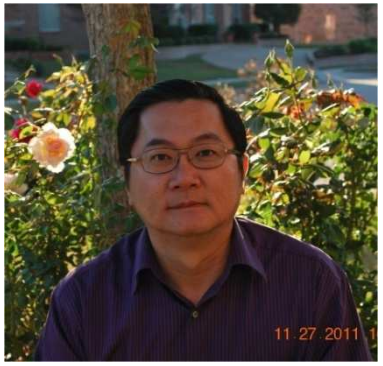
Effective Generation of Test Sequences for Structural Testing of Concurrent Programs

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
http://www.utdallas.edu/~ewong

Speaker Biographical Sketch



- Professor & Director of International Outreach
  Department of Computer Science
  University of Texas at Dallas
- Guest Researcher
  Computer Security Division
  National Institute of Standards and Technology (NIST)
- Vice President, IEEE Reliability Society
- Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)
- Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project
  - *Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses*
- Founder & Steering Committee co-Chair for the SERE conference (*IEEE International Conference on Software Security and Reliability*) (http://paris.utdallas.edu/sere13)

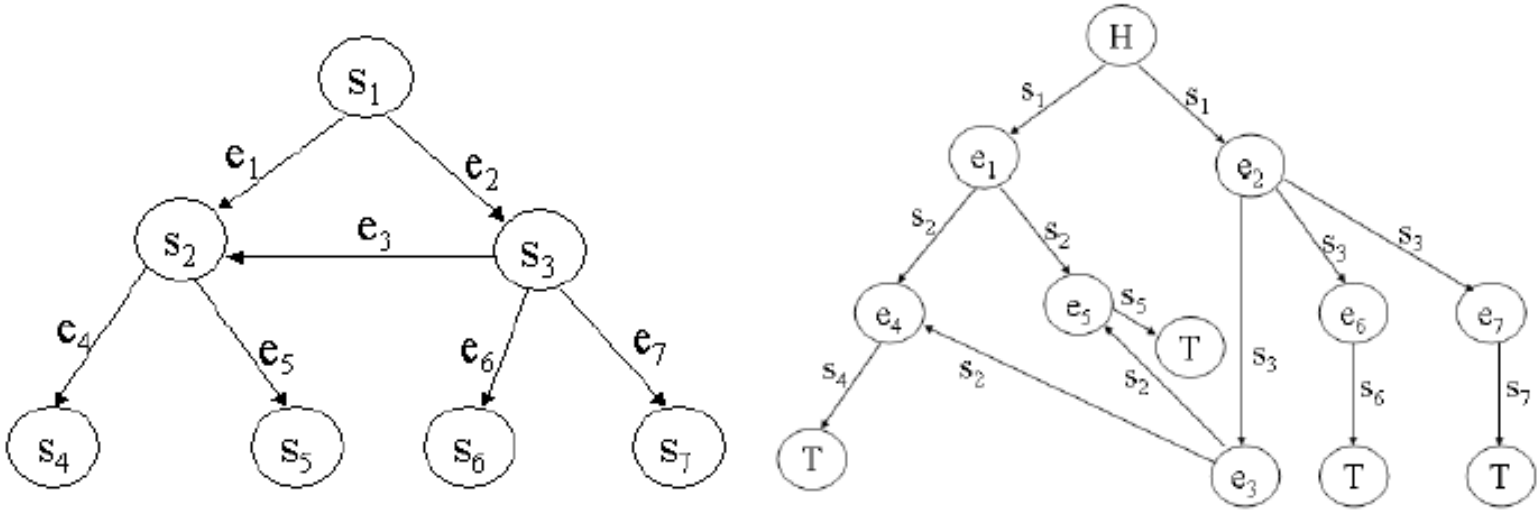
Motivation

- Let *P* be a concurrent program.
- Structural testing of *P* involves
  - White box-based testing
  - Deriving the reachability graph of *P*
  - Generating a set of test sequences from this graph to satisfy some well-defined coverage criterion
    - all-node criterion
    - all-edge criterion
- In order to reduce testing effort and costs, it is important to reduce the number of test sequences being selected
- Four different methods
  - Two based on *hot spot prioritization*
  - Tow based on *topological sort*

All-node versus all-edge/graph versus dual graph (1)

- Without losing generality, we only discuss the all-node 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
  - Each edge in the dual graph represents one node in the original graph.
  - Refer to the figure on the following slide for details.
    - A pseudo node H is added to the dual graph as the head of the edge converted from the root of the original graph (s_1 in our case), and
    - A pseudo node T is added to the dual graph as the tail of the edge converted from each leaf of the original graph (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, the methods that solve the all-node coverage problem can also be applied to solve the all-edge coverage problem.

All-node versus all-edge/graph versus dual graph (2)



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

Hot spot prioritization-based methods

- Two methods M_1 and M_2 are derived with this strategy
- The major difference between these two methods
  - M_1 uses a conservative approach to identify hot spots
  - M_2 uses an aggressive approach
- A hot spot is an uncovered node with the highest composite weight (to be explained) so that covering this node *can increase the overall coverage in an efficient way*

Method M_1 (1)

- M_{11}: Assign a weight to each node.
  - Each node is initially assigned a weight of one.
  - If a node is covered by at least one test sequence, its weight is reassigned to zero.
  - Once a node's weight becomes zero, it will not be changed again, i.e., once a node is covered, it stays covered.
  - This also implies that every uncovered node has a weight of one and every covered node has a weight of zero.
  - Note that the weights in our graph are associated with nodes instead of edges.

Method M_1 (2)

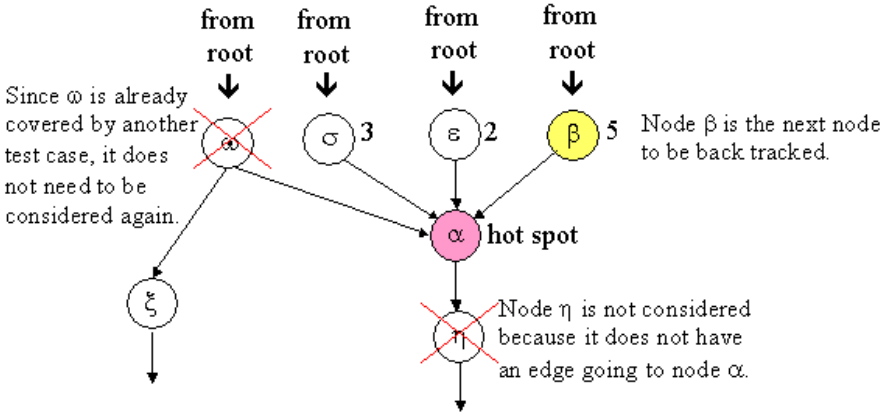
- M_{12}: If all the nodes have a weight zero, i.e., every node has been covered, STOP.
- M_{13}: Compute the composite weight for each uncovered node.
  - The composite weight of a node is the number of uncovered nodes that are guaranteed to be covered, if this node is covered.
  - This is why we say method M_1 uses a conservative approach to identify hot spots
  - It is the number of uncovered nodes (i.e., nodes whose weight is one) on the shortest path from the root of the graph to the given node.
  - Covered nodes (i.e., nodes whose weight is zero) on this path do not count.
  - The computation can be done by using a modified breadth first search.

Method M_1 (3)

- M_{14}: Identify the hot spot which is the node with the highest composite weight.
  - If there is more than one hot spot, identify (by a random selection) a hot spot which is one of the nodes with the highest composite weight.
- M_{15}: Backtrack from the hot spot identified at step M_{14} 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.
  - During the back tracking, a greedy approach is used at each step.
  - For a given hot spot (say α), we compare the composite weight of all the uncovered nodes that can go to α in one step, i.e., have an edge from itself to α.
    - These nodes must be the neighboring nodes of α.
    - However, since a reachability graph is a directed graph, not every uncovered neighboring node of α needs to be examined.

Method M_1 (4)

- M_{15}: (cont'd)
  - Refer to the following reachability graph.
  - The number next to each node is its composite weight.
  - We start from the hot spot α. The composite weights of nodes σ, ε, and β (but not η or ω) are compared with each other.
  - Since node β has the highest composite weight, it is selected as the next node to be back tracked.
  - This implies we are going to select a test sequence covering the hot spot α containing the edge from β to α as part of the sequence.



  - Mark β as covered and change its weight to zero.
  - Repeat the same process for β to find the next node to be back tracked.
  - Continue this process until the root node is reached.

Method M_1 (5)

- M_{16}: Go back to step M_{12}
- After each test sequence is selected, all the nodes on the sequence are marked as *covered* with their weights changed to zero at step M_{15}, and the composite weight of every uncovered node is recomputed at step M_{13}.
- As a result, hot spots (i.e., the remaining uncovered nodes with the highest composite weight) move to different locations to provide new guidelines on how to select the next efficient test sequence.
- Finally, every test sequence selected by using the back tracking at step M_{15} is feasible because this is part of the nature of reachability graphs.

Method M_2 (1)

- The composite 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 (after the weight negation and possible cycle deletions)* from the root to the given node.
- This also implies that M_1 uses *a conservative approach* to identify hot spots, whereas M_2 uses *an aggressive approach.*
- Once again, similar to method M_1, the *weights in our graph are associated with nodes instead of edges.*

Method M_2 (2)

- M_{21}: This step is the same as step M_{11} except that the initial weight of each node is *negative one* instead of one.
  - In the negated graph, every uncovered node has a weight of negative one and every covered node has a weight of zero.
- M_{22}: If all the nodes have a weight zero, i.e., every node has been covered, STOP.
- M_{23}: Use a modified DSP (DAG-SHORTEST-PATHS) algorithm to compute the composite weight for each uncovered node in the corresponding *negated acyclic graph.*
  - The well-known Dijkstra's algorithm cannot be used here because it requires all weights to be non-negative which is not the case in method M_2.

Method M_2 (3)

- M_{23}: (cont'd)
  - 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.
  - We used *a "modified" DFS-based algorithm* that can not only remove cycles by deleting all the back edges but also return a topological sort of the corresponding acyclic graph.
    - Cycle deletion and topological sort are done simultaneously by using one algorithm.
    - Since the purpose of M_2 is to select an efficient 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.

Method M_2 (4)

- M_{23}: (cont'd)
  - The composite weight of a given node is the number of uncovered nodes on the *shortest path from the root of the negated acyclic graph* to the given node.
  - This corresponds to the number of uncovered nodes on the *longest path from the root of the "modified" original graph* (obtained by deleting all the cycles in the original graph) to the given node.

Method M_2 (5)

- M_{24}: Find a hot spot.
  A hot spot is the node whose composite weight has the smallest (negative) value. If there is more than one hot spot, we can randomly select one.
- M_{25}: Conduct a back tracking from the hot spot to the root of the negated acyclic graph. The same approach as in M_{15} is also used here.
- M_{26}: Go back to step M_{22}.

Topological sort-based methods (1)

- *A topological sort* is used to list all the nodes of a directed acyclic graph in a sequential listing such that if there is an edge from node *u* to node *v*, then *u* precedes *v* in the listing.
  - If the graph has cycles, these cycles will be deleted as explained in methods M_3 and M_4.
- *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 efficient 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 in the reversed listing first* (i.e., the nodes starting from the right end of original topological listing) has a good chance of selecting a path that covers more nodes. As a result, it is more likely to increase the coverage efficiently.

Topological sort-based methods **(2)**

- Two methods M_3 and M_4 are derived with this strategy
- Method M_3 uses a BFS-based algorithm, whereas method M_4 uses a DFS-based algorithm.

Method M_3 (1)

- M_{31}: Conduct a BFS-based topological sort to generate a sequential listing of all the nodes.
  - As discussed before, the underlying graph 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 identify and ignore cycles but also return a topological listing of the corresponding acyclic graph.
- M_{32}: Reverse the listing obtained at step M_{32}.
- M_{33}: If the reversed listing is empty, STOP.
  Otherwise, remove the first node from this listing.
  - In the beginning, the reversed listing contains all the nodes in the graph.

Method M_3 (2)

- 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 a test sequence selected for covering the target node.
  - All the nodes in the sequence are marked as "covered" and removed from the remaining listing.
- M_{35}: Go back to step M_{33}.

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}.

Observation (1)

- In M_1 and M_2, a re-computation of the composite 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.
- Only one topological sort is needed in M_3 and M_4 to generate a sequential listing of all the nodes.

Run-time complexity for M_1 *and* M_2

- For methods M_1 and M_2, the most expensive step is to compute the composite weights at steps M_{13} and M_{23}, respectively.
- In the first iteration, the composite 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, composite 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 $\Theta(nV(V+E))$.

Run-time complexity for M_3 *and* M_4

- For method 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_{33}).
- This implies the run-time complexity for selecting 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 number of test sequences selected.
- 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 a one-time sorting so it has little impact on the overall run-time complexity.

Space complexity

- Since graphs are saved as adjacency lists, the space required for each method is the same and in the order of $\Theta(V+E)$.

Case Study

- To demonstrate the effectiveness of our methods, we conducted a case study using reachability graphs generated for five well-known distributed algorithms.
  - le-n: the leader election algorithm where n is the number of processes in the ring
  - tp-n: the token passing algorithm where n is the number of processes in the ring
  - me-n: the mutual exclusion algorithm where n is the number of processes in the network
  - fl-n: the flooding protocol where n is the number of process in the network
  - sw-n: the sliding window protocol where n is the window size
- Reduced reachability graphs of the above algorithms are also constructed.

Data for the all-node criterion

Reachability Graph	Number of nodes	Number of edges	Number of test cases				The longest test case			
			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)

Data for the all-edge criterion

Reachability Graph	Number of nodes	Number of edges	Number of test cases				The longest test case			
			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)

Observation (2)

- In spite of the run-time complexity based on the worst case analysis, all the experiments in our study can find a set of test sequences to cover all the nodes (or edges) in a reasonable time.
- In general, a graph containing more nodes and edges requires more test sequences to cover all the nodes and edges.
  - There are exceptions.
    - Graphs with a linked-list structure only need one test sequence.
- There is no monotonically increasing relation between the number of nodes/edges and the number of test sequences.
  - 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.

Observation (3)

- 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 nodes and edges increases), the number of test sequences selected by M_1 or M_2 is smaller than that selected by M_3 or M_4.
- Such a difference becomes more significant when the graph becomes larger.
- One may argue 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
  - It is never an issue in our study even for graphs with a few thousand nodes or edges.
  - Of course, this may not always be the case.
  - More studies on the trade-off between run-time complexity and the number of test sequences selected are to be conducted.

Observation (4)

- In general, the difference in terms of the number of test sequences selected by methods M_3 (which uses a BFS-based topological sort) and M_4 (which uses a DFS-based topological sort) is small.
  - There are exceptions.

Observation (5)

- M_3 has all the deepest leaf nodes appear in the beginning of its listing, there is a very good chance that the longest test sequence (i.e., the test sequence with the most number of nodes) is in the first few selected test sequences.
  - For M_3, nodes are covered level-by-level starting from the deepest level, that is, nodes having the largest distance (i.e., largest number of edges) from the root are covered first.
- Some leaf nodes might appear in the middle of the listing generated by using M_4. As a result, it is possible that the longest test sequence so selected is also in the middle of all selected test sequences.
- This information is important because if resources only allow us to select a few test sequences, it is better to select longer test sequences than shorter test sequences. In this regard, M_3 can perform better than M_4 for the graphs used in our study.

Observation (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.
  - Each curve has a steeper slope in the beginning, implying that coverage increases in a more efficient 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 selected by the first two methods are more efficient than those by the last methods on increasing the coverage.
  - The first "few" test sequences selected by our methods can increase the node and edge coverage in a very significant way.

Conclusion (1)

- We present four different test sequence selection methods (two based on hot spot prioritization and two based on topological sort) to effectively select a small set of test sequences to cover all the nodes in a reachability graph.
- The same methods can also be used to select test sequences for the all-edge criterion by applying them to the corresponding dual graphs.
- Of these methods, M_1 and M_2 select 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 practical applications, the size of the reachability graphs can be very large which may prevent M_1 and M_2 from being applied.
  - Under this condition, M_3 and M_4 seem to be a more practical choice.

Conclusion (2)

- Our data also indicate that the coverage can be increased in a significant way by the first few test sequences selected by our methods.
- While the advantage of using a smaller set of test sequences is obvious in terms of *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 sequence selection methods discussed in this presentation to real-life concurrent software and determine how they can help testing practitioners do a better job in finding software bugs.