A Global Algorithm for Model-Based Test Suite Generation

Anders Hessel[1] and Paul Pettersson[1,2]

[1] Department of Information Technology, Uppsala University, P.O. Box 337, SE-751 05 Uppsala, Sweden. E-mail: {hessel,paupet}@it.uu.se.
[2] Department of Computer Science and Electronics, Mälardalen University, P.O. Box 883, SE-721 23, Västeras, Sweden. E-mail: Paul.Pettersson@mdh.se.

Abstract. Model-based testing has been proposed as a technique to automatically verify that a system conforms to its specification. A popular approach is to use a model-checker to produce a set of test cases by formulating the test generation problem as a reachability problem. To guide the selection of test cases, a *coverage criterion* is often used. A coverage criterion can be seen as a set of items to be covered, called coverage items. We propose an on-the-fly algorithm that generates a test suite that covers all feasible coverage items. The algorithm returns a set of traces that includes a path fulfilling each item, without including redundant paths. The reachability algorithm explores a state only if it might increase the total coverage. The decision is global in the sense that it does not only regard each individual local search branch in isolation, but the *total coverage in all branches* together. For simpler coverage criteria as location of edge coverage, this implies that each model state is never explored twice.
The algorithm presented in this paper has been implemented in the test generation tool UPPAAL CO✓ER. We present encouraging results from applying the tool to a set of experiments and in an industrial sized case study.

1 Introduction

The bulk of verification efforts in software industry today is performed using various testing techniques. In conformance testing, the behavior of an implemented system, or system part, is checked to agree with its specification. This is typically done in a controlled environment where the system is executed and stimulated with input according to a test specification, and the responses of the system are checked to conform to its specification. To reduce the costs of this process, the execution of software testing is often automated, whereas the production of test suites are mostly done by hand. Techniques to automatically generate test suites, or to combine generation and execution, are emerging and getting more mature [31,9,28,19].

In this paper, we study techniques for model-based conformance testing in a setting where the test suite is automatically generated from a model before the actual testing takes place — sometimes referred to as *offline testing* in contrast to *online testing* [23]. In order to guide the generation of tests and to describe how thorough the tests should be, we select tests following a particular coverage criterion, such as coverage of control states or edges in a model. Many coverage criteria have been suggested in the

literature [27,6,12] ranging from simple structural criteria to complex data-flow criteria characterized as path properties. Many algorithms for generating test suites following a given coverage criterion have also been proposed [29,22,18,13], including algorithms producing test suites optimal in the number of test cases, in the total length of the test suite, or in the total time required to execute the test suite.

In this paper, we study test suite generation algorithms inspired by reachability analysis techniques used in model-checkers such as SPIN [16] and UPPAAL [24] — an approach shared with, e.g., [19]. Such algorithms essentially perform reachability analysis to generate and explore the state space of a model in order to find a set of paths that follows a given coverage criterion, which can be interpreted as a test suite. To generate a path, a coverage criterion can be regarded as a set of independent *coverage tasks* [11] or *coverage items* [4] to be covered. Reachability analysis is applied to generate a set of paths for all reachable coverage items. We review this technique and suggest a number of modifications to improve the efficiency of the analysis.

The main contribution of this paper is a novel on-the-fly algorithm for generating test suites by reachability analysis. It can be seen as a trade-off between performance of the algorithm, in terms of time and space requirements, and generating a test suite with reasonable characteristics. The result is an algorithm that in each step uses global information about the state space generated so far to guide the further analysis and to speed up termination. The generated test suite is reasonable in the sense that each coverage item is reached by a path from the initial state to the first found state in which it is satisfied.

During the state-space exploration, the algorithm stores a set of paths to the coverage items satisfied so far. This information is used to prune search branches that will not be able to contribute to the total coverage — a technique that improves the performance of the algorithm. In experiments we justify this statement by presenting how the algorithm, implemented in the UPPAAL CO✓ER tool [3], performs on a set of examples from the literature.

The rest of the paper is organized as follows: in Section 2 we describe the model used in this paper, and review techniques for test case generation based on reachability analysis. In Section 3 we describe a reachability analysis algorithm for test case generation. In Section 4 we present a novel algorithm for test case generation that uses global information about the generated state-space to determine termination and pruning. In Section 5 we describe the results of experiments comparing the different techniques. The paper ends with conclusions in Section 6.

Related Work: Our work is mostly related to test case generation approaches inspired by model-checking techniques, including [5,13,23,19,17,28].

In [28], Nielsen and Skou generate test cases that cover symbolic states of Event Recording Automata. Like our work, the proposed state-space exploration algorithm is inspired by model-checking, however the work is focused on timed system and uses a fixed coverage criterion.

[3] See the web page `http://user.it.uu.se/~hessel/CoVer/` for more information about the UPPAAL CO✓ER tool.

In [19], Hong et al show how several flow-based coverage criteria can be expressed in temporal logic and how the test case generation problem can be solved by model-checking. Hong and Ural [17] continue this work and study how coverage items can subsume each other, and propose a solution to the problem. These works use an existing CTL model-checker to solve the test case generation problem, whereas we propose a specialized algorithm for test case generation.

Our work is also related to directed model-checking techniques, where state-space exploration is guided by the property to be checked. In [8], the authors use a bitstate hashing based iterated search refinement method to guide a model-checker to generate test cases. This method can be seen as a meta algorithm using an existing model-checker iteratively. Thus the actual model-checking algorithms is not refined for test case generation.

2 Preliminaries

We will present ideas and algorithms for test case generation applicable to several automata based models, such as finite state machines, extended finite state machines (EFSM) as, e.g., SDL [20], or timed automata [1]. Throughout this paper, we shall present our results using the model of communicating EFSMs.

2.1 The Model

An EFSM F over actions Act is a tuple $\langle L, l_0, V, E\rangle$, where L is a set of locations, $l_0 \in L$ the initial location, V is a finite set of variables with finite value domains, and E is a set of edges. An edge is of the form $\langle l, g, \alpha, u, l'\rangle \in E$, where $l \in L$ is the *source location* and $l' \in L$ the *destination location*, g is a *guard* (a predicate) over V, $\alpha \in Act$ an action, and u is an *update* in the form of an assignment of variables in V to expressions over V.

A *state* of an EFSM is a tuple $\langle l, \sigma\rangle$ where $l \in L$ and σ is a mapping from V to values. The initial state is $\langle l_0, \sigma_0\rangle$ where σ_0 is the initial mapping. A transition is of the form $\langle l, \sigma\rangle \xrightarrow{\alpha} \langle l', \sigma'\rangle$ and is possible if there is an edge $\langle l, g, \alpha, u, l'\rangle \in E$ where the g is satisfied for the valuation σ, the result of updating σ according to u is σ', and α is an action.

A *network of communicating EFSMs* (CEFSM) over Act is a parallel composition of a finite set of EFSMs $F_1 \ldots, F_n$ for a given synchronization function. A state in the network is a tuple of the form $\langle\langle l_1, \sigma_1\rangle, \ldots, \langle l_n, \sigma_n\rangle\rangle$, where $\langle l_i, \sigma_i\rangle$ is a state of F_i. We assume a hand-shaking synchronization function similar to that of CCS [26]. A transition of a CEFSM is then either (i) an internal transition of one EFSM, i.e., $\langle\langle l_1, \sigma_1\rangle, ...\langle l_k, \sigma_k\rangle, ..., \langle l_n, \sigma_n\rangle\rangle \overset{\tau}{\rightsquigarrow} \langle\langle l_1, \sigma_1\rangle, ..., \langle l'_k, \sigma'_k\rangle, ..., \langle l_n, \sigma_n\rangle\rangle$ if $\langle l_k, \sigma_k\rangle \xrightarrow{\tau} \langle l'_k, \sigma'_k\rangle$ or (ii) synchronization of EFSMs, i.e., $\langle\langle l_1, \sigma_1\rangle, ..., \langle l_k, \sigma_k\rangle, ..., \langle l_m, \sigma_m\rangle, ..., \langle l_n, \sigma_n\rangle\rangle \overset{\alpha}{\rightsquigarrow} \langle\langle l_1, \sigma_1\rangle, ..., \langle l'_k, \sigma'_k\rangle, ..., \langle l'_m \sigma'_m\rangle, ..., \langle l_n, \sigma_n\rangle\rangle$ if $\langle l_k, \sigma_k\rangle \xrightarrow{\alpha?} \langle l'_k, \sigma'_k\rangle$, $\langle l_m, \sigma_m\rangle \xrightarrow{\alpha!} \langle l'_m, \sigma'_m\rangle$, and $\alpha?$ and $\alpha!$ are complementary synchronization actions.

Wherever it is clear from the context, we will use term *model state* denoted s to refer to a state of a CEFSM and the term *model transitions* denoted $s \overset{\alpha}{\rightsquigarrow} s'$ or t for a CEFSM transition.

2.2 Test Case Generation

We will focus the presentation on generating test suites with a certain coverage in a CEFSM. Coverage criteria are often used by testing engineers to specify how thorough a test suite should test a system under test. Examples of coverage criteria used in model-based testing include structural criteria such as location or edge coverage, data-flow properties such as definition-use pair coverage, and semantic coverage on, e.g., states of an EFSM or the time regions of a timed automata [28,30]. A coverage criterion typically consists of a list of items to be covered or reached. We shall call those items *coverage items*, and use C to denote a set of coverage items, C_0 the initial C, and $|C|$ to denote the size of C.

If the coverage criterion stipulates a path property of the kind used in, e.g., data flow criteria as definition-use pairs, we need to handle information about partially satisfied coverage items. We use the definition-use pair coverage criterion [10] to illustrate the concept. It should cover all paths where a given variable x is first *defined* in an EFSM edge e_d active in a transition t_d, and later *used* in (usually) another EFSM edge e_u active in a transition t_u, without any redefinitions of x along the path from t_d to t_u. We shall store such *partial coverage item*, i.e., that x was defined on the EFSM edge e_d active in t_d, in a set denoted A.

In the algorithms, we shall extend the CEFSM state s to (s, C, A) or (s, C) when A is not needed. We further extend the model transition relation to transitions of the form $(s, A, C) \stackrel{t}{\Rightarrow}_c (s', A', C')$ where t is a model transition $s \stackrel{\alpha}{\rightsquigarrow} s'$, C' and A' are the result of updating C and A according to the coverage achieved in transition $s \stackrel{\alpha}{\rightsquigarrow} s'$. For a detailed description of how A' and C' are updated, see e.g, [14].

We shall use traces to represent test cases generated from models. We use ϵ to denote the empty trace, and $\omega.t$ to denote the trace ω extended with transition t. Further, we use $|\omega|$ to denote the length of ω, defined as $|\epsilon| = 0$ and $|\omega.t| = |\omega| + 1$.

3 A Model Checking Approach to Test Suite Generation

3.1 A Local Algorithm

The problem of generating a test suite for a given coverage criteria by reachability analysis has been studied in many settings in the literature, see e.g., [25,19,13,4]. The authors of this paper suggest an algorithm for minimal test suite generation from models of real-time systems described as networks of timed automata in [13] and for untimed systems modeled as extended finite state machines in [4]. A version of these algorithms is shown in Figure 1, but modified so that it returns a shortest path (in the number of steps) with maximum coverage, if the algorithm is executed in a breadth-first manner.

The algorithm is essentially an ordinary reachability analysis algorithm that uses two data structures WAIT and PASS to hold states waiting to be examined and states already examined, respectively. In addition, the global integer variable max is used to (invariantly) store the maximum coverage witnessed so far, and the variable ω^{max} stores a path reaching a state with maximum coverage. Initially PASS is empty and WAIT holds the initial combined state of the form (s_0, C_0, ϵ), where s_0 is the initial state of the model, C_0 is the coverage of s_0, and ϵ is the empty path.

```
(01) PASS:= ∅ ; WAIT:= {(s_0, C_0, ε)} ; ω^max := ε ; max := |C_0|
(02) while WAIT≠ ∅ do
(03)    select (s, C, ω) from WAIT; add (s, C, ω) to PASS
(04)    for all (s', C', ω.t) : (s, C, ω) ⇒^t_c (s', C', ω.t) do
(05)       if |C'| > max then
(06)          ω^max := ω.t ; max := |C'|
(07)       if ¬∃(s_i, C_i, ω_i) : (s_i, C_i, ω_i) ∈ PASS ∪ WAIT ∧ s_i = s' ∧ C' = C_i then
(08)          add (s', C', ω.t) to WAIT
(09)    od
(10) od
(11) return ω^max
```

Fig. 1. A reachability analysis algorithm for test suite generation.

The lines (03) to (08) are repeated until WAIT is empty. Alternatively, if the maximal number of coverage items is known on beforehand, the loop can terminate when the coverage is reached. At line (03) a state is taken from WAIT, and at line (04) the successors of the state are generated. At line (05) and (06) a new path is saved and a new maximum coverage is saved if the current successor covers more items than the previous maxima. The successor state $(s', C', \omega.t)$ is put on WAIT if there is no state with the same model state and the same set of covered items, i.e., no state (s_i, C_i, ω_i) with $s_i = s'$ and $C_i = C'$ can be found in WAIT nor PASS.

It can be shown (see e.g., [13,4]) that the algorithm of Figure 1 returns a shortest path with maximum coverage if the select in line (03) is done so that the algorithm explores the state space in breadth-first order.

Resets: Note that the algorithm of Figure 1 may return a trace ω^{max} that does not include all feasible coverage items. This can happen if there are two states s_i and s_j in the state space of the model, such that s_i cannot reach s_j or the other way around. We can avoid this problem by adding a state $(s_0, C, \omega.reset)$ to every successor set at line (04), where $reset$ is a distinct symbol representing that the model restarts from its initial state. This guarantees that the algorithm in Figure 1 will always return a path with all feasible coverage.

Coverage subsumption: A first improvement of the algorithm, described in [14] and in analogy with the inclusion abstraction described in [7], is to change line (07) so that instead of requiring equality of the coverage items $C_i = C'$, inclusion of coverage items is used, i.e., $C' \subseteq C_i$. The algorithm will now prune an extended state (and thus the part of the state space it may reach) if there exists an extended state with the same model state and a (non-strict) superset of its coverage.

It is also possible to further improve the algorithm in the dual case, i.e., if a state (s', C', ω') is put on WAIT, such that states (s_i, C_i, ω_i) exist in WAIT or PASS with $s' = s_i$ and $C' \supset C_i$. In this case, all states (s_i, C_i, ω_i) can be removed from WAIT and PASS. Note that, as a consequence some states put on WAIT will never be further explored. Instead subsuming states will be explored. This in turn may change the order in which

states are searched. The same technique has successfully been used to speed up model-checking tools such as UPPAAL [2]. The result is an algorithm that explores fewer states, but ordinary breadth-first search is no longer guaranteed to produce a shortest trace.

3.2 Partial Coverage

The algorithm in Figure 1 is applicable to coverage items (i.e., criteria) that can be determined locally in a single model transition, such as the *location* or *edge* coverage criteria. If the coverage criterion stipulates a path property of the kind used in e.g, data-flow criteria as definition-use pairs, the algorithm must be adjusted to handle information about partial coverage items.

Algorithms inspired by model-checking for this class of coverage criteria have been proposed in, e.g., [13,14,4,19]. To modify the algorithm of Figure 1 amounts to storing the partial coverage items in the structure C, together with the ordinary coverage items, and modify the behavior of the operator $|C|$ (used at line `(06)`) so that partial coverage items are not considered. That is, partial coverage is represented and stored in the same way as ordinary coverage, but they are not considered when the number of (fully satisfied) coverage items is computed.

We also note that the coverage subsumption discussed above is not affected by the existence of partial coverage items in C. The reset must also be done on the partial coverage items, i.e., $(s_0, A_0, C, \omega.reset)$ is added at successor generation.

4 A Global Algorithm for Test Suite Generation

A well-known problem with algorithms like the one described in the previous section is the time consumed to explore the state space, and the space required to represent WAIT and PASS. The algorithm in Figure 1 explores states of the form (s, C, ω), resulting in a state space with size defined by the number of model states s in product with the number of possible coverage sets C (the trace ω does not influence the number of states in the state space).

In this section, we describe algorithms that avoid exploring all states of the form (s, C, ω) and still generates a reasonable test suite. The idea is to collect and combine information from the whole generated state space so that each model state s is not explored more often than necessary. In particular, we store a set COV of all distinct coverage items covered so far, i.e., COV $= \bigcup_i C_i$ for all explored states (s_i, C_i, ω_i). Additional information, including a trace to each coverage item $c \in$ COV is stored in a structure SUITE, that is used to generate the test suite returned by the algorithm. We first describe an algorithm for coverage criteria without partial coverage items in Section 4.1, followed by an algorithm handling partial coverage in Section 4.2.

4.1 A Global Algorithm

The algorithm shown in Figure 2 is a modified version of the algorithm in Figure 1. It works in a similar way, but collects coverage from all explored states (i.e., branches) in the variables COV and SUITE. The variable COV holds the total set of coverage

```
(01) PASS:= ∅ ; WAIT:= {(s0, C0, ε)} ; SUITE:= ∅ ; COV:= C0
(02) while WAIT≠ ∅ do
(03)   select (s, C, ω) from WAIT; add (s, C, ω) to PASS
(04)   for all (s', C', ω.t) : (s, C, ω) ⇒^t_c (s', C', ω.t) do
(05)     if C' ⊈ COV then
(06)       add (ω.t, C') to SUITE; COV := COV ∪ C'
(07)     if ¬∃(si, Ci, ωi) : (si, Ci, ωi) ∈ PASS ∪ WAIT ∧ si = s' then
(08)       add (s', C', ω.t) to WAIT
(09)   od
(10) od
(11) return SUITE
```

Fig. 2. A global coverage algorithm for test suite generation.

items found by the algorithm, i.e., COV $= \bigcup_i C_i$ for all explored states (s_i, C_i, ω_i). For every explored state with new coverage, a tuple (ω_i, C_i) is added to the set SUITE. This makes SUITE a set of tuples with one trace ω to each coverage item in COV. With ordinary breadth-first search strategy, SUITE will hold a trace to coverage item with the minimum number of model transitions. The additional information stored in SUITE will be used to improve the algorithm and the test suite, later in this section.

The loop of the algorithm in Figure 2 has two differences from the algorithm in Figure 1. In lines `(05)` and `(06)` the variables COV and SUITE are updated if the explored state contains new coverage items that the algorithm has not seen before. Note that in line `(07)`, we do not consider the coverage C' of the generated states. As a result, a state $(s', C', \omega.t)$ is not further explored (i.e., added to WAIT) if the model state s' has been previously explored. The algorithm terminates when WAIT is empty which is the case when all reachable states from s_0 have been explored. At this point, each model state has been explored only once, COV contains all reachable coverage items, and SUITE includes at least one trace to each coverage item in COV. We shall return to the problem of generating a test suite from COV in Section 4.3.

4.2 Partial Coverage

We now describe how to modify the algorithms above so that it can be used for coverage criteria that requires partial coverage items (in analogy with the modified algorithm presented in Section 3.2). Recall that partial coverage items are needed when the coverage criteria requires path properties to be covered, like in the definition-use pair criterion (see Section 2.2).

The modified algorithm is shown in Figure 3. It operates on extended states of the form (s, A, C, ω), where C and A are the *coverage items* and the *partial coverage items* respectively, collected on the path ω reaching s. The only principal difference compared to the algorithm of Figure 2 is on line `(07)` where the most recently generated state $(s', A', C', \omega.t)$ is examined. Here, the state is not further explored if an already explored state $(s_i, A_i, C_i, \omega_i)$ with $s_i = s'$ and $A' \subseteq A_i$ exists in PASS or WAIT. If this is the case, it can be deduced that further exploration of $(s', A', C', \omega.t)$ is not needed,

```
(01) PASS:= ∅ ; WAIT:= {(s0, A0, C0, ϵ)} ; SUITE:= ∅ ; COV := C0
(02) while WAIT≠ ∅ do
(03)    select (s, A, C, ω) from WAIT; add (s, A, C, ω) to PASS
(04)    for all (s', A', C', ω.t) : (s, A, C, ω) ⇒^t_c (s', A', C', ω.t) do
(05)       if C' ⊈ COV then
(06)          add (ω.t, C') to SUITE; COV := COV ∪ C'
(07)       if ¬∃(si, Ai, Ci, ωi) : (si, Ai, Ci, ωi) ∈ PASS ∪ WAIT ∧ si = s' ∧ A' ⊆ Ai then
(08)          add (s', A', C', ω.t) to WAIT
(09)    od
(10) od
(11) return SUITE
```

Fig. 3. A global algorithm with partial coverage items.

since the state is not able to contribute coverage items other than those that further exploration of $(s_i, A_i, C_i, \omega_i)$ will yield.

The algorithm of Figure 3 terminates when WAIT it empty. At this point, all reachable model states from s_0 have been explored, COV contains all reachable coverage items, and SUITE is a set of pairs of the form (ω_i, C_i), where ω_i is a trace ending in a state with coverage C_i, and $\bigcup_i C_i = \text{COV}$. It is easy to prove that the algorithm is sound. It is also complete since for each reachable partial coverage item a_i, an extended state (s, A, C, ω), such that $a_i \in A$, has been explored. This guarantees that all feasible coverage items will be generated since item $c_i \in \text{COV}$ depends only on one (or zero) partial coverage items in A.

4.3 Improving the Test Suite

When the algorithm above terminates, SUITE is a set $\{(\omega_0, C_0), \ldots, (\omega_{n-1}, C_{n-1})\}$. Ideally, this set should be reduced so that the total coverage $\bigcup_i C_i$ is not changed, and the length of the test suite, i.e., $\sum_i |\omega_i|$, is minimized. The remaining traces ω_i can then be used as the test suite. However, selecting a subset of traces with this property is a version the well-known *set covering problem* which is NP-hard [21].

The likewise well-known technique of *prefix elimination* can be used as an approximative solution to the problem, i.e., to make sure that there are no pair of traces ω, ω' in SUITE such that ω is a prefix of ω' of the other way around. However, this approach has some obvious problems, including the fact that SUITE could still include redundant traces, i.e. traces that can be removed without reducing the total coverage of SUITE.

We have chosen an algorithm that can be performed incrementally, as part of the main test case generation algorithm. It can also be applied to SUITE when the main algorithm has terminated. It checks that each (ω_j, C_j) in SUITE satisfies the condition $C_j \not\subseteq C_0 \cup \ldots \cup C_{j-1} \cup C_{j+1} \cup \ldots \cup C_{n-1}$, i.e., (ω_j, C_j) contributes to the total coverage of SUITE. As we shall see in the next section, this approach has worked well in our experiments.

Table 3. Time (in seconds) and space (in MB) performance of the algorithms.

		Local Algorithm					Global Algorithm				
		time	**mem**	**len**	**states**	**tr**	**time**	**mem**	**len**	**states**	**tr**
Train e 6 (0)	full	2.9	9.6	15	3353	1	1.25	9.3	15	1645	1
	reset	3.85	10.5	15	7375	1	-	-	-	-	-
Train du 12 (5)	full	37	14	47	27129	1	2.4	9.3	l56	1717	4
	reset	107	33	47	114697	1	-	-	-	-	-
Philips du 109 (42)	stop 30	9	10	30	5797	1	1	9.2	80	170	8
	stop 60	1085	87.7	69	461305	1	2.6	9.2	204	393	16
	full	-	-	-	-	-	16.4	10.6	681	7579	37
WAP e 68 (0)	stop 20	4.29	12.7	55	5105	1	4.17	12.5	161	4361	5
	stop 30	30.8	31.2	86	35615	1	4.51	12.5	175	4395	5
	stop 68	-	-	-	(7348332)	0	38	32	818	37503	12
	full	-	-	-	-	-	3871	1946	818	3850877	12
WAP ei 68 (68)	stop 20	10.85	12.7	58	5404	1	8	12	58	5299	1
	stop 30	525	52	93	71328	1	15.9	14.9	189	7291	3
	stop 68	-	-	-	-	-	280	111.2	1718	200011	19
	full	-	-	-	-	-	4093	2025	1718	3937243	19

5 Experiments

For the experiments in this section we use our tool UPPAAL CO✓ER that takes as input a timed automata model and a coverage criterion.

Models and Coverage Criteria: We will use three models that are documented in the literature: a train gate example (Train) where four trains and a controller are modeled [32], a audio-control protocol with bus collision detection by Philips (Philips) [3], and a WAP stack (WAP) modeled and tested in [15].

We present experiments of three different coverage criteria edge (e), definition-use pair (du), and edge init (ei). In the *edge* coverage criterion a coverage item is a traversed edge in an automaton. If several instances of the same automaton are used, it does not distinguish between different instances exercising the edge. The du-pair criterion is described in Section 2.2 of this paper. The *edge init* coverage criterion requires an edge to be exercised, as in the edge coverage criterion. For the coverage item to be satisfied, the test case must also put the system back to a state that is similar to the initial system state.

Results: In Table 3 the performance of the local and global algorithms is presented. The algorithms were executed using breadth-first search strategy on a SUN Ultra SPARC-II 450MHz.

The leftmost column of the table specifies the input used in the experiments, i.e., the model, the coverage criterion, the number of coverage items, and (in parentheses) the number of partial coverage items existing in the model. In the second column the following keywords are used: *full* for exhaustive search, *stop* x for termination after x

found coverage items, and *reset* if resets are used in the model (as described in Section 3.1 this is applicable only in the local algorithm). For both the local and global algorithm we give the following numbers/columns, generation time (**time**), memory (**mem**), length of the trace (**len**), number of states generated (**states**), and number of traces generated (**tr**).

The rows marked *Train e 6 (0)* show performance for the train gate model with the edge coverage criterion on the instances of the train automaton. There are six coverage items to be covered, with zero partial coverage items. The global algorithm generates 1645 states which is the size of the model. The local algorithm generates 3353 or 7375 states without and with resets, respectively.

For the rows *Train du 12 (5)* the definition-use criterion has been used. There are 12 different coverage criteria and five partial coverage items. The size of the generated state space of the global algorithm increases (due to the partial coverage items) modestly to 1717 ($+4.3\%$ compared with the actual size of the model state space). For the local algorithm this increase is from 3357 states to 27129 (or 114697 when resets are used). We note that the global algorithm performs substantially better than the local algorithms. In fact, it generates only 6% (or 2%) of the states used by the local algorithm(s). The gain in execution time is similar.

For the models in the rest of the table we have not been able to run exhaustive analysis with the local algorithm, nor have been able to use resets. Still the experimental results show how the algorithms scale up for different models and coverage criteria. In all the examples, the global algorithms outperforms the local algorithm.

6 Conclusion

In this paper, we have studied algorithms and ideas for test suite generation applicable to automata models with semantics defined as a (finite) transition system. We have reviewed algorithms derived from ordinary reachability analysis algorithm, similar to those used in many model-checking and planning tools. Such algorithms can be used to generate test suites that follow a given coverage criterion and are optimal in e.g, the total number of model transitions the test suite will exercise. We have further elaborated these algorithms by adopting existing abstraction and pruning techniques often used in model-checking algorithms.

The main contribution of this paper, is a novel global algorithm for model-based generation of test suites following a given coverage criteria. At any given point, the algorithm — which is inspired by the priorly described reachability analysis algorithms — uses knowledge about the total coverage found in the currently generated state space to guide and prune the remaining exploration. In this way, the algorithm avoids unnecessary exploration and generates a test suite with reasonable characteristics.

All algorithms presented in this paper have been implemented in our test case generation tool UPPAAL CO✓ER. To compare and evaluate the algorithms, we have performed a number of experiments on a set of models previously described in the literature. In particular, the evaluation gives experimental evidence that the suggested global algorithm uses substantially less memory and time than local algorithms, and outputs test suites

that are not far from optimal. In this respect, the suggested global algorithm increases the maximum size of models for which test suites can be algorithmically generated.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. G. Behrmann, J. Bengtsson, A. David, K. G. Larsen, P. Pettersson, and W. Yi. Uppaal implementation secrets. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 3–22, London, UK, 2002. Springer-Verlag.
3. J. Bengtsson, W.O. David Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In R. Alur and T. A. Henzinger, editors, *Proc. 8^{th} Int. Conf. on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 244–256. Springer–Verlag, July 1996.
4. J. Blom, A. Hessel, B. Jonsson, and P. Pettersson. Specifying and generating test cases using observer automata. In J. Gabowski and B. Nielsen, editors, *Proc. 4^{th} International Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 125–139. Springer–Verlag, 2005.
5. R. Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. *Formal Aspects of Computing*, 12(5):350–371, 2000.
6. L. A. Clarke, A. Podgurski, D. J. Richardsson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. on Software Engineering*, SE-15(11):1318–1332, November 1989.
7. Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In Bernard Steffen, editor, *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 313–329. Springer–Verlag, 1998.
8. J.P. Ernits, A. Kull, K. Raiend, and J. Vain. Generating tests from efsm models using guided model checking and iterated search refinement. In K. Havelund, M. Nez, G. Rosu, and B. Wolff, editors, *FATES/RV 2006*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer–Verlag, 2006.
9. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29, 1997.
10. P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498, 1988.
11. G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 134–143, 2002.
12. P.M. Herman. A data flow analysis approach to program testing. *Australian Computer J.*, 8(3), 1976.
13. A. Hessel, K. G. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-Optimal Real-Time Test Case Generation using UPPAAL. In A. Petrenko and A. Ulrich, editors, *Proc. 3^{rd} International Workshop on Formal Approaches to Testing of Software 2003 (FATES'03)*, volume 2931 of *Lecture Notes in Computer Science*, pages 136–151. Springer–Verlag, 2004.
14. A. Hessel and P. Pettersson. A test generation algorithm for real-time systems. In H-D. Ehrich and K-D. Schewe, editors, *Proc. of 4^{th} Int. Conf. on Quality Software*, pages 268–273. IEEE Computer Society Press, September 2004.

15. A. Hessel and P. Pettersson. Model-Based Testing of a WAP Gateway: an Industrial Study. In L. Brim and M. Leucker, editors, *Proc.* 11^{th} *International Workshop on Formal Methods for Industrial Critical Systems (FMICS'06)*, Lecture Notes in Computer Science. Springer–Verlag, 2006.
16. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.
17. H. S. Hong and S. Ural. Using model checking for reducing the cost of test generation. In J. Gabowski and B. Nielsen, editors, *Proc.* 4^{th} *International Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 110–124. Springer–Verlag, 2005.
18. H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE'03:* 25^{th} *Int. Conf. on Software Enginering*, pages 232–242, May 2003.
19. H.S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems :* 8^{th} *International Conference, (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer–Verlag, 2002.
20. ITU, Geneva. *ITU-T, Z.100, Specification and Description Language (SDL)*, November 1999.
21. R. M. Karp. Reducibility among combinatorial problems. In Thomas J., editor, *Complexity of Computer Computations, Proc. Sympos.*, pages 85–103. IBM, 1972.
22. M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In Laurent Mounier Susanne Graf, editor, *Proc.* 11^{th} *International SPIN Workshop on Model Checking Software (SPIN'04)*, volume 2989 of *Lecture Notes in Computer Science*, pages 182–197. Springer–Verlag, 2004.
23. K. G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using uppaal. In J. Gabowski and B. Nielsen, editors, *Proc.* 4^{th} *International Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer–Verlag, 2005.
24. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.
25. D. Lee and M. Yannakakis. Optimization problems from feature testing of communication protocols. *icnp*, 00:66, 1996.
26. R. Milner. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, 1989.
27. G. Myers. *The Art of Software Testing*. Wiley-Interscience, 1979.
28. Brian Nielsen and Arne Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer*, 5:59–77, 2003.
29. S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985.
30. J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing Timed Automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.
31. J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99:* 7^{th} *European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.
32. W. Yi, P. Pettersson, and M. Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In Dieter Hogrefe and Stefan Leue, editors, *Proc. of the* 7^{th} *Int. Conf. on Formal Description Techniques (FORTE'04)*, pages 223–238. North–Holland, 1994.