphore Lock **to protect the critical section** CS**.**

3.1. A small Example

This subsection illustrates the algorithm with a step-by-step example, using the automata model in Figure 5. The model is simplified to focus how this algorithm works on the example. The model does not consider time, scheduling or the behavior of the tasks. The only detail left is a critical section (CS) and the protocol for *entry* and *exit*. The algorithm is used to generate all orders with which the tasks may enter and exit CS. Hence, we have selected the edges where semaphore Lock is used and inserted a p-point at each. As shown in Figure 5, the p-point identities are 11, 12, 21, and 22. P-points 11 and 21 are placed at the entries to CS for Task 1 and Task 2 respectively and p-points 12 and 22 are placed at the exits from CS for Task 1 and Task 2 respectively. We push the empty sequence ϵ and 0 onto the stack, S to begin the first iteration. Each iteration begins by modifying the model file, M, with values fetched from S. The query, Q, is modified before each invocation of the model checker. The first iteration begins by popping $< \epsilon, 0 >$ from S. In the first iteration (see Table 1) we find two alternatives for the first p-point, namely 11 and 21. This results in the two items $< 11, 1 >$ and $< 21, 1 >$

pushed onto the stack. The third query is falsified. Iteration 2 starts by popping the last pushed item, $<21, 1>$, from the stack. *M* is modified and the search is continued for a p-point that may extend p-path 21.

The second iteration gives p-point 22 as the only possible extension to p-path 21. The reason is that Task 2 has entered the critical section CS and the entry is locked for Task 1. The search is continued with p-path $21 \cdot 22$.

The third iteration finds two possible extensions to p-path $21 \cdot 22$. The extensions are p-points 11 and 21. The items $< 21 \cdot 22 \cdot 11, 3 >$ and $< 21 \cdot 22 \cdot 21, 3 >$ are pushed onto *S* and the search continues with p-path $21 \cdot 22 \cdot 21$ in iteration 4.

In the fourth iteration, the model checker returns false in response to the first query. This means that it is not possible to find an extension to p-path $21 \cdot 22 \cdot 21$ without re-entering a state that has already been visited, i.e., a loop. The p-path is complete and therefore, saved. We pop $< 21 \cdot 22 \cdot 11, 3 >$ and continue the search for an extension to p-path $21 \cdot 22 \cdot 11$ in iteration 5.

The fifth iteration finds one possible extension to p-path $21 \cdot 22 \cdot 11$, namely p-point 12. The search is continued for extensions to $21 \cdot 22 \cdot 11 \cdot 12$.

The sixth iteration finds two possible extensions to p-path $21 \cdot 22 \cdot 11 \cdot 12$. These are the p-points 21 and 11. The items $< 21 \cdot 22 \cdot 11 \cdot 12 \cdot 21, 5 >$ and $< 21 \cdot 22 \cdot 11 \cdot 12 \cdot 11, 5 >$ are pushed onto *S* and we continue with p-path $21 \cdot 22 \cdot 11 \cdot 12 \cdot 11$.

The seventh iteration finds no extension given p-path $21 \cdot 22 \cdot 11 \cdot 12 \cdot 11$. We have found our second complete p-path in this example. The p-path is therefore, saved. We pop $< 21 \cdot 22 \cdot 11 \cdot 12 \cdot 21, 5 >$ from *S* and continue the search with p-path $21 \cdot 22 \cdot 11 \cdot 12 \cdot 21$.

The eight iteration finds p-point 22 as the only possible extension to p-path $21 \cdot 22 \cdot 11 \cdot 12 \cdot 21$. The search is continued with p-path $21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22$.

The ninth iteration finds only one extension given the popped item $< 21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22, 6 >$, namely p-point 21. The search continues from p-path $21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22 \cdot 21$.

The tenth iteration finds no possible extension given $< 21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22 \cdot 21, 7 >$. $21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22 \cdot 21$ is the third complete p-path in our example and it is therefore, saved. The search continues with $< 11, 1 >$. At this point, we have covered all p-paths that begin with p-point 21, i.e., all orders where Task 2 is first to execute the entry protocol to CS. The saved p-paths so far are $21 \cdot 22 \cdot 21$, $21 \cdot 22 \cdot 11 \cdot 12 \cdot 11$, and $21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22 \cdot 21$. Continuing the search will give us the additional p-paths $11 \cdot 12 \cdot 11$, $11 \cdot 12 \cdot 21 \cdot 22 \cdot 21$, and $11 \cdot 12 \cdot 21 \cdot 22 \cdot 11 \cdot 12 \cdot 11$, 7 as shown in Figure 6. These are all orders where Task 1 is first to execute the entry protocol to CS. Since Task 1 and Task 2 implement the same behavior, the left part of Figure 6 is similar to the right part. When the last p-path beginning with p-point 11, i.e., p-path $11 \cdot 12 \cdot 21 \cdot 22 \cdot 11 \cdot 12 \cdot 11$, is found, the stack is empty and the algorithm terminates.

Table 1. The result from invoking the model checker with query Q and model M is shown in the rightmost column.

Iteration 1: $< \epsilon, 0 >$ is popped and the modified model *M* is checked for extensions to ϵ		
S:	$[\emptyset]$	
Q:	$\exists \Diamond Next \neq 0$	11
S:	$[< 11, 1 >]$	
Q:	$\exists \Diamond Next \neq 0 \wedge Next \neq 11$	21
S:	$[< 11, 1 >, < 21, 1 >]$	
Q:	$\exists \Diamond Next \neq 0 \wedge Next \neq 11 \wedge Next \neq 21$	F
It. 2: $< 21, 1 >$ is popped and the modified model *M* is checked for extensions to 21		
S:	$[< 11, 1 >]$	
Q:	$\exists \Diamond Next \neq 0$	22
S:	$[< 11, 1 >, < 21 \cdot 22, 2 >]$	
Q:	$\exists \Diamond Next \neq 0 \wedge Next \neq 22$	F
It. 3: $< 21 \cdot 22, 2 >$ is popped and the modified model *M* is checked for extensions to $21 \cdot 22$		
S:	$[< 11, 1 >]$	
Q:	$\exists \Diamond Next \neq 0$	11
S:	$[< 11, 1 >, < 21 \cdot 22 \cdot 11, 3 >]$	
Q:	$\exists \Diamond Next \neq 0 \wedge Next \neq 11$	21
S:	$[< 11, 1 >, < 21 \cdot 22 \cdot 11, 3 >, < 21 \cdot 22 \cdot 21, 3 >]$	
Q:	$\exists \Diamond Next \neq 0 \wedge Next \neq 11 \wedge Next \neq 21$	F
It. 4: $< 21 \cdot 22 \cdot 21, 3 >$ is popped and the modified model *M* is checked for extensions to $21 \cdot 22 \cdot 21$		
S:	$[< 11, 1 >, < 21 \cdot 22 \cdot 11, 3 >]$	
Q:	$\exists \Diamond Next \neq 0$	F
$\rightarrow$ **Save** $21 \cdot 22 \cdot 21$		
It. 5: $< 21 \cdot 22 \cdot 11, 3 >$ is popped and *M* is checked for extensions to $21 \cdot 22 \cdot 11$		
S:	$[< 11, 1 >]$	
Q:	$\exists \Diamond Next \neq 0$	12
S:	$[< 11, 1 >, < 21 \cdot 22 \cdot 11 \cdot 12, 4 >]$	
Q:	$\exists \Diamond Next \neq 0 \wedge Next \neq 12$	F
It. 6: $< 21 \cdot 22 \cdot 11 \cdot 12, 4 >$ is popped and *M* is checked for extensions to $21 \cdot 22 \cdot 11 \cdot 12$		
S:	$[< 11, 1 >]$	
Q:	$\exists \Diamond Next \neq 0$	21
S:	$[< 11, 1 >, < 21 \cdot 22 \cdot 11 \cdot 12 \cdot 21, 5 >]$	
Q:	$\exists \Diamond Next \neq 0 \wedge Next \neq 21$	11
S:	$[< 11, 1 >, < 21 \cdot 22 \cdot 11 \cdot 12 \cdot 21, 5 >, < 21 \cdot 22 \cdot 11 \cdot 12 \cdot 11, 5 >]$	
Q:	$\exists \Diamond Next \neq 0 \wedge Next \neq 21 \wedge Next \neq 11$	F

Figure 6. A tree showing all orders with which the p-points can be traversed in the model given in Figure 5

It. 7: $<21 \cdot 22 \cdot 11 \cdot 12 \cdot 11, 5>$ is popped and *M* is checked for extensions to $21 \cdot 22 \cdot 11 \cdot 12 \cdot 11$		
S:	$[<11, 1>, <21 \cdot 22 \cdot 11 \cdot 12 \cdot 21, 5>]$	
Q:	$\exists \Diamond Next \neq 0$	F
$\rightarrow$ **Save** $21 \cdot 22 \cdot 11 \cdot 12 \cdot 11$		
It. 8: $<21 \cdot 22 \cdot 11 \cdot 12 \cdot 21, 5>$ is popped and extensions to $21 \cdot 22 \cdot 11 \cdot 12 \cdot 21$ searched		
S:	$[<11, 1>]$	
Q:	$\exists \Diamond Next \neq 0$	22
S:	$[<11, 1>, <21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22, 6>]$	
Q:	$\exists \Diamond Next \neq 0 \wedge Next \neq 22$	F
It. 9: $<21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22, 6>$ is popped and extensions to $21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22$ searched		
S:	$[<11, 1>]$	
Q:	$\exists \Diamond Next \neq 0$	21
S:	$[<11, 1>, <21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22 \cdot 21, 7>]$	
Q:	$\exists \Diamond Next \neq 0 \wedge Next \neq 21$	F
It. 10: $<21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22 \cdot 21, 7>$ is popped and extensions to $21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22 \cdot 21$ searched		
S:	$[<11, 1>]$	
Q:	$\exists \Diamond Next \neq 0$ F	
$\rightarrow$ **Save** $21 \cdot 22 \cdot 11 \cdot 12 \cdot 21 \cdot 22 \cdot 21$		

4. Case Study: Execution Orders

We have conducted a case study to investigate the relation between testability and the execution environment when testing for timeliness. Schütz showed that testability is inherently low in dynamic, event-triggered real-time systems when compared with time-triggered systems [18]. Existing theoretical bounds for testability are based on parameters of the execution environment (e.g., potential preemptions) [18, 6]. This study evaluates this theory by modeling 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.

We approximate testability in dynamic, event-triggered systems by the number of execution orders. This is justified by noting that more variations with respect to execution orders means the results are less trustworthy. *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 due to elements that cannot be controlled in software (for example, varying efficiency of hardware acceleration components).

The study uses sequences of events (identified by type, time of occurrence, and possibly additional parameters) as test cases. For each test case selected, we count the number of potential execution orders. By varying the 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.

4.1. Approach

We model a real-time system as a timed automaton and use model checking to explore its behavior. We create a set of variant models by setting different values for selected execution environment parameters. The behavior of each variant model is then explored, enabling us to compare testability between the variants. It is therefore, essential that all execution orders are found by the model checker. UPPAAL is used to explore the system behavior.

The system being modeled is summarized in Figure 7. 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. The 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). The execution environment consists of an observer, a scheduler, and a resource handler. The observer observes events in the controlled environ-

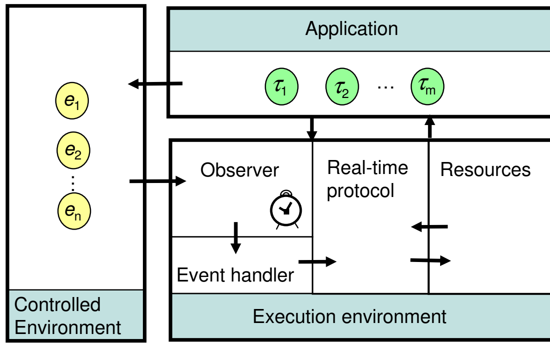


Figure 7. The modeled system

ment and communicates with the scheduler. Tasks are triggered on event observations and scheduled according to their deadlines in an earliest deadline first manner. Execution of tasks are conducted in non-preemptive intervals separated by designated preemption points. A p-point is passed each time a task is given access to the processor. In this specific case, the p-point is a preemption point and each p-path is an execution order.

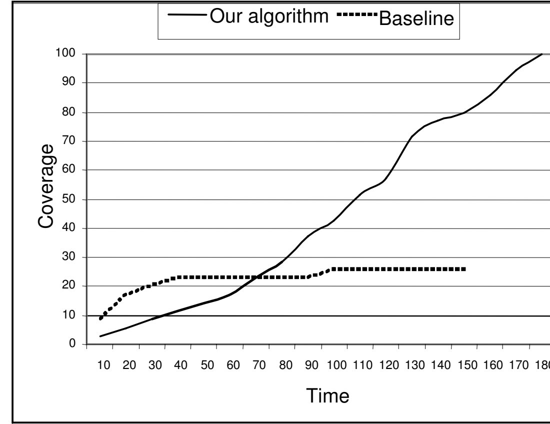
In order to evaluate the performance of our algorithm, we compared it with ordinary model checking where we ask the model checker to return complete p-paths, one at a time. This approach requires a defined final state S. The initial query asks the model checker if S is reachable ($\exists \Diamond S$). The resulting 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 $\exists \Diamond S \wedge \neg pp_1 \wedge \ldots \wedge \neg pp_n$ is falsified, meaning there are no more p-paths. This procedure is automated and we refer to it as the *base-line algorithm*.

The baseline approach only invoke the model checker once per p-path and it does not require the model to be altered during the search. The consequence is that the first p-paths are found rather quickly compared with our algorithm. However, the search is not guided and the model checker has to explore a larger part of the state-space for each additional p-path. The final invocation, which falsifies the query for an additional p-path, forces the model checker to explore the complete state-space. Hence, if the model is too complex for exhaustive search, then at some point in the search, the baseline algorithm will run out of memory. This leaves part of the state space unexplored and the set of p-paths found incomplete.

We applied our algorithm and the baseline algorithm to the same model[2]. The complete set of p-paths contained 35 different orders. Table 2 shows that our algorithm covered all p-paths in the model within 3 hours and the p-paths were found with a constant rate. The baseline algorithm delivered the first p-paths quickly but the performance dropped severely already after 20% coverage. It only managed to find nine p-paths, a coverage less than 26%. At that point the model checker stopped with an "out of memory" error message.

[2]The model included a total of 34 TA processes, eight of them modeling tasks annotated with p-points as shown in Figure 3. The experiment was performed on US-II sparc 12 CPU multiprocessor with 400MHz, 4Mb cache, and 6144Mb memory. The search order option in the model checker was set to depth-first.

Table 2. Percentage of p-paths covered by our algorithm and the baseline algorithm



4.2. Parallel Search for Traces

The trace-finding algorithm described in Section 3 split the search space into partitions along the p-paths. The algorithm finds one p-path at a time, using a stack to remember where to continue the search. However, it is important to note that, due to the tree structure and the fact that the search begins at the root node, each item on the stack contains enough information for an independent search for all continuations of that particular p-path prefix. Hence, this algorithm is particularly suitable for parallelization. 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, increasing efficiency by an order of magnitude. The limitation of this approach is, of course, the size of the partitions. Threading on a multiprocessor implies shared memory and might create a memory consumption problem. If this happens, the partitions can easily be distributed over separate processors. The cost for distribution in terms of communication is bounded by the num-

ber of partitions since there is only need for one message per p-path.

5. Discussion

Systematic testing of complex systems requires strategies that allow the tester to select test cases according to test criteria. Test criteria can be based on many aspects of the software, including the input domain, code structure, design models, or the ordering of events such as context switches or resource allocation. In a complex model, we cannot know these orders beforehand. Therefore, the algorithm presented in this paper collects them iteratively. We present an approach to ensure that the model checker finds all orders in which a specified set of transitions can be traversed. The approach is based on the idea of including the specification of current order, called p-path, in the model. Hence, at each invocation of the model checker, the state-space is limited to the current p-path. This approach mitigates the state-space explosion problem that is common in model checking.

The method uses hidden variables to avoid unfolding loops in the symbolic state-space of the analyzed model. Without this, states that are normally treated as equivalent or included in other states by a symbolic reachability analysis algorithm can become non-equivalent (or not be included). In such cases, the algorithm could theoretically slow down or not terminate. We overcome this problem by hiding the information of the progress along the p-path (the value of iterator i in the guiding automaton) in a hidden variable.

6. Conclusions

This paper presents a model checking method that generates a set of traces instead of individual traces. By specifying certain transitions in timed automata as p-points, the algorithm generates all potential orders with which these transitions can be traversed. This method is useful in testing, where sets of related traces are needed to satisfy coverage criteria. For example, if a test method focuses on execution orders, each transition that contains a context switch should be specified as a p-point. In this case, the set of traces generated would cover all execution orders. Other orders that might be interesting from a test perspective are communication and access to shared data.

Memory consumption is a major problem when verifying complex timed automata models. Our method dynamically divides the state space into smaller partitions along the traces as they are generated. These partitions are independent, so the memory needed by the traces is not increased. In addition, the trace generation can easily be distributed over several computers.

Our experience from using the method is good. We applied this method on a large case study with very promising results. The method enabled us to find **all** execution orders in a timed automaton model of a dynamic real-time system. Our method generated all traces, whereas ordinary model checking was only able to cover 26% of them.

Our ongoing work on this project will deepen this result. Important questions are ”how can the p-paths be useful in testing?” and ”can they help detect more faults?”. These are important questions to the practical relevance of the novel technique presented in this paper.

References

[1] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM’98)*, pages 46–54, December 1998.

[2] G. Behrmann, K. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Eleventh International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 341–353. Springer-Verlag, 1999.

[3] J. Bengtsson and W. Yi. Reducing memory usage in symbolic state-space exploration for timed systems. Technical report, Department of Information Technology, Uppsala University, 2001.

[4] J. Bengtsson and W. Yi. 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.

[5] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *In Lecture Notes on Concurrency and Petri Nets*, Lecture Notes in Computer Science vol 3098. Springer–Verlag, 2004.

[6] R. Birgisson, J. Mellin, and S. Andler. Bounds on Test Effort for Event-Triggered Real-Time Systems. In *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA’99)*, December 1999.

[7] E. M. Clarke and A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *In Logic of Programs: Workshop, Lecture Notes in Computer Science*, 131:52–71, 1981.

[8] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[9] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[10] E.M.Clarke, A.Cimatti, F.Giumchiglia, and M.Roveri. NuSMV: A new symbolic model checker. *Software Tools for Technology Transfer*, 2(4):401, 2000.

[11] A. Gargantini and C. L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC / SIGSOFT Foundations of Software Engineering*, pages 146–162, 1999.

[12] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[13] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[14] 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, Oct. 1997.

[15] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Space Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[16] R. Nilsson, S. Andler, and J. Mellin. Towards a Framework for Automated Testing of Transaction-Based Real-Time Systems. In *Proceedings of Eigth International Conference on Real-Time Computing Systems and Applications (RTCSA2002)*, pages 93–97, Tokyo, Japan, March 2002.

[17] J. P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR . In *Proc. 5*th *Int. Symp. on Programming*, number 137, pages 195–220, Berlin, 1982. Springer–Verlag.

[18] W. Schütz. *The Testability of Distributed Real-Time Systems*. Kluwer Academic Publishers, 1993.

[19] H. Thane and H. Hansson. Towards Deterministic Testing of Distributed Real-Time Systems. In *Swedish National Real-Time Conference (SNART'99)*, August 1999.

[20] W. Visser, C. Pasareanu, and S. Khurshid. Test input generation with Java pathfinder. In *International Symposium on Software Testing and Analysis*, pages 97–107, Boston, Massachusetts, USA, 2004. ACM Press.