State-based Tests Suites Automatic Generation Tool (STAGE-1)

Hoda Khalil
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada
hodakhalil@cmail.carleton.ca

Yvan Labiche
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada
labiche@sce.carleton.ca

***Abstract*—State diagrams are widely used to model software artifacts, making state-based testing an interesting research topic. When conducting research on state-based testing for evaluating different testing criteria, often there is a need to devise numerous test suites in a systematic way according to selection criteria such as all-edges, all-transition-pairs, or the transition tree (W-method). Moreover, one also needs to satisfy each criterion in as many ways as possible to account for possible stochastic phenomena within each criterion. The main issue is then: how to automate the generation of as many, or even all, the different test suites for each criterion? This paper presents the first part of a framework, an automation tool chain that generates test trees from a state machine diagram, extracts test cases from the generated trees, and composes a test suite from each generated tree. This tool is the first to generate all possible distinctive trees using Depth and Breadth First graph traversal algorithms. The tool chain should be of interest to researchers in state-based testing as well as practitioners who are interested in alternative adequate test suites especially for comparing the effectiveness of the different test suites satisfying one criterion and the effectiveness of the other different criteria.**

Keywords--state based testing; transition path coverage; mutation analysis; transition trees; automation.

I. INTRODUCTION

Researchers as well as practicing engineers are avid to facilitate and accelerate the process of conducting empirical studies and experiments as per the many teams studying test suites characteristics generated by different state-based testing methods [7, 14]

Given the need for automation and the huge number of software applications which behavior is specified with finite state machine (FSM) models [9], the focus of this document is to describe a novel tool that can facilitate the process of conducting empirical studies and experiments.

The State-based Test Suite Automatic Generation and Execution Tool (STAGE) is a multi-step framework tool chain. First, it gives the user the option to produce test suites according to different state-based adequacy criteria, currently including round-trip-path [11], using either breadth or depth First graph traversal algorithms. The main novel contribution of STAGE is that it generates all traversals for the above-mentioned criteria. The tool also supports random traversal of the FSM graph as a benchmark. The configurable tool accepts an input algorithm choice capable of generating several test suites. A single test suite output is mostly of interest to practitioners who want to obtain a test suite to verify a piece of software. Multi test suites output is interesting to researchers who want to compare all the possible adequate test suites for a criterion such as recognizing that different test suites for the same criterion may have different cost and effectiveness (e.g., at finding faults).

Automation is paramount with large test suites. During experimentation, an FSM with 17 states and 65 transitions led to 101,947 different test suites for the depth first traversal technique making manual generation very time consuming, error-prone, and sometimes nearly impossible.

II. BACKGROUND

STAGE is based on two verification concepts: state-based testing and the transition tree coverage criteria.

A. State Based Testing

State based testing belongs to the category of testing that represents the artifact under test using graph models. A state machine is a collection of states and transitions. A state is defined by the values of a set of variables, while a transition is the change of one or more of the values that occurs in zero time. An *FSM* is a graph where nodes represent states in the execution behavior of the software and edges represent transitions among the states [9].

B. Transition Tree Coverage

STAGE is concerned with path testing which is defined as testing designed to execute all or selected paths through a computer program [5]. STAGE focuses on complete round trip paths. A *round trip path* is a prime path of non-zero length that starts and ends at the same node; A path from n_i to n_j is a *prime path* if it is a simple path that does not appear as a proper sub path of any other simple path; A *simple path* contains only one iteration of a loop, if a loop is present in some sequence [9].

III. MOTIVATION

The general algorithm used by STAGE to convert an FSM into a transition tree representing test cases (each

path from root to leaf in the tree is a test case) is based on an algorithm introduced by Chow (the W-method) [17]. Chow's algorithm produces a breadth first (BFS) traversal tree of a given FSM and can also be used to produce depth first search (DFS) traversal trees [11]. Both traversal procedures are important to study since this impact effectiveness of the generated test suites [7]. Further experimentation is required to generalize the findings and compare tree generation techniques. We claim that STAGE can facilitate such experimentations.

Analyzing experimental results of using the different traversal algorithms can lead to better understanding of the factors affecting coverage or fault detection. For instance, exercising some sequences of states and transitions may be important for higher fault detection [7], some conjecture that some notion of variability among test cases impacts fault detection [4]. This would lead to new ideas to improve the possible ways to generate test cases from an FSM. Conducting such experimental studies requires that various case studies be used, that various adequacy criteria be used, and that as many adequate test suites as possible for each criterion is created. STAGE is a tool that supports doing so.

STAGE is the only available tool that generates all test suites that satisfy depth first and breadth first traversal for the round-trip paths criterion. This tool can be used to determine the best and worst cases for the criterion in terms of cost and effectiveness, test all possible scenarios, pick one test suite that satisfies the criterion the best and use it during the course of software verification, and finally deduce non-biased evaluations of criteria based on statistical data.

Table 1 Comparison with existing tools

Tool Name	Input	Criteria	Output
STRAP	SSG	elementary transition paths	all trans paths in 1 test suite
ParTeg	UML SD	control & boundary	1 test suite, both criteria satisfied
Kansomkeat & Rivepiboon	UML SD	state & transition	test cases collection, both criteria satisfied
AGeTeSC	Object	transition paths	boundaries based minimized set of test cases
ConData	EFSM in PSL	Transition & equivalence partitioning	one test suite satisfying all criteria
TRUST	UML SD	all transitions, & all round trip	one test suite
STAGE	FSM in DOT format	round trip, random, depth traversal, and breadth traversal.	all test suites for breadth /depth. distinct test suites in random, and 1 in round trip.

The novel algorithms provided by STAGE guarantees the generation of complete sets of trees based on the chosen traversal method, as well as the uniqueness of the generated trees.

IV. Related Work

Sarma and Mall [8] presented a technique applicable for system testing rather than testing the behavior of a single object. The test generation methodology uses information from use cases, sequences and state diagrams. STRAP generates one test suite for one criterion (to exercise elementary paths), while STAGE has the capability of generating all possible distinctive test suites, utilizing one of four different traversal techniques. ParTeG (Partition Test Generator) is an Eclipse plugin that combines control flow coverage criteria and boundary based coverage criteria. However, ParTeG does not target satisfying a specific test criterion, e.g., transition coverage, or transition path coverage. Instead, it aims at increasing the test suite fault detection ability. After generating the required test suite, ParTeG measures the effectiveness based on mutation analysis [16]. In contrast to STAGE, the user of ParTeG has no control over the selected test goals of the generated test suites. Instead, ParTeG dynamically adapts the test goals to increase the mutation score [15]. In other words, unlike STAGE, one does not have the choice to trade-off quality for cost minimization, neither does one have the ability to compare the different outcomes of using different types of generated test suites.

Kansomkeat and Rivepiboon automatically generate test cases from UML state diagrams. This method transforms a UML diagram into a Testing Flow Graph (TSG), which is a simple structure diagram that flattens the structure of the states and reduces the complexity of the UML diagram. Then, the TFG is used to generate test cases that satisfy the two criteria of state and transition coverage. Finally, the effectiveness of the test cases is proved using mutation testing on simple test cases with a limited number of mutations (26 mutants) [12]. This differs from STAGE, which generates alternative (instead of only one) adequate test suites for a criterion.

AGeTeSC constructs then traverses a state diagram of a given object to select predicates transforming them to source code, then constructs an EFSM from the source code, and generates the test cases from that EFSM. The authors used ModelJUnit [18] to prove the effectiveness of their approach [10]. STAGE differs as it depends on the state diagram specification not the already implemented object, which allows the oracle to test the actual code against the specifications. In addition, AGeTeSC provides one test suite for each traversal type (either depth or breadth traversal), while STAGE provides all possible test suites for the chosen traversal technique.

Generação Automática de Casos de Teste Baseada em Statecharts (GTSC) [19] is an environment that allows the automatic generation of test cases from FSM models and Statecharts. GTSC supports the switch cover, Distinguished Sequence, and Unique Input Output criteria for an FSM [2], and transition coverage for Stateccharts. GTSC takes as input a PcML specification and uses a tool to generate the corresponding FSM in XML. It then uses another tool ConData[3], that is tailored for testing

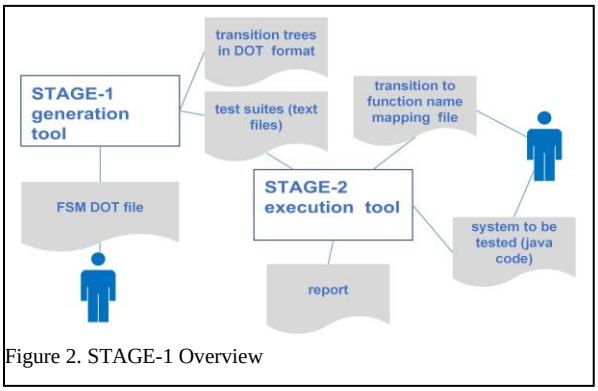

Figure 2. STAGE-1 Overview

communication protocols, to generate the test cases. It implements the all-transition criterion for its control part, while applying syntax testing and equivalence partitioning data criteria. Contrary to STAGE, It generates one adequate test suite.

Transformation-based tool for UML-based testing (TRUST) [13] supports the all-transitions, and all-round-trip-paths criteria and generates an all-round-trip-paths adequate test suite slightly differently from STAGE. Since the all-round-trip-paths criterion requires covering all paths that starts and ends in the same state, the path from the initial state of the FSM to the state where the round-trip path starts also needs to be identified. We call this part of the test path the "prefix". While TRUST uses depth first search to reach this node, STAGE finds the shortest path to minimize the contribution of the *prefix* to the effectiveness of the test suite (e.g., mutation score) to objectively compare different criteria.

The paper in hand is concerned with the test suites generation functionality rather than the effectiveness of the test suites themselves when executed, nevertheless, the generated test suites have been executed using the second tool in STAGE tool chain (STAGE-2) and a tool was used to measure fault detection effectiveness.

Some researchers suggest methodologies upon which automation techniques could be founded, without providing actual implementation of the suggested methodologies [1, 9]. Others develop automation tools such as STRAP [8] and ParTeG [16]. Table 1 summarizes how STAGE compares to the surveyed tools in terms of

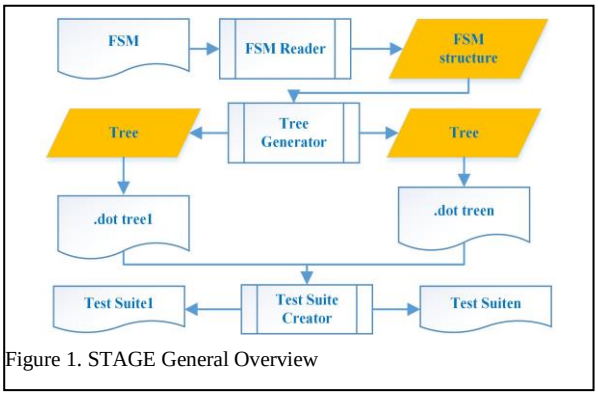

Figure 1. STAGE General Overview

input, supported criteria, and output. To our knowledge, STAGE is the only available tool that gives the user control to choose among different sets of test generation algorithms, that satisfies different criteria, as well as generates all possible adequate test suites for one chosen generation algorithm/criterion. The set of features provided by STAGE enables the user to reach a balance between cost and effectiveness. It also allows comparing the results achieved by different test suites for drawing general conclusions on which type of test suites is suitable for specific categories of case studies.

V. THE PROPOSED SOLUTION

Figure 1 is a high-level depiction of the two main phases of STAGE tool chain, their inputs and outputs and how they are related, STAGE-1 only being the focus of this paper.

Step one of the automation solution generates test suites for a given state diagram. Given a graph description language (DOT) file of the state diagram representing the system to be tested (top left of Figure 2), the tool reads the file and transforms it into an internal representation of the FSM graph. As shown in The outputs produced by STAGE-1 are: , spanning trees of the input graph are generated based on one of four algorithms. Other algorithms can easily be plugged into the current implementation.

For one input state diagram, there is one or more generated traversal trees using either the breadth first, depth first, round trip, or random traversal algorithms. Those traversal trees are saved in DOT files, one file per tree (test suite). Finally, each of the generated trees are converted into a collection of test cases, where each test case is one complete path in the tree starting by the root node of the tree (i.e., initial node of the state diagram) and ending at one of the leaf nodes.

The outputs produced by STAGE-1 are:

1. A collection of (DOT) files representation of generated trees where each file represents one spanning tree produced by applying the chosen algorithm. In the case of the depth first traversal and breadth first traversal, more than one (DOT) tree file is typically generated; our implementations produce all the possible traversals, given a specific stopping criterion. In the case of a random traversal of the FSM graph, we obtain a constant number of distinctive (DOT) files.

2. A collection of text files where each file has a list of test cases. Each test case maps to a path in a tree. The test cases in one file collectively form one possible test suite.

A. Algorithms

1) Depth First Traversal

Depth First Traversal is a tree and graph traversal algorithm which extends the current path as far as possible before backtracking to the last choice point and trying the next alternative path. When traversing a graph, one must decide how to handle cycles. STAGE implantation of depth first traversal handles the cycles by not branching a

node more than once. This means that the stopping criterion is hitting either a leaf node (i.e. a node with no outgoing edges), or a node that has been branched before during the current traversal. Once the algorithm reaches a node that has been branched before, this node gets marked for having more than one incoming edge. Such nodes are called, by STAGE, multiple incoming edges nodes (*MIEN*).

STAGE implementation of the depth first traversal ensures all the edges and consequently all the nodes of the FSM are exercised: each tree is all-edges and all-nodes adequate.

Error! Reference source not found. shows the pseudo code of STAGE-1's depth first traversal algorithm that generates all possible depth first trees of a given directed graph with a known initial node. (Note that if the FSM has several initial nodes, it is easy to create one unique pseudo node that becomes the unique initial node for the entire FSM.) The algorithm relies on the observation that the presence of nodes with more than one incoming edge results in having more than one depth first traversal tree, since this implies having more than one path to reach this node from the initial node. However, having more than one outgoing edge for a single node does not have the same implication, since the different outgoing edges can be directed to different destination nodes and this does not necessarily indicate different traversal trees. The first step of the algorithm generates one initial depth first traversal tree: DFS (G, $N_{initial}$). For each MIEN node, there is a list of possible paths leading to that node. A path is a series of edges, each connecting two nodes together, which would define the route taken from the source node until the destination

The STAGE algorithm creates all possible paths combinations that preserve the properties of a depth first traversal. One *combination* of paths is a collection or a set of paths, where each path leads to one of the MIEN nodes in the graph. For each combination, the algorithm clones

```
Algorithm CreateDepthForest
Input: A directed graph G with a root node N_initial
Output: All possible depth first trees of G in TreeList
T_0 ← DFS (G, N_initial)
Clist ← CreateDepthPathsCombinations (G)
i ← 0
for each C ϵ Clist
  T_i ← Clone(T_0)
  QuickSort (C)
  for each Path p in C
    for each Edge e in p
      N_1 ← GetMIENPosition (e)
      N_2 ← GetMIENPositionInInitialTree (N_1)
      SwitchSubtrees (T_i, N_1, N_2)
    endfor
  endfor
  TreeList = TreeList + T_i
Endfor
```

Figure 3 Depth Frist Traversal Pseudocode

the initial tree: Clone(T_0). The parents of each MIEN node must be placed in the target positions before attempting to move the descendants in the desired location. Otherwise, when moving the subtrees rooted at the parent, the locations of the descendants will get reshuffled. Since, by definition, the path to reach a parent node from the root of the graph has to be shorter than any of the paths leading to one of the descendants of that parent (a property of a depth first traversal), the algorithm uses a "quick sort" technique to sort each combination **C** from the shortest path to the longest path. Now that each node has a set of known possible positions, the algorithm iterates on the edges of each paths, gets the destination node of each edge (a MIEN node), and relocates the position of that node (N_2) in the initial tree according to its new suggested location (N_1) by the path generated in the previous step.

This algorithm has been tested using many case studies of different sizes. The largest FSM or graph that the algorithm has been used against was a VCR simulation that has 17 states and 65 transitions. The depth first traversal resulted in 101,947 distinctive test suties. The number of resulting trees for this case study as an example emphasizes the need to automate the process as it is obviously impossible to perform such traversals manually.

2) Breadth First Traversal

The breadth first graph traversal algorithm begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores its unexplored neighbor nodes and so on [6]. This means that all nodes at the same horizontal level are branched first before moving to the lower horizontal level representing the children of the branched nodes.

As in the depth first algorithm, branching stops when reaching either a graph leaf node or a node that has been already branched from. The breadth first traversal algorithm covers also all the edges as well as all the nodes of the FSM graph: generated test suites are all-edges adequate. The structure of this algorithm is similar to that of **Error! Reference source not found.**. For the VCR case study mentioned earlier, the breadth first algorithm results in 24 distinctive trees.

3) Round Trip Algorithm

Mouchawrab and colleagues showed experimentally that trees produced using a BFS or DFS algorithm exercise some round-trip paths entirely while other round trip paths are only exercised in a piecewise manner [14].

STAGE, on the contrary, includes an implementation of a round trip paths graph traversal algorithm that ensures all the round-trip paths in the FSM are exercised in their entirety. The first part of the algorithm generates all the round-trip paths that start in each node in the FSM graph. The second part of the algorithm searches, for each of the nodes starting a round trip path, a shortest path from the root to that node and adds this shortest path as a prefix for all generated round trip paths of this node.

In contrast to the previous two algorithms, the round-trip path does not necessarily subsume the all edges criterion nor the all nodes criterion [9]. The resulting test suite will only contain edges and nodes composing part of a round trip path or that are necessarily to construct a prefix for any of the round-trip paths.

4) Random Traversal Algorithm

The random algorithm results in a distinctive constant number of random traversal trees of the input graph. It adds the initial node of the state machine to the resulting tree, randomly picks one of the outgoing edges of that node, then it adds the destination node of the chosen edge to the tree. At any stage of the traversal, an edge of the collection of all the outgoing edges of the nodes that has been already added to the resulting tree is randomly chosen to be added to the tree together with its destination node.

The algorithm stops when all the edges and the nodes of the input graph are added to the tree. Hence, the resulting tree is all edges and all nodes adequate.

B. Validating Results

The results of STAGE have been validated in three different ways on multiple FSM examples:

1. Uniqueness of the generated trees: A simple automatic tree comparison tool has been developed to verify that the resulting trees are distinctive. When STAGE traverses an input graph, using one of the traversal algorithms, the comparison tool is used to verify that all trees are unique. The tool iterates on all trees and compares all the nodes to their equivalent nodes in other trees that have been processed before. The comparison includes the incoming transition and all outgoing transitions for each node in the tree.
2. Correctness of traversal trees: Validating the correctness of the resulting trees is performed by picking random samples of the resulting trees and manually validating that they conform to the criteria used, and while complying with the original state machine specification fed to the system.
3. Completeness of the tree generation solution: The algorithm has also been tested using smaller size input graphs to manually validate that the resulting set of trees is complete.

VI. CASE STUDIES

STAGE has been tested and used to perform several experiments on several case studies among which is an Automated Teller Machine (ATM) simulator, a Cruise Control system, a Videocassette Recorder (VCR), and one (ordered set) data structure. The size of the FSM of each case study vary as shown in . This paper focuses on the Ordered Set case study. This case study system has been used by others [7], [14] and models a resizable ordered set of integers: the overall size of the set should not exceed a maximum set size, only a fixed number of resizes is allowed. Figure 4 is the state diagram of the ordered set system.

Table 2 Output Matrix

Case Study Name		Ordered Set	VCR	Cruise Ctrl	ATM
No. of States		5	17	5	10
No. of Transitions		17	65	29	22
BF Trees	*Trees*	2	24	3	9
	Test Cases	26	48	24	13
	Time (ms)	1	6	<1	<1
DF Trees	*Trees*	7	101,947	3	27
	Test Cases	26	48	24	13
	Time (ms)	5	156,475 (2mn36s)	4	20
Round-Trip	*Test Cases*	26	1406	46	15
	Time (ms)	16	798	15	<1
Random Tree	*Test Cases*	26	48	24	13
	Time (ms)	<1	31	<1	<1

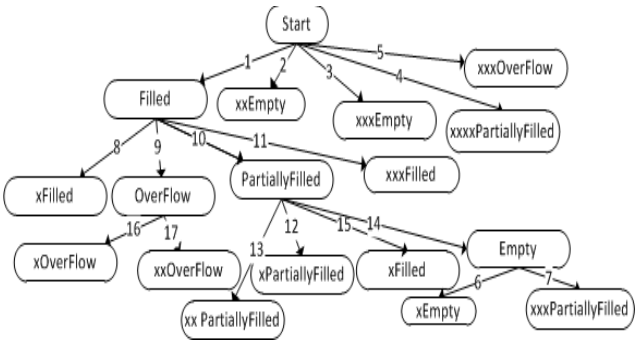

Figure 5 shows one of the trees generated by depth first where a state name prefixed with a "xx…" indicates the state has been hit before by the current path or a previous path along the traversal process and therefore the path terminates at this end node; However, this state has to be reached again through another path in order to cover the other transitions leading to this node.

compares the four case studies in terms of number of states, transitions, generated traversal trees and execution time for each criterion. As evident from the table, as the number of states and transitions of the input FSM increases, the number of generated trees increases in an exponential manner and hence the algorithm execution time. For example, while generating seven depth first

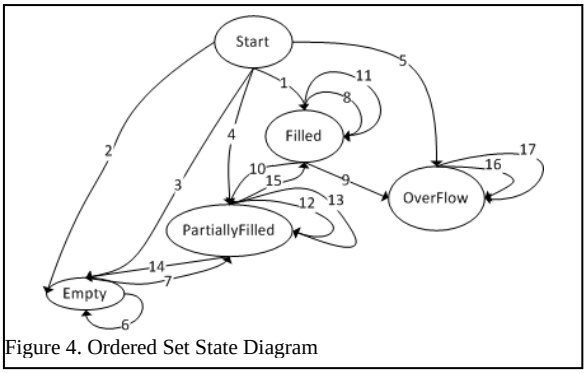

Figure 4. Ordered Set State Diagram

traversal trees for the ordered set case study with a total of

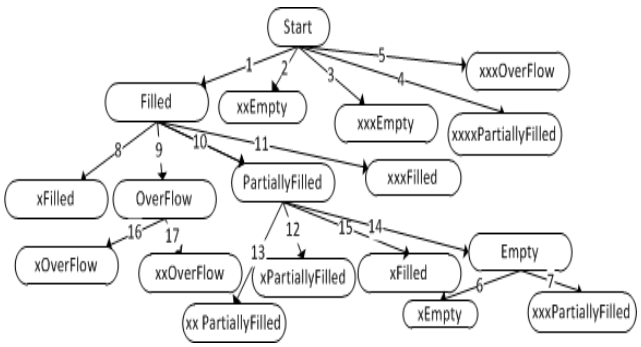

Figure 5. Ordered Set Depth First Traversal Tree

five states and eighteen transitions takes 5 milliseconds, generating 101,947 trees for the VCR FSM that is composed of 17 states and 65 transitions takes around 2.5 minutes. The number of generated trees using each algorithm is highly dependent on the FSM structure. For example, the highly connected and cyclic cruise control leads to three breadth traversal trees that are identical to the trees generated using depth first traversal. However, as the depth traversal performs more complex checks to guarantee that the depth first properties are not violated, the algorithm is more time consuming than the breadth algorithm.

VII. CONCLUSION

The tool chain presented in this paper aims at helping researchers as well as practitioners.

The tool chain is divided into two main subsystems; the test suite generation subsystem presented in this paper and the test driver/test oracle subsystem. The test suite subsystem accepts an FSM diagram that specifies the behavior of the system under test as an input. The tool then transforms the FSM provided into all possible traversal trees based on a chosen algorithm. The chosen algorithm can be a depth first, breadth first, round trip, or random traversal. We note that the tree generation algorithm can be changed: our tool chain is a framework; and therefore, the number of generated trees can vary. Since we are interested in providing tool support to facilitate experiments, our DFS and BFS tree generators generate all the possible traversals. Then, the generated trees are converted into test suites, where each tree corresponds to one test suite. The test suite is a set of test cases that represent all the paths present in the resulting tree.

To the best of our knowledge, STAGE is the only tool that aims at automatically generating different types of trees for an adequacy criterion from a state diagram specification for the purpose of comparing the effectiveness of the resulting test suites. The framework supports the definition of all the possible test suites for a specific criterion. The tool's functionality and use has been tested using several case studies.

Expansions to the tool could include encompassing more tree traversal algorithms for generating more diverse types of test suites as well as supporting additional adequacy criteria, such as Unique Transition Paths, that have been defined in the literature. Finally, testing the tool chain against more case studies with different characteristics to evaluate the performance and look for areas where the execution time could be optimized.

REFERENCES

[1] Briand, L.C., Labiche, Y. and Cui, J. 2005. Automated support for deriving test requirements from UML statecharts. *SoSyM*. 4, 4, 399–423.

[2] D. P. Sidhu and T. . Leung, "Formal methods for protocol testing: A detailed study," *IEEE TSE*, vol. 15, no. 4, pp. 413–426, Apr. 1989.

[3] E. Martins, S. B. Sabiao and A. M. Ambrosio, "ConData: a tool for automating specification-based test case generation for communication systems," *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, 2000, pp. 10 pp. vol.1-.

[4] H. Hemmati, A. Arcuri and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM TOSEM*, vol. 22, no. 1, 2013.

[5] IEEE Draft Standard for Software and Systems Engineering--Software Testing--Part 1: Concepts and Definitions, in *ISO/IEC/IEEE P29119-1-FDIS, March 2013* , vol., no., pp.1-66, May 1 2013.

[6] J. Z. Z. Gao, J. T. S. H, Y. Wu, H. . J. Tsao, J. Tsao, and T. . H. Jacob, *Testing and quality assurance for component-based software*. Boston: Artech House Publishers, 2003.

[7] M. Khalil and Y. Labiche, "On the round trip path testing strategy," *2010 IEEE 21st International Symposium on Software Reliability Engineering*, Nov. 2010.

[8] M. Sarma and R. Mall, "Automatic generation of test specifications for coverage of system state transitions," *Information and Software Technology*, vol. 51, no. 2, pp. 418–432, Feb. 2009.

[9] P. Ammann and J. Offutt, *Introduction to software testing*. New York: Cambridge University Press, 2008.

[10] R. Swain, V. Panthi, P. Kumar Behera, and D. Prasad Mohapatra, "Automatic test case generation from UML state chart diagram," *International Journal of Computer Applications*, vol. 42, no. 7, pp. 26–36, Mar. 2012.

[11] R. V. Binder, *Testing object-oriented systems: Models, patterns, and tools*. Reading, MA: Addison-Wesley Educational Publishers, 1999.

[12] S. Kansomkeat and W. Rivepiboon. 2003. Automated-generating test case using UML statechart diagrams. In *Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology* (SAICSIT '03), 296-300.

[13] S. Ali, H. Hemmati, N. E. Holt, E. Arisholm, and L. C. Briand, "Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies," Simula Research Laboratory, Technical Report (2010-01)2010.

[14] S. Mouchawrab, L. C. Briand, and Y. Labiche, "Assessing, comparing, and combining Statechart- based testing and structural testing: An experiment," *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, Sep. 2007.

[15] S. Weißleder and D. Sokenou, "ConSequence — model-based testing with state machines and Concatenated sequence diagrams," *Softwaretechnik-Trends*, vol. 32, no. 1, pp. 4–5, Feb. 2012.

[16] Stephan, "ParTeG ,". http://ParTeG .sourceforge.net. Accessed: 2015-12-11.

[17] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE TSE*, vol. SE-4, no. 3, pp. 178–187, May 1978.

[18] "The ModelJUnit test generation tool,". http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/. Accessed:

Dec. 8, 2016.
[19] V. Santiago, N. L. Vijaykumar, D. Guimarães, A. S. Amaral, and É. Ferreira, "An environment for automated test case generation from Statechart-based and finite state machine-based behavioral models," *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, 2008.