International Journal of Software Engineering
and Knowledge Engineering
Vol. 18, No. 6 (2008) 803–822

World Scientific
www.worldscientific.com

REACHABILITY GRAPH-BASED TEST SEQUENCE GENERATION FOR CONCURRENT PROGRAMS

W. ERIC WONG

Department of Computer Science, University of Texas at Dallas,
Richardson, TX 75083, USA
ewong@utdallas.edu

YU LEI

Department of Computer Science and Engineering,
University of Texas at Arlington, Arlington, TX 76019, USA
ylei@cse.uta.edu

Received 9 May 2005
Revised 10 January 2007
Accepted 5 June 2007

One common approach to test sequence generation for structurally testing concurrent programs involves constructing a reachability graph (RG) and selecting a set of paths from the graph to satisfy some coverage criterion. It is often suggested that test sequence generation methods for testing sequential programs based on a control flow graph (CFG) can also be used to select paths from an RG for testing concurrent programs. However, there is a major difference between these two, as the former suffers from a feasibility problem (i.e., some paths in a CFG may not be feasible at run-time) and the latter does not. As a result, even though test sequence generation methods for sequential programs can be applied to concurrent programs, they may not be efficient. We propose four methods — two based on hot spot prioritization and two based on topological sort — to effectively generate a small set of test sequences that covers all the nodes in an RG. The same methods are also applied to the corresponding dual graph for generating test sequences to cover all the edges. A case study was conducted to demonstrate the use of our methods.

Keywords: Reachability graph; structural testing; concurrent program testing; all-node criterion; all-edge criterion.

1. Introduction

A concurrent program consists of a set of cooperative threads that can execute in parallel. To accomplish a common goal, these threads synchronize and communicate with each other by accessing shared variables and/or by exchanging messages [26]. Because of the existence of race conditions, e.g., variations in thread scheduling and message latencies, the behavior of a concurrent program is inherently non-

deterministic. Multiple executions of a concurrent program with a fixed input may exercise different sequences of synchronization events and may even produce different results [1, 2]. This nondeterministic behavior makes it notoriously difficult to test concurrent programs. Note that a synchronization event is an event that represents the execution of a synchronization operation, i.e., an operation that is performed on a synchronization and/or communication object. Examples of synchronization operations include send and receive operations on communication channels, P and V operations on semaphores, lock and unlock operations on locks, etc. Hereafter, we refer to a sequence of synchronization events as a SYN-sequence. The notion of a SYN-sequence has been defined for many different concurrent programming languages and constructs [1], and has been used for the specification and testing of concurrent programs [3, 12, 15, 27].

One simple approach to dealing with the nondeterministic behavior is to execute a program with a fixed input many times in the hope that faults will be exposed by one of these executions [6,17]. This type of testing, called nondeterministic testing, is easy to carry out, but can be very inefficient. It is often the case that only a small set of distinct SYN-sequences is actually exercised during nondeterministic testing, which results in very limited test coverage. An alternative approach is to generate a set of SYN-sequences and then force them to be exercised during test runs [2, 12]. This type of testing, called deterministic testing, often generates SYN-sequences to satisfy certain well-defined criteria [25, 30, 31, 33] and thus allows one to discover more subtle bugs. Moreover, testing in a deterministic manner makes it possible to reproduce a failed test run, which is a significant advantage for subsequent debugging activities.

The main challenge of deterministic testing is how to generate a good set of SYN-sequences. One common approach to SYN-sequence generation involves constructing the reachability graph (RG) of a concurrent program, typically from its design. An RG is a graph in which every node represents a reachable state, and every edge represents a transition between two reachable states. An RG can be constructed using a technique called reachability analysis, which starts from an initial state and then repeatedly derives the successor states of the states that are encountered [12, 16, 21, 28]. Reachability analysis ensures that each path in an RG corresponds to a feasible SYN-sequence, i.e., a SYN-sequence that can be exercised by at least one program execution. Let P be a concurrent program. Structural testing of P involves generating a set of paths from an RG of P to satisfy some well-defined structural testing criteria such as the all-node or all-edge criterion [25]. Each SYN-sequence, along with its data input, is then fed to a deterministic execution environment for actual test execution. Note that such data input can be derived using other techniques like domain partitioning and/or constraint solving [19]. Also note that if a SYN-sequence is generated at a certain level of abstraction, the SYN-sequence needs to be mapped to a concrete SYN-sequence before it is used for deterministic execution [3, 12]. The issues of how to generate data input and how to map an abstract SYN-sequence to a concrete one are not part of our study.

In the remainder of this paper, we use the terms SYN-sequence, test path, and test sequence interchangeably.

Although test sequence generation for testing sequential programs has been studied, little attention has been paid to test sequence generation for testing concurrent programs. It is often suggested that control flow graph (CFG)-based test sequence generation methods for sequential programs can also be used to select paths from an RG for testing concurrent programs [12]. This is because from an abstract point of view, both types of test sequence generation deal with a similar problem of how to select a set of paths from a graph to satisfy some structural coverage criterion. However, a CFG often contains infeasible paths, i.e., paths that cannot be actually exercised at run-time. As a result, CFG-based test generation methods are often tailored to deal with the presence of infeasible paths. In contrast, every path in an RG must be feasible,[a] which makes special handling for infeasible paths no longer necessary. Moreover, since the size of an RG can be much larger than a CFG, the requirement on the run-time complexity for test generation methods based on an RG is more stringent than those based on a CFG. Therefore, it is necessary to remove the special handling for infeasible paths, which can be expensive and inefficient, from a CFG-based method so that its applicability to an RG can be increased. Finally, in order to reduce the effort and costs spent on test execution, it is important to reduce the number of test sequences being generated because fewer tests in general are also likely to require less test cost. That is, we need methods which can generate test sequences to increase the coverage in an effective way. To the best of our knowledge, none of the CFG-based test generation methods has a focus on this.[b]

In this paper, we present four different test generation methods, two based on hot spot prioritization and two based on topological sort, to generate a small set of test sequences that cover all the nodes in an RG. The same methods are also applied to the corresponding dual graph for generating test sequences to cover all the edges. A case study on graphs for five commonly used distributed algorithms was conducted. Our results indicate that for each graph our methods only need to generate a small set of test sequences to satisfy the all-node or all-edge criterion. Moreover, the first few test sequences so generated can increase the coverage in a much more effective way than the rest of the test sequences.

The rest of the paper is organized as follows. Section 2 describes our methods for selecting test sequences to cover all the nodes or edges in an RG. Section 3 reports a case study using the methods proposed in this paper. Observations based

[a]If a path in an RG is infeasible, the subject program must be incorrect. In this case, a fault has been detected. This is in contrast with a CFG, where the existence of an infeasible path does not necessarily imply the existence of a fault.

[b]Some methods discuss test sequence reduction from an existing set of test sequences. However, they do not provide hints as we do on which part(s) of the program should be covered first in order to effectively increase the coverage. That is, none of the existing CFG-based test generation methods focus on how *new* tests can be generated to effectively increase coverage.

on our results are also made. Section 4 presents an overview of some related studies. Finally, in Sec. 5 we offer our conclusions and recommendations for future research.

2. Methodology

In this section, we explain how to generate a small set of test sequences to satisfy the all-node and all-edge criteria. This is achieved through an effective increase in the coverage by each single test sequence.

2.1. *Test sequence generation with respect to the all-node criterion*

This criterion requires that every node in an RG be covered by at least one test sequence. Four different methods are used to generate these test sequences: two are hot spot prioritization-based and two are topological sort-based. Details of these methods are explained below.

2.1.1. *Hot spot prioritization-based methods*

Two methods M_1 and M_2 are derived based on hot spot prioritization. The major difference between these two methods is that M_1 uses a conservative approach to identify hot spots, whereas M_2 uses an aggressive approach. A hot spot is an uncovered node with the highest weight (to be explained in the subsequent sections) so that covering this node can increase the overall coverage in an effective way.

Method M_1:

The procedure for method M_1 is as follows.

M_{11}: Assign an initial value of one to each node.
M_{12}: Compute the weight for each uncovered node. The weight of a node is the number of uncovered nodes on the shortest path from the root of the graph to the given node.
M_{13}: Identify a hot spot, which is a node with the highest weight. If there is more than one hot spot, identify one of them by a random selection.
M_{14}: Backtrack from the hot spot identified at step M_{13} to the root of the graph. In this way, we have selected a path from the root to the hot spot, i.e., a test sequence to cover this hot spot. Mark all the nodes on this path as *covered* and change their values and weights to zero.
M_{15}: If all the nodes have a value zero, i.e., every node has been covered, STOP. Otherwise, go back to step M_{12}.

The computation at step M_{12} can be done by using a modified breadth first search (BFS).[c] After the computation, the weight of a node indicates the smallest number of uncovered nodes (i.e., nodes whose value is one) that are guaranteed to be

[c]Hereafter, we use BFS as the abbreviation for "Breadth First Search."

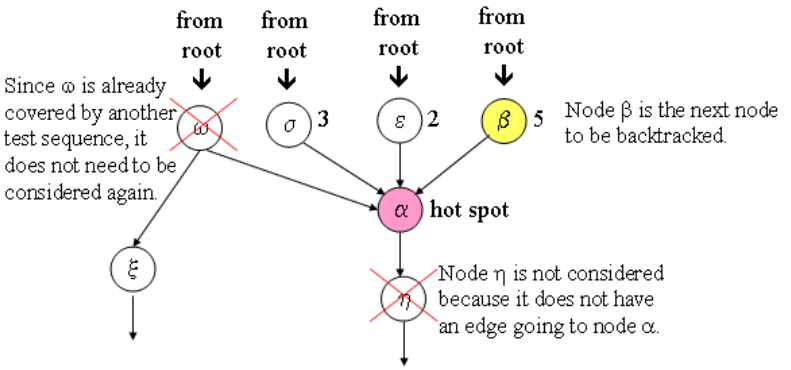

Fig. 1. A sample part of a reachability graph.

covered, if this node is covered. This is why we say method M_1 uses a conservative approach to identify hot spots.

During backtracking at step M_{14}, a *greedy* approach is used at each step. For a given hot spot (say α), we compare the weight of all the uncovered nodes that can go to α in one step (i.e., have an edge from itself to α). Identify the nodes with the highest weight. If there is more than one, randomly select one (say β) as the next node to be backtracked. Mark β as covered and change its value and weight to zero. Repeat the same process for β to find the next node to be backtracked and change its value and weight to zero. Continue this process until the root node is reached. Without losing generality, we use Fig. 1 to illustrate the backtracking process. The number next to each node is its weight. We start from the hot spot α. Suppose node ω is already covered and node η does not have an edge going to α. As a result, we only need to compare the weights of nodes σ, ε, and β (but not η or ω) with each other. Since node β has the highest weight, it is selected as the next node to be backtracked and its value and weight is changed to zero. This implies we are going to generate a test sequence covering the hot spot α containing the edge from β to α as part of the sequence.

Every time the process goes back to M_{12}, the weight of every uncovered node is recomputed. As a result, hot spots (i.e., the remaining uncovered nodes with the highest weight) may move to different locations to provide new guidelines on how to generate the next test sequence.

Method M_2:

In method M_1, we use a conservative approach to compute the weight for each uncovered node, whereas in this method we use an aggressive approach. More specifically, the weight of a given node in M_1 is the number of uncovered nodes on the shortest path from the root of the graph to the node. But, in M_2, it is the number of uncovered nodes on the longest path from the root of the "modified" graph (obtained by deleting all the cycles in the original graph) to the given node.

In method M_2, we first generate a *negated* graph from the original graph by assigning an initial value of *negative* one, instead of *one* as in method M_1, to every node. Any cycles in the original graph and the negated graph need to be removed (refer to step M_{22}). An algorithm for single-source shortest paths is used to compute the weight for each uncovered node in the negated acyclic graph. The Dijkstra's algorithm cannot be used because it requires all weights to be non-negative which is not the case in M_2. Hence, we have to use a different algorithm that can compute shortest paths even with negative-weight nodes. We name this algorithm DSP standing for DAG_SHORTEST_PATHS [5]. The time complexity of DSP is in the order of $V + E$, where V and E are the number of nodes and edges, respectively.

The procedure for method M_2 is outlined as follows:

M_{21}: This step is the same as M_{11} except that every node is assigned an initial value of negative one.

M_{22}: Use a modified DSP algorithm to compute the weight for each uncovered node in the corresponding negated acyclic graph, which is obtained by deleting all the cycles, if any, in the negated graph generated at step M_{21}. The weight of a given node is the number of uncovered nodes on the shortest path from the root to the given node.

M_{23}: Find a hot spot. A hot spot is a node whose weight has the smallest (negative) value. If there are more than one hot spot, we can randomly select one.

M_{24}: Conduct a backtracking from the hot spot to the root of the negated acyclic graph. A similar approach as in M_{14} is also used here.

M_{25}: If all the nodes have a value zero, i.e., every node has been covered, STOP. Go back to step M_{22}.

There are two important points at step M_{22} worth noting.

- The first step in DSP is to conduct a topological sort with respect to all the nodes in the graph. Since this can only be done on acyclic graphs, all the cycles (if any) in the graph have to be removed. To do so, a "modified" DFS-based[d] algorithm is used which can not only remove cycles by deleting all the back edges but also return a topological sort of the corresponding acyclic graph [5]. Since the purpose is to generate a small set of test sequences to cover all the nodes in a graph, it is all right to remove some back edges in order to break the cycles in the graph.
- The weight of a given node in method M_2 is the number of uncovered nodes on the shortest path from the root to the given node in the negated acyclic graph obtained by deleting all the cycles in the negated graph as described above. This corresponds to the number of uncovered nodes on the longest path from the root to the given node in the "modified" graph obtained by deleting all the cycles in the original graph.

[d]Hereafter, we use DFS as the abbreviation for "Depth First Search".

2.1.2. *Topological sort-based methods*

In this section, we explain a different strategy to decide which node in an RG should be covered first. Instead of the shortest path as in method M_1 or the "longest" path as in method M_2, a topological sort is used to list all the nodes of a directed acyclic graph[e] in a sequential listing such that if there is an edge from node u to node v, then u precedes v in the listing. In methods M_3 and M_4, we reverse such a topologically sorted order and make one pass over the nodes in the reversed listing in order to increase the coverage with respect to the all-node criterion in an effective way. The underlying motivation is that a topological sort of a graph can be viewed as an ordering of its nodes along a horizontal line so that all directed edges go from left to right. Covering the nodes from the beginning of the reversed listing (i.e., the nodes starting from the right end of the original topological listing) has a good chance of generating a path that covers more nodes. As a result, it is more likely to increase the coverage effectively. Depending on which algorithm is used for the sorting, there may exist more than one topological order of a directed acyclic graph. In our study, method M_3 uses a BFS-based algorithm, whereas method M_4 uses a DFS-based algorithm [5, 13]. The impact of these two algorithms on test generation is discussed in Sec. 3.

Method M_3:

The procedure for method M_3 is outlined as follows:

M_{31}: Conduct a BFS-based topological sort to generate a sequential listing of all the nodes.

M_{32}: Reverse the listing obtained at step M_{31}.

M_{33}: If the reversed listing is empty, STOP. Otherwise, remove the first node from this listing.

M_{34}: Find a path from the root of the graph to the node selected at step M_{33} using a "modified" DFS algorithm. The DFS is modified in a way such that it will check every visited node to see whether it is the target node. If so, the search is terminated. And, we have generated a test sequence for covering the target node. All the nodes on the path are marked as "covered" and are removed from the remainder of the reversed listing.

M_{35}: Go back to step M_{33}.

As discussed before, the underlying graph at step M_{31} has to be acyclic, i.e., all the cycles (if any) in the graph have to be removed. This is done by a "modified" BFS-based algorithm that can not only remove cycles by deleting all the back edges but also return a topological listing of the corresponding acyclic graph.

[e]If the graph has cycles, these cycles will be deleted as explained in methods M_3 and M_4.

Method M_4:

The procedure for M_4 is the same as that for M_3 except that the topological sort at step M_{41} is a DFS-based algorithm, whereas it is a BFS-based algorithm at step M_{31}.

An important observation on these four methods is that in M_1 and M_2, a re-computation of the weight for all remaining uncovered nodes is necessary in order to identify a new hot spot every time after the previous hot spot is covered; whereas, only one topological sort is needed in M_3 and M_4 to generate a sequential listing of all the nodes. The impact on the run-time complexity and number of test sequences generated is discussed in Secs. 2.3 and 3, respectively.

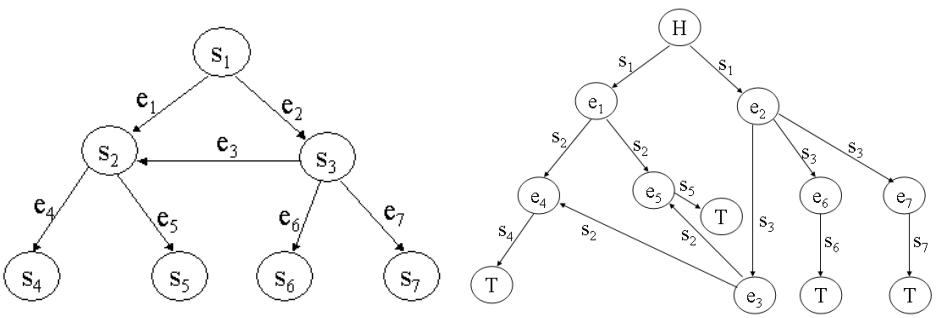

Fig. 2. A sample graph (left) and its dual graph (right). Nodes T and H are "pseudo" nodes in the dual graph.

2.2. *Test sequence generation with respect to the all-edge criterion*

To cover all the edges of a reachability graph, we first generate a dual graph with respect to the original graph such that each node in the dual graph represents one edge in the original graph and each edge in the dual graph represents one node in the original graph. Without losing generality, we use the graphs in Fig. 2 as an example for further explanation. The left side of this figure is the original graph and the right side is its corresponding dual graph. A pseudo node (H in our case) is added to the dual graph as the head of the edge corresponding to the root of the original graph (s_1 in our case). Another pseudo node (T in our case) is added as the tail of the edge corresponding to each node of the original graph which has no outgoing edges (s_4, s_5, s_6, and s_7 in our case). These pseudo nodes will be ignored while generating test sequences to cover all the nodes in the dual graph, i.e., to cover all the edges in the original graph. Through this conversion, our methods (namely, M_1, M_2, M_3, and M_4) that solve the all-node coverage problem discussed in Sec. 2.1 can also be applied here to solve the all-edge coverage problem.

2.3. *Run-time and space complexity*

For M_1 and M_2, the most expensive step is to compute the weights at steps M_{12} and M_{22}, respectively. It requires an order of $\Theta(V+E)$ for each computation, where V and E are the number of nodes and edges in the graph. In the first iteration, the weight of every node in the graph needs to be computed because no node has been covered yet. This requires an order of $\Theta(V(V+E))$. In the subsequent iterations, weight is computed only for the remaining uncovered nodes. Clearly, the number of such uncovered nodes is less than the total number of nodes in the graph. Hence, the time complexity of each subsequent iteration will not exceed the order of $\Theta(V(V+E))$. Suppose there are n iterations (i.e., n test sequences to satisfy the all-node or all-edge criterion), the overall run-time complexity for M_1 and M_2 is $\Theta(nV(V+E))$.

For M_3 and M_4, the most expensive step is to find a path from the root to a selected node using a "modified" DFS algorithm (refer to step M_{34}). This implies the run-time complexity for generating each test sequence is in the order of $\Theta(V+E)$. Hence, the overall run-time complexity for M_3 and M_4 is in the order of $\Theta(n'(V+E))$, where V and E are the number of nodes and edges in the graph, and n' is the number of test sequences generated. Note that a topological sorting at steps M_{31} and M_{41} requires an order of $\Theta(V+E)$. However, this only needs to be performed once so it has little impact on the overall run-time complexity.

Based on the above discussion, M_1 and M_2 have a higher run-time complexity than M_3 and M_4, but our case study reported in the next section (refer to Tables 1 and 2) suggests that M_1 and M_2 are more effective than M_3 and M_4 in generating test sequences with respect to the all-node and all-edge criteria. More discussions appear in Sec. 3. Finally, since graphs are saved as adjacency lists, the space required for each method is the same and in the order of $\Theta(V+E)$.

3. A Case Study

To demonstrate the effectiveness of our methods, we conducted a case study using the reachability graphs generated for five commonly used distributed algorithms. Below is a brief description of these algorithms and their corresponding notations:

- *le-n*: an algorithm for leader election, i.e., for determining the leader among a set of n processes organized in a unidirectional ring structure [23].
- *tp-n*: a token passing-based algorithm for ensuring mutual exclusion to shared resources among a set of n processes organized in a ring structure [23].
- *me-n*: an algorithm for ensuring mutual exclusion among a set of n processes on a fully connected network [23].
- *fl-n*: a flooding-based algorithm for constructing a minimum spanning tree in a randomly connected network of n processes [24].
- *sw-n*: a simplified implementation of the sliding window protocol, which exercises flow control for reliable data transfer between two processes, where n is the window size [24].

Table 1. Data for the all-node criterion.

Reachability graph	Number of nodes	Number of edges	Number of test sequences				The longest test sequences			
			M_1	M_2	M_3	M_4	M_1	M_2	M_3	M_4
le-3.reduced	9	8	1	1	1	1	1(9)	1(9)	1(9)	1(9)
tp-5.reduced	56	56	1	1	1	1	1(56)	1(56)	1(56)	1(56)
me-3.reduced	19	21	4	4	4	4	1(7)	1(7)	1(7)	1(7)
fl-3.reduced	21	28	9	9	10	10	1(6)	1(6)	1(6)	4(6)
fl-4.reduced	34	51	13	13	17	17	1(7)	1(7)	1(7)	9(7)
fl-5.reduced	50	83	18	17	26	26	1(8)	1(8)	1(8)	12(8)
Sw-2.reduced	219	585	41	36	60	47	1(115)	1(84)	45(154)	20(154)
le-3.full	200	411	20	19	37	37	1(43)	1(43)	1(43)	1(43)
me-3.full	964	2476	115	110	212	212	1(33)	1(33)	1(33)	1(33)
fl-3.full	70	119	10	9	17	17	1(18)	1(18)	1(18)	1(18)
fl-4.full	153	310	21	18	35	35	1(22)	1(22)	1(22)	1(22)
fl-5.full	320	773	42	36	69	69	1(26)	1(26)	1(26)	1(26)

Reduced reachability graphs of the above algorithms are also constructed by using the blocking-based simultaneous reachability analysis [16]. This reduction can preserve fault detection capabilities such as deadlock detection while skipping a large number of intermediate states by allowing multiple processes to proceed simultaneously.

Data collected from our study are presented in Tables 1 and 2. The first column gives a description of the graph being tested. For example, the entries labeled *fl*-3.reduced, *fl*-4. reduced, and *fl*-5.reduced are the reduced graphs for the flooding protocol with 3, 4, and 5 processes, respectively. Similarly, the entries labeled *fl*-3.full, *fl*-4.full, and *fl*-5.full are the corresponding full reachability graphs. The second and third columns are the total number of nodes and edges, respectively, in each graph. The number of test sequences generated by each method for the all-node and the all-edge criteria appear in columns four to seven. For example, from Table 1 we find that method M_1 generates 18 test sequences to cover all the nodes in *fl*-5.reduced, and from Table 2 method M_2 generates 53 test sequences to cover all the edges in *le*-3.full. Finally, each entry of the last four columns has a format $\alpha(\beta)$ showing that of all the test sequences generated with respect to the given graph and method, the αth test sequence is the one which most improves the all-node or all-edge coverage by covering β nodes or β edges.

From these two tables, we make the following observations. Without losing generality, all the discussion is with respect to the all-node criterion and the same can be applied to the all-edge criterion, unless otherwise specified.

(1) In general, a graph containing more nodes and edges (i.e., the corresponding concurrent program having more states and transitions) requires more test sequences. For example, the graph for *le-3.full* has more nodes and edges than that for *le-3.reduced*, and the former also requires more test sequences to cover

Table 2. Data for the all-edge criterion.

Reachability graph	Number of nodes	Number of edges	Number of test sequences				The longest test sequences			
			M_1	M_2	M_3	M_4	M_1	M_2	M_3	M_4
le-3.reduced	9	8	1	1	1	1	1(8)	1(8)	1(8)	1(8)
tp-5.reduced	56	56	1	1	1	1	1(56)	1(56)	1(56)	1(56)
me-3.reduced	19	21	6	6	6	6	1(6)	1(6)	1(6)	1(6)
fl-3.reduced	21	28	14	14	14	14	1(5)	1(5)	1(5)	8(5)
fl-4.reduced	34	51	25	23	27	27	1(6)	1(6)	1(6)	16(6)
fl-5.reduced	50	83	35	35	46	46	1(7)	1(7)	1(7)	40(7)
sw-2.reduced	219	585	123	118	194	183	1(258)	1(114)	66(316)	43(316)
le-3.full	200	411	70	53	213	213	1(42)	1(42)	1(42)	1(42)
me-3.full	964	2476	491	346	1516	1516	1(32)	1(32)	1(32)	1(32)
fl-3.full	70	119	23	19	52	52	1(17)	1(17)	1(17)	17(17)
fl-4.full	153	310	69	57	161	161	1(21)	1(21)	1(21)	40(21)
fl-5.full	320	773	200	153	458	458	1(25)	1(25)	1(25)	45(25)

all the nodes than the latter. However, the relation between the number of nodes/edges and the number of test sequences is not monotonically increasing. Moreover, there are exceptions; for example, the graph for *fl*-5.reduced has less nodes and edges than that for *fl*-3.full, but the former requires more test sequences to cover all the nodes than the latter. This is because the number of nodes and edges is not the only factor in deciding how many test sequences are needed to cover all the nodes or all the edges. The structure of the graph also has an important impact.

(2) When the graph is small, the number of test sequences needed to cover all the nodes is about the same for every method. But, as the size of the graph increases (or more precisely, as the number of states and transitions of the corresponding concurrent program increases), the number of test sequences generated by M_1 or M_2 is smaller than that generated by M_3 or M_4. Such a difference becomes more significant when the graph becomes larger. One reason is that M_1 and M_2 re-compute the weights for all remaining uncovered nodes after each test sequence generation, which can identify a *new* hot spot — the node that should be covered by the next sequence — for the next test generation. Another reason is that M_1 and M_2 use a greedy strategy in backtracking; namely, for a given node we choose its neighboring uncovered node with the maximum weight as the next node to be visited. Although a local optimization at each node does not always guarantee a global optimization to give the optimal test sequence, a test sequence constructed to cover the hot spot using our greedy backtracking can, in general, increase coverage in a very effective way. One may point out that this advantage (i.e., smaller number of test sequences) is accompanied by a trade-off in terms of higher run-time complexity for M_1 and M_2 (refer to Sec. 2.3); this is not an issue in our study. Our data show that the time

required for test generation on a Pentium 4 desktop (with a 2.8 GHz processor and 512 MB memory) varies from less than one millisecond (e.g., generating test sequences to cover all the nodes or edges of *le*-3.reduced) to about 53 seconds (e.g., generating test sequences to cover all the edges of *me*-3.full using method M_2). In fact, aside from a few exceptions, all other test generations in our study can be completed in less than three seconds, with many in less than one second. However, the exact time may vary depending on the user environment. In practice, the difference in run-time complexity can have an impact on reachability graphs with a large number of nodes and edges. More studies on the trade-off between run-time complexity and the number of test sequences generated are to be conducted.

(3) In all the cases, M_2 generates either the same number or a smaller number of test sequences than M_1. Recall that M_1 uses a conservative approach to identify hot spots, whereas M_2 uses an aggressive approach. For discussion purposes, let us suppose there are two paths (say P_α and P_β) from the root to a given node ω. Suppose also the number of uncovered nodes on P_α and P_β is N_α and N_β, where $N_\alpha < N_\beta$. Method M_1 will assign N_α (instead of N_β) as the weight of ω so that no matter which path is executed, at least additional N_α nodes are covered, whereas method M_2 assigns N_β (instead of N_α) as the weight of ω in order to identify the hot spots differently.

Given a node in a CFG, there may be more than one path from the root to the node. Different paths may contain a different number of uncovered nodes, and some paths may be infeasible. Since there is no guarantee that a randomly selected path in a CFG is always feasible, a better way to compute the weight of a given node is to use a *conservative* approach such as M_1 so that no matter which path is selected, we can guarantee that covering the given node guarantees the coverage of at least n nodes where n equals the weight of the given node. This implies that M_1 can be applied to both RG- and CFG-based test generation methods. On the other hand, M_2 assumes that every path from the root to a given node is feasible and uses an *aggressive* approach to identify hot spots. As a result, it is only appropriate for RG-based, but not CFG-based, test generation methods. Our data show that in all the cases, M_2 generates either the same number or a smaller number of test sequences than M_1. This confirms that CFG-based test generation methods with special tailoring measures for infeasible paths may render them less efficient (from the coverage improvement point of view) than RG-based test generation methods.

(4) In all the cases, M_3 (which uses a BFS-based topological sort) and M_4 (which uses a DFS-based topological sort) generate the same number of test sequences except for *sw*-2.reduced where the difference is small. Since M_3 and M_4 use different topological sort algorithms, this suggests that the specific ordering of nodes in the initial sorted list (refer to step M_{31}) has little impact on the number of generated test sequences.

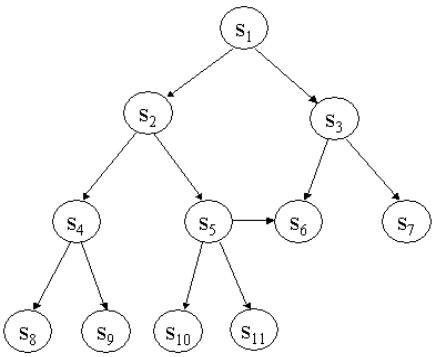

Fig. 3. A sample reachability graph.

The first response to this observation might be that it must be a coincidence because of the graphs used in our study. After a careful examination of these two methods, we notice that this does not happen just by chance. For explanatory purposes, consider the graph in Fig. 3. The listing of nodes in the order that should be covered is S_6, S_{11}, S_{10}, S_9, S_8, S_7, S_5, S_4, S_3, S_2, S_1 for M_3, and S_8, S_9, S_4, S_{10}, S_{11}, S_6, S_5, S_2, S_7, S_3, S_1 for M_4. Indeed, these two listings have different orders. For M_3, nodes are covered level-by-level starting from the deepest level, followed by those having the next deepest level, etc., whereas for M_4, nodes are covered starting from those which are at the "deepest" positions in a given sub-graph, followed by their ancestors in the same sub-graph before moving to another sub-graph. However, it does not matter which listing is used: we only have to cover the leaf nodes because other "interior" nodes will automatically be covered if all the leaves are covered. More specifically, for M_3, we only need to select test sequences to cover S_6 followed by S_{11}, S_{10}, S_9, S_8, and S_7. After this, all other nodes are covered automatically. And for M_4 we only need to select test sequences to cover S_8 followed by S_9 (no need for S_4 since it will be covered automatically), S_{10}, S_{11}, S_6, (no need for S_5, and S_2), and S_7 (no need for S_3, and S_1). As a result, we have the same number of test sequences for M_3 and M_4 with respect to the all-node criterion. As indicated earlier, there may exist some exceptions. For example, *sw*-2.reduced gives different results on these two methods. There are 60 and 194 test sequences for the all-node and the all-edge criteria, respectively, if M_3 is used, and 47 and 183, respectively, if M_4 is used. In this example, M_3 requires more test sequences than M_4.

(5) Another interesting observation about M_3 and M_4 is that since a BFS-based topological sort usually has all the deepest leaf nodes at the beginning of its listing, M_3 is likely to generate the test sequence which most improves the coverage in the first few generated test sequences. On the other hand, some leaf nodes might appear in the middle of the listing generated by a DFS-based topological sort such as S_6 and S_7 in Fig. 3. As a result, it is possible that

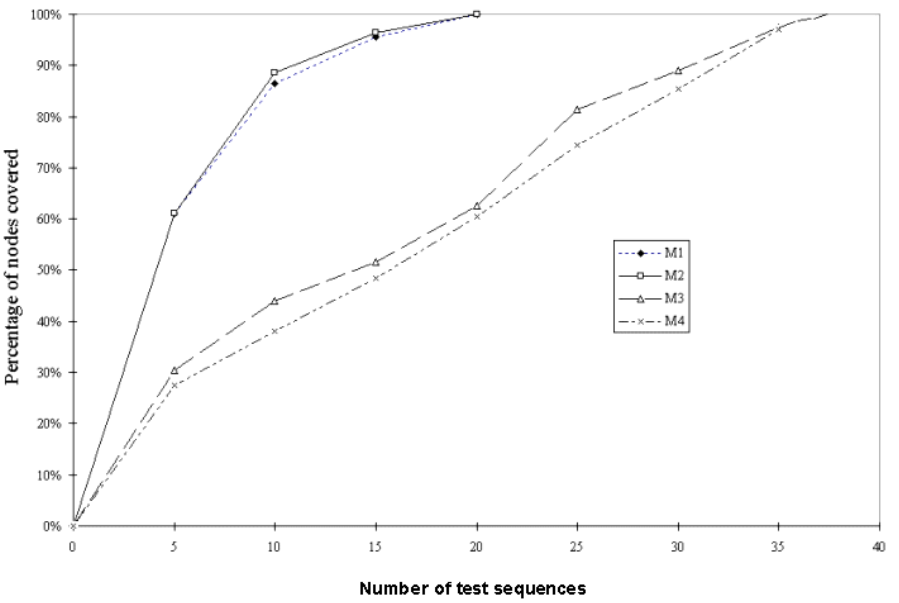

Fig. 4. Percentage of node coverage versus the number of test sequences for *le*-3.full.

such a test sequence generated by M_4 is also in the middle of all generated test sequences. For example, with respect to *fl*-5.reduced and the all-node criterion, the test sequence which most improves the coverage is the 1st test sequence (with 8 nodes) if M_3 is used, and the 12th test sequence (with 8 nodes) if M_4 is used. Similarly, with respect to the same graph and the all-edge criterion, the test sequence which most improves the coverage is the 1st test sequence (with 7 edges) if M_3 is used, and the 40th test sequence (with 7 edges) if M_4 is used. As for M_1 and M_2, the first test sequence is the one which most improves the coverage in all cases.

This information is important because if resources prevent us from generating an all-node or all-edge adequate test set, we might end up only generating the first few test sequences. If so, it is better to generate such test sequences to achieve higher coverage (refer to Table 3). In this regard, M_3 can perform better than M_4 for the graphs used in our study.

(6) To observe how each additional test sequence increases the coverage, we can plot a curve by using the percentage of coverage as the index for the vertical axis and the number of test sequences as the index for the horizontal axis. Due to the space limit, we only present the curve for *le*-3.full with respect to the all-node criterion in Fig. 4 for explanation, and the same applies to other graphs. Each curve has a steeper slope in the beginning, implying that coverage increases in a more effective way with respect to the first few test sequences. This is particularly true for M_1 and M_2. In addition, the curves for M_1 and M_2 have steeper slopes than those for M_3 and M_4 which implies test sequences

Table 3. Coverage obtained by using the first *few* test sequences. If the total number of test sequences generated is less than/equal to five, a notation "—" is entered for that entry. Refer to the text for more details.

Reachability graph	Number of nodes	Number of edges	All-node criterion				All-edge criterion			
			M_1	M_2	M_3	M_4	M_1	M_2	M_3	M_4
le-3.reduced	9	8	—	—	—	—	—	—	—	—
tp-5.reduced	56	56	—	—	—	—	—	—	—	—
me-3.reduced	19	21	68.4	68.4	68.4	63.2	57.1	57.1	52.3	33.3
fl-3.reduced	21	28	47.6	47.6	38.1	33.3	32.1	35.7	28.6	17.9
fl-4.reduced	34	51	35.3	35.3	26.5	23.5	29.4	23.5	23.5	15.7
fl-5.reduced	50	83	28.0	28.0	24.0	20.0	21.7	20.5	15.7	9.6
sw-2.reduced	219	585	59.8	47.1	47.0	8.2	51.1	49.5	58.1	16.6
le-3.full	200	411	33.0	33.0	24.5	24.5	16.1	21.2	15.8	12.4
me-3.full	964	2476	16.3	16.2	7.6	7.6	6.2	6.2	4.4	1.7
fl-3.full	70	119	38.6	47.1	32.9	32.9	34.5	27.7	20.7	15.9
fl-4.full	153	310	34.6	26.8	20.9	20.9	18.4	16.5	10.3	8.7
fl-5.full	320	773	21.9	22.2	11.3	11.3	13.6	14.2	4.7	4.0

generated by the first two methods are more effective than those by the last methods at increasing the coverage. This is consistent with what we discussed in item (2) listed above.

(7) The cumulative coverage obtained using the first "*few*" test sequences generated by each method is listed in Table 3. More specifically, let N be the total number of test sequences. Depending on whether N is greater than 5 and less than 20 (i.e., $5 < N < 20$), between 20 and 80 (i.e., $20 \leq N \leq 80$) or greater than 80 (i.e., $N > 80$), the all-node and all-edge coverage is measured using the first 2, 3, and 5 test sequences, respectively. If N is less than or equal to 5, no measurement is done. For the graph *fl*-5.reduced, there are 18, 17, 26, and 26 test sequences generated by M_1, M_2, M_3, and M_4 for the all-node criterion, and 35, 35, 46, and 46, respectively, for the all-edge criterion. Of these test sequences, the first two or three are chosen from each set which correspond to 11.1% (M_1), 11.8% (M_2) and 11.5% (M_3 and M_4), respectively, of all the test sequences for the all-node criterion, and 8.6% (M_1 and M_2) and 6.5% (M_3 and M_4), of all the test sequences for the all-edge criterion. The cumulative coverage ranges from 20% (M_4) to 28% (M_1 and M_2) for all-node, and 9.6% (M_4) to 21.7% (M_1) for all-edge.

Similarly, for a larger graph such as *me*-3.full, there are 115, 110, 212, and 212 test sequences generated by M_1, M_2, M_3, and M_4 for the all-node criterion, and 491, 346, 1516, and 1516, respectively, for the all-edge coverage. Of these test sequences, the first five are chosen from each set. This corresponds to 4.3% (M_1), 4.6% (M_2), and 2.4% (M_3 and M_4) of all the test sequences for the all-node criterion, and 1.0% (M_1), 1.4% (M_2), and 0.3% (M_3 and M_4) of all the test sequences for the all-edge

criterion. Yet, the coverage achieved by the first five ranges from 7.6% (M_3 and M_4) to 16.3% (M_1) for all-node, and 1.7% (M_4), 4.4% (M_3) and 6.2% (M_1 and M_2), respectively, for all-edge.

This observation is consistent with what we discussed in item (6). From the data in Table 3 and the curves in Fig. 4, it is clear that compared with the rest of the test sequences, the first few test sequences can increase the coverage in a much more effective way.

4. Related Studies

A survey on unit test coverage and adequacy can be found in [32], and on test case generation in [7, 18]. In particular, a number of test coverage criteria, as well as methods for selecting paths to satisfy them, have been developed for testing sequential programs based on their CFGs [1, 4, 9, 20, 22]. There are two major differences between these studies and our methods. First, none of them is focused on how *new* test sequences can be generated to effectively increase coverage. Second, they have to deal with the *feasibility* problem. As CFGs represent static program structure, some paths in a CFG may not be feasible at runtime. This has a very significant implication on the path selection process. For example, a set of test paths that is generated to satisfy the all-node criterion may not be able to actually cover all the nodes, as some paths in the set are infeasible. Some CFG-based techniques such as [8, 29] have been proposed to address this infeasibility issue. In contrast, structural testing of a concurrent program is usually based on its RG, where every node is a reachable state and every path is a feasible test execution sequence. Therefore, even though some CFG-based techniques can be applied to testing of concurrent programs, they may not be effective.

Taylor *et al.* [25] proposed a framework to conduct structural testing of concurrent programs. A family of coverage criteria for testing concurrent programs was developed, including all-concurrency-states, all-synchronizations, and all-concurrency-path. The subsumption relationship among these criteria was studied. However, the authors did not discuss how to actually select test paths to satisfy these criteria.

Koppol *et al.* presented an incremental approach to structurally test concurrent programs [12]. In their approach, a global state graph is constructed incrementally to alleviate the state explosion problem. The focus of [12] is to develop an annotated LTS model to preserve important test information during LTS reduction. This work is complementary to ours in that the test generation methods in our paper can be applied to the incremental testing process described in [12].

Lee and Hao [14] proposed different algorithms for selecting a small subset of test sequences from a large set of test sequences without sacrificing some "fault" coverage. A commonly used criterion of fault coverage is to test each edge in the reachability graph at least once. While their study focused on test sequence reduction from an existing set of test sequences, our methods are used to generate test sequences directly from the reachability graph.

Some test generation methods based on path analysis are also proposed for testing concurrent programs [11, 30, 31]. In these methods, local paths are first generated for individual processes based on their CFGs. Such generation can be accomplished by test path selection techniques for sequential programs. These local paths are then used to compose global paths. A global path consists of a set of local paths, one for each process, in which every synchronization statement has to be syntactically matched with a counterpart, e.g., a *send* statement must be matched with a *receive* statement. These methods are mainly concerned with how to compose global paths. None of them provides hints as we do on which part(s) of a program should be covered first in order to increase the coverage in an effective way.

Reachability testing [10, 15] is a dynamic approach to testing concurrent programs. During reachability testing, each test run is monitored and the SYN-sequence that is exercised is recorded in an execution trace. At the end of a test run, the trace is analyzed to derive "race variants" of the trace. A race variant represents the beginning part of a SYN-sequence that definitely could have happened but did not, due to the way race conditions were arbitrarily resolved during execution. Each of the race variants is used to conduct a prefix-based test run in which the variant is forced to be exercised deterministically, and then the test run is allowed to proceed non-deterministically. The new SYN-sequences exercised by these prefix-based test runs are then analyzed to derive new race variants, and so on. This process is repeated until all the SYN-sequences of the subject program are exercised. Our work is different from reachability testing in the following two aspects. First, reachability testing is an implementation-based approach in which test sequences are generated on-the-fly, without constructing any static model. In contrast, our work is model-based in the sense that an RG is first constructed and then test sequences are generated from the RG. Model-based and implementation-based approaches are complementary to each other, since some faults can only be detected by model-based or implementation-based approaches, but not both. Second, reachability testing is an exhaustive approach, i.e., it intends to execute all possible SYN-sequences that could be exercised by the subject program. Exhaustive testing can maximize test coverage, but is often impractical to accomplish due to resource constraints. In contrast, our work generates test sequences to satisfy a given coverage goal, and has the potential to be applied to larger and/or more complex programs.

Finally, we note that model checking is an alternative approach to testing concurrent programs. In this approach, the reachability graph of a given program is directly searched for software bugs. Our work is complementary to model checking in the following aspects. First, a reachability graph is often constructed from the design of a concurrent program, which can be considered as an abstraction of the program. Searching through a reachability graph verifies the abstraction, not the program. Therefore, there is still a need to generate test sequences from the reachability graph and then use them to test the program. Second, many concurrent

programs need to interact with various physical devices. Such interactions cannot be directly represented in a reachability graph, and thus cannot be directly verified by searching the graph. Finally, in order to apply model checking, the properties that need to be checked must be formally specified, which is often difficult for practical applications where a software developer knows what behaviors are expected but does not know how to specify them using a required formal notation.

5. Conclusion

In this paper, we present four different test sequence generation methods (two based on hot spot prioritization and two based on topological sort) for structurally testing concurrent programs. We also explain the differences between CFG-based and RG-based test generation methods. Our methods can effectively generate a small set of test sequences to cover all the nodes in a reachability graph (i.e., all the states in the corresponding concurrent program). The same methods can also be used to generate test sequences for the all-edge criterion (i.e., all transitions in the program) by applying them to the corresponding dual graphs. Of these methods, M_1 and M_2 generate fewer test sequences than M_3 and M_4 to achieve 100% node and edge coverage. However, M_1 and M_2 have a higher run-time complexity than M_3 and M_4. For complex programs, the size of the reachability graphs can be very large which may make the use of M_1 and M_2 less attractive. Under this condition, M_3 and M_4 seem to be a more practical choice. Note that since our methods generate test sequences from a reachability graph and are essentially independent from how the graph is constructed, they can be applied to different types of concurrent programs, including multithreaded programs in which threads communicate by accessing shared memory and distributed programs in which threads communicate by sending and receiving messages.

Our data also indicate that the coverage can be increased in a much more effective way by the first few test sequences generated by our methods than by the rest of the test sequences. This is very important because when resource constraints only allow us to generate a few test sequences, it is usually better to use those which can give a higher coverage. Finally, while the advantage of using a smaller set of test sequences is obvious in terms of test sequence management, output verification, etc., it is also important to examine the fault detection effectiveness of these test sequences using real defect data collected in practice. Our ongoing research is to apply test generation methods discussed in this paper to real-life concurrent programs and determine how they can help testing practitioners do a better job in finding software bugs.

References

1. R. H. Carver and K. C. Tai, *Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs* (Wiley, 2006).
2. R. Carver and K. C. Tai, Replay and testing for concurrent programs, *IEEE Software* **8**(2) (1991) 66-74.

3. R. H. Carver and K. C. Tai, Use of sequencing constraints for specification-based testing of concurrent programs, *IEEE Trans. Software Engineering* **24**(6) (1998) 471–490.
4. L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, A formal evaluation of data flow path selection criteria, *IEEE Trans. Software Engineering* **15**(11) (1989) 1318–1332.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. (MIT Press, Cambridge, MA, 2002).
6. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, Multithread Java program test generation, *IBM Systems J.* **41**(1) (2002) 111–125.
7. J. Edvardsson, A survey on automatic test data generation, in *Proc. Second Conf. on Computer Science and Engineering*, Linkoping, October 1999, pp. 21–28.
8. P. G. Frankl and E. J. Weyuker, Data flow testing in the presence of unexecutable paths, in *Proc. Workshop on Software Testing*, Banff, Alberta, Canada, July 1986, pp. 4–13.
9. W. E. Howden, Reliability of the path analysis testing strategy, *IEEE Trans. Software Engineering* **2** (1976) 208–214.
10. G. H. Hwang, K. C. Tai, and T. L. Huang, Reachability testing: An approach to testing concurrent software, *Int. J. Software Engineering and Knowledge Engineering* **5**(4) (1995) 493–510.
11. T. Katayama, E. Itoh, and Z. Furukawa, Test-case generation for concurrent programs with the testing criteria using interaction sequences, in *Proc. 6th Asian-Pacific Software Engineering Conf.*, Takamatsu, Japan, December 1999, pp. 590–597.
12. P. V. Koppol, R. H. Carver, and K. C. Tai, Incremental integration testing of concurrent programs, *IEEE Trans. Software Engineering* **28**(6) (2002) 607-623.
13. R. Kruse, C. L. Tondo, and B. Leung, *Data Structures & Program Design in C*, 2nd ed. (Prentice Hall, 1997).
14. D. Lee and R. Hao, Test sequence selection, in *Proc. PSTV* (*Protocol Specification, Testing, and Verification*)–*FORTE* (*Formal Techniques for Networked and Distributed System*), Cheju Island, Korea, August 2001, pp. 269–284.
15. Y. Lei and R. H. Carver, Reachability testing of concurrent programs, *IEEE Trans. Software Engineering*, **32**(6) (2006) 382–403.
16. Y. Lei and K. C. Tai, Blocking-based simultaneous reachability analysis of asynchronous message-passing programs, in *Proc. Int. Symp. Software Reliability Engineering*, November 2002, pp. 316–326.
17. Y. Lei and W. E. Wong, A novel framework for non-deterministic testing of message-passing programs, in *Proc. Int. Symp. High Assurance Systems Engineering* (*HASE*), Heidelberg, Germany, October 2005, pp. 66–75.
18. P. McMinn, Search-based software test data generation: A survey, *Software Testing, Verification and Reliability* **14**(2) (2004) 105–156.
19. G. Myers, *The Art of Software Testing* (John Wiley & Sons, 1979).
20. S. C. Ntafos, A comparison of some structural testing strategies, *IEEE Trans. Software Engineering* **14**(6) (1988) 868–874.
21. K. Ozdemir and H. Ural, Protocol validation by simultaneous reachability analysis, *Computer Communications* **20** (1997) 772–788.
22. S. Rapps and E. J. Weyuker, Selecting software test data using data flow information, *IEEE Trans. Software Engineering* **11**(4) (1985) 367–375.
23. M. Raynal, *Distributed Algorithms and Protocols* (John Wiley & Sons, 1988).
24. A. Tanenbaum, *Computer Networks* (Prentice Hall, 1996).

25. R. N. Taylor, D. L. Levine, and C. D. Kelly, Structural testing of concurrent programs, *IEEE Trans. Software Engineering* **18**(3) (1992) 206–214.
26. K. C. Tai, Testing of concurrent software, in *Proc. 13th Annual Int. Computer Software and Applications Conf.*, Orlando, Florida, September 1989, pp. 62–64.
27. K. C. Tai, R. H. Carver, and E. Obaid, Debugging concurrent ada programs by deterministic execution, *IEEE Trans. Software Engineering* **17**(1) (1991) 45–63.
28. A. Valmari, A stubborn attack on state explosion, *Formal Methods in System Design* **1**(4) (1992) 297–322.
29. The χSuds Toolsuite (http://xsuds.argreenhouse.com/), Telcordia Technology (formerly Bellcore or Bell Communications Research), Morristown, New Jersey, 1998.
30. R. D. Yang and C. G. Chung, Path analysis testing of concurrent programs, *Information and Software Technology* **34**(1) (1992) 43–56.
31. C. Yang, A. L. Souter, and L. L. Pollock, All-du-path coverage for parallel programs, in *Proc. Int. Symp. Software Testing and Analysis*, Clearwater Beach, FL, March 1998, pp. 153–162.
32. H. Zhu, P. Hall, and J. May, Software unit test coverage and adequacy, *ACM Computing Survey* **29**(4) (1997) 366–427.
33. H. Zhu and X. He, A methodology of testing high-level petri nets, *Information and Software Technology* **44**(8) (2002) 473–489.