SymTest : A Framework for Symbolic Testing of Embedded Software

Sujit Chakrabarti
International Institute of Information Technology
Bangalore, India
sujitkc@iiitb.ac.in

Ramesh S.
General Motors
Warren, MI
USA
ramesh.s@gm.com

ABSTRACT

In test case generation methods based on symbolic testing and/or model checking, the primary emphasis is on covering code/model elements and not on optimising test sequence length. However, in certain domains, e.g. embedded systems, GUI, networking software, testing process may involve interaction with other physical subsystems, possibly remotely situated. Thus, test sequence length may have important implications on the cost of testing. In this paper, we present SymTest, a novel framework for test sequence generation for testing embedded systems. SymTest selects *good* control flow paths so as to generate shorter test sequences. In case of unsatisfiability, SymTest explores the neighbouring paths using backtracking and heuristics. SymTest is distinctive w.r.t. other related methods in its attempt to generate shorter test sequences while searching for feasible paths. The other novelty is that SymTest allows plugging in heuristics in a flexible way, a feature because of which we call SymTest a framework and not an algorithm. Part of SymTest's power is in its extensibility to seamlessly accommodate more heuristics, thus enhancing its ability to generate shorter test sequences with economy of effort. Our experiments with SymTest show that SymTest achieves significantly shorter test sequences, with comparatively higher test coverage, as compared with other methods.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing tools, symbolic execution; C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms

Algorithms, measurement, experimentation

Keywords

Test generation, embedded systems, symbolic execution

1. INTRODUCTION

To the present day, testing continues to remain the primary method of quality assurance in software development. Testing has been an area of active research for several decades. Research has given software engineers a wide choice of testing tools for automatic execution and generation of tests. The primary emphasis of most research in testing has to been on coverage. The prime objective of test generation algorithms is to generate test data which *somehow* cover the target elements of the model/representation of the software system which is used by the algorithm. *Somehow*, because once a test case is demonstrated to produce the expected coverage, its *quality* is not reviewed. For example, the test input length is usually not considered by test generation algorithms. In many scenarios, test input length has significant implication on testing cost. It has direct implication on the test execution time. In embedded systems, involving interaction with the environment and physical subsystems, unnecessarily long test sequences may waste time or result in wear and tear of mechanical and electrical components. Another example is GUI testing, where long test sequences may hold up the time of a test engineer, particularly when interactive test execution is involved. In distributed systems, excessively long test sequences may block costly network resources.

In this paper we present a new method of structural test input generation called *SymTest*. SymTest uses analysis and heuristics to optimise the length of test sequences generated. SymTest is similar to a number of other methods in that it uses a combination of symbolic execution and constraint solving. For example, concolic testing and related approaches begin with a randomly selected control flow path and then systematically exploring its neighbourhood. CBMC based approaches use bounded model checking to do exhaustive exploration of execution paths to a particular bounded depth. Each method is distinct based on the way it explores the execution paths of a program under test.

SymTest is distinct from all the related methods in the emphasis it places on finding an optimised test path. SymTest's approach can be summarised as follows:

1. Start by identifying the *syntactically optimal path* that covers the set of target edges. If this turns out to be feasible, output the corresponding test sequence.
2. On encountering infeasibility, explore 'neighbouring' paths. For this, interleave *backtracking* and *heuristics*.

In our case studies, we have found that, for a significant number of systems we considered, the step 1 above is successful. This means that SymTest indeed succeeds in computing the optimal test sequence for these systems. We have designed a number of heuristics to go into the step 2 above. We have found that heuristics are fairly successful in generating test sequences which are not very far from the syntactic optimum, though their theoretical optimality can not be guaranteed. There is a possibility of designing a very large number of similar heuristics. With each added heuristic, SymTest's power is highly likely to increase. Therefore, we have architected SymTest to facilitate plugging in of heuristics. Finally, when all else fails, backtracking approach is an effective last resort, in a spirit similar to that employed by other techniques like concolic testing. It exhaustively searches all paths to execution depth d_{max} provided by the user.

1.1 Contributions

The contributions of our work are as follows:

- A novel characterisation called *acyclic core control flow graph* of embedded software systems which we use as an input to our test sequence generation approach
- A symbolic execution + constraint solving test generation algorithm which differs from other related work on the following:
  1. It covers the edges as specified by the user, as opposed to trying to cover all edges or all paths. This is a more generalised version of edge coverage.
  2. It tries to optimise the test sequence length while providing the required coverage.
  3. It controls the path that is traversed during test execution as opposed to other related methods where in the algorithm usually has little control on the actual path that is traversed.
- An extensible plug-in architecture which allows seamless integration of additional heuristics
- Some example heuristics
- Experimental results which highlight that the test sequences generated by SymTest are indeed shorter than other related approaches

The rest of the paper is organised as follows: In section 2.1, we present a motivating example. In section 3 and 4, we present the details of SymTest algorithm. Section 6 briefly describes the scheme used to apply SymTest algorithm to test Simulink models. We present an example of test generation using SymTest in section 5. We present related work and comparison of our work with them in section 8. Section 10 concludes the paper with a summary and scope of future work.

2. THE PROBLEM

2.1 Motivating Example

Consider the control flow graph of an example system shown in figure 1(a). The edges marked in bold are the target edges, the edges we want to cover during testing. Consider how concolic testing would generate a test sequence for this system. The first path executed during concolic testing is due to randomly generated input. Assume that it leads the execution to the path highlighted in the first graph in figure 2(a). The sequence in which the coverage will happen from this point on is shown in the second and the third part of figure 2(a). Therefore, concolic achieves coverage of the target edges in three iterations.

Bounded model checking (BMC) can be used to generate test sequences for the above system. BMC may generate a different test sequence depending on the unrolling depth d chosen by the user. However, internally it will potentially explore all the path up to the d. This is exponential in the number of decisions up to d. Hence, BMC employs a potentially expensive method to compute the test sequence. Also, the probability of the optimal test sequence being generated is no greater than that of any other valid test sequence within depth d.

The Simulink system shown in figure 1(b) has two if blocks both having the same input u. It is, in fact, the system from which the control flow graph of figure 1(a) has been generated. Suppose that we are trying to achieve data flow coverage which will happen when the value of u flows to all the outputs v1 (u→v1), and v3 (u→v3). This coverage, in turn, is analogous to the branch coverage as shown in figure 1(a). A randomly generated input, say u = 1, will cover branch u→v2 and u→v4. However, the probability of covering u→v3 using random-testing is infinitesimally small, and on an average would take a very long test sequence to achieve. Another alternative would take each target at a time and try to cover that using backward slicing of the model and symbolic evaluation [26]. It would target u→v1 first. This may give u = 11. Then, it will target u→v3. For this, it will generate u = 20. Since this would also cover u→v1. But, the first input value is redundant, and the simulation is one cycle too long w.r.t the required.

In contract to all the above cases, SymTest would generate a test sequence [u = 20]. The coverage happens as shown by the bold edges of figure 2(b). The test sequence length is 1! The same test sequence also achieves the desired coverage for the Simulink model shown in figure 1(b). This shortening of test sequence could be achieved because some conditions can be conjuncted to form one formula, potentially reducing the length of the test sequence generated.

2.2 Terms and Concepts

In order to ensure consistent use of terms, we present some terms and their definitions here.

DEFINITION 1 (ATOMIC VALUE). *We define an* atomic value *as a data value assignable to a variable at any point in time during the execution of the program under test (PUT).*

Note: The type of the data value could be a scalar or a vector.

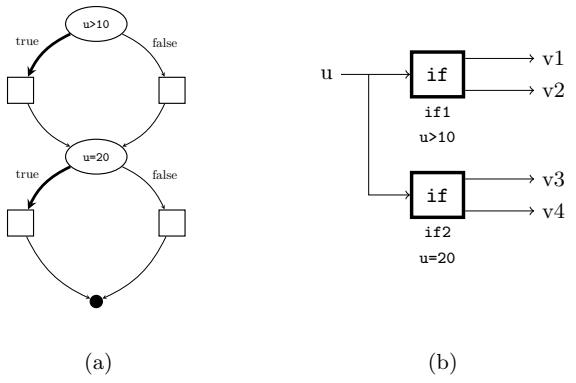

Figure 1: Motivation examples: (a) CFG of a program under test; (b) A fragment of Simulink model

Simulation Step	Input Variable	
	u	v
step 1	[4, 3]	7
step 2	[1, 2]	8

Table 1: An example of a test sequence

Definition 2 (Input Vector). *We define* input vector *as the collection of atomic values assigned to the input variables by the environment (test driver) at any time instant during the execution of the PUT.*

Definition 3 (Test Sequence). *Test sequence is defined as the complete sequence of input vectors to the PUT as given in one complete run of a test.*

Note: *Test sequence* and *test input* are used as synonyms in the context of this work.

Example: Consider a system with two input variables u : int array and v : int. During a testing process, consider the simulation run shown in table 1. The values [4, 3], [1, 2] 7 and 8 are atomic values. There are two input vectors: [[4, 3], 7] and [[1, 2], 8] each corresponding to a single simulation step. [[[4, 3], 7], [[1, 2], 8]] is a test sequence.

2.3 Acyclic Core

We present SymTest's function on a specific subclass of control flow graphs which have the structure shown in figure 3. The CFG has a block I at the beginning consisting of some initialisation computation that happens only once in one execution. I may have decisions and loops. This is followed by an infinite loop whose body is a computation block summarised by two blocks: *inp*, the block in which the input variables get new values from the environment, and the composite node FP. FP is an acyclic piece of computation, which means that it does not have any loops. Structurally, it is a *directed acyclic graph.* Thus, the body of the outer loop is a piece of acyclic computation. We call this the *acyclic core* of the CFG, and the CFG is said to have an acyclic core.

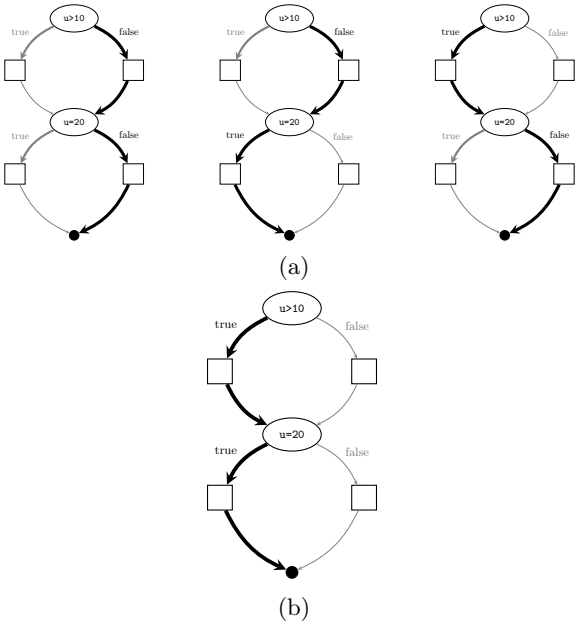

Figure 2: Test sequence: (a) Generated using concolic testing; (b) SymTest

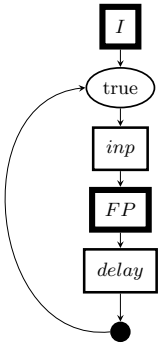

Figure 3: A control flow graph with Acyclic Core

It turns out that there are specific non-trivial classes of systems, particularly data-flow systems like Simulink models or digital circuits, which, when translated into a control-flow form, would be an acyclic core CFG. In fact, fairly complex Simulink models – even those with delays, feedback, nested sub-systems and even multiple clocks – can be represented using an acyclic core CFG.

2.4 Problem Definition

Given a program in the form of its control flow graph(G) and a set of edges in it called T, we wish:

1. to generate a test sequence covering all the edges in T.
2. to keep the length of the generated test sequence as small as possible.

Note: We use *edge coverage* as the test coverage criterion in this paper. The technique developed in this work is suitable

to be tuned for other test criteria as well.

3. SYMTEST ALGORITHM

3.1 Overall Approach

Algorithm 1 SymTest Algorithm

```
1: procedure SYMTEST(G, T)
2:     s ← empty stack
3:     repeat
4:         p ← FINDPATH(s)
5:         if p is satisfiable then
6:             return computed test sequence
7:         else
8:             pre ← LONGESTSATISFIABLEPREFIX(p)
9:             UPDATESTACK(s, pre)
10:            PUSH(s, (e, true)) where pre ⌢ e is unsatisfiable.
11:            if heuristics succeed then
12:                return computed test sequence
13:            else
14:                s ←BACKTRACK(s)
15:            end if
16:        end if
17:        this
18:    until top(s) = top of loop
19:    announce failure
20: end procedure

21: procedure UPDATESTACK(s, p)
22:    push (e, true) to s for all decision edges e in p following the part of p already in s
23: end procedure
```

SymTest algorithm uses symbolic execution combined with constraint solving to generate test sequences for a system under test guaranteeing coverage of a given set of edges. SymTest attempts to keep the length of the test sequence as small as possible. The problem of generating the shortest possible test sequence is undecidable in general. SymTest gets around this problem by using heuristics.

At the top level, SymTest algorithm takes as input the program under test in the form of its control flow graph(CFG) and generates a test sequence that guarantees certain coverage criterion over the CFG. An additional input is a set of edges in the CFG identified as targets that the SymTest algorithm needs to cover.

The pseudo-code for the SymTest algorithm is presented in algorithm 1. At the outset, SymTest uses the algorithm FINDPATH to compute an *optimal syntactic path* P through the CFG G which covers all the target edges listed in set T. By *syntactic*, we mean that this path is indeed a path in G in a strictly graph theoretic sense, but may or may not correspond to a feasible control flow path through P. By *feasibility* of P, we mean that there exists at least one test sequence which will lead an execution of G through P. By *optimal*, we mean that among all paths through G which cover T, P makes the least number of iterations through G's main loop. Assuming that P is feasible, SymTest does a *symbolic execution* of G along P. This gives a *symbolic execution trace*. From this, SymTest extracts the *path predicate* corresponding to P. Just to recapitulate, a path predicate is logical formula whose atoms are symbolic variables generated during symbolic execution and which is created as a conjunct of the symbolic expressions to which all the decision conditions that belong to P evaluate to during the symbolic execution of P. Further, this path predicate is input to an SMT solver. A successful solution of the path predicate by the SMT solver indicates that P is indeed feasible. The solution also provides concrete valuations of the symbolic variables of the path predicate. These concrete values can be directly used to compose a test sequence. This test sequence is a valid input to lead execution of G through P. The symbolic execution based test generation is a fairly well-known technique of test generation. Therefore, in the interest of brevity, we have encapsulated it in lines 5, 6 and 12 of algorithm 1.

The call to the SMT solver can, however, fail. This indicates that P is a *semantically* infeasible path. This necessitates exploration of G for other paths which are feasible. For this, firstly, SymTest computes LSP, the *longest satisfiable prefix*(LSP) of P. An LSP of P is a prefix of P, is a feasible path, and the path obtained by extending LSP with its following edge in P is infeasible. It is this LSP whose neighbourhood is explored for the feasible path which also gives us the coverage of T. For this, SymTest employs two approaches: *backtracking* and *heuristics*.

Backtracking systematically explores the neighbouring paths. On the positive side, this exploration is *exhaustive*, i.e., given a depth d_{max} of exploration (measured as the number of iterations of the main loop of G), this will eventually end up exploring all the syntactic paths up to depth d_{max}. Therefore, if a solution exists among these paths, backtracking is guaranteed to find it. However, on negative side, the number of such paths may be exponential in d_{max}. Therefore, even for moderately large values of d_{max}, backtracking may end up taking unacceptably large amount of time to complete the exploration. A judicious choice of d_{max} is therefore very important to make backtracking work. Further, backtracking is exhaustive only within d_{max}. A feasible path deeper than d_{max} will not be discovered, even if it covers T and is optimal. We give further details of the backtracking algorithm in section 4.1.

To get around the first disadvantage of backtracking, SymTest uses heuristics. Heuristics may come back with quick results without having to do the hard work that backtracking may have to, sometimes unnecessarily. However, by their definition, heuristics do not come with any guarantees of exhaustiveness. That is, a failure of heuristics to discover a solution may not be a reliable indication of the absence of such a solution. Having a large number of good heuristic will enhance the probability of success in finding a test sequence with heuristic. SymTest's architecture supports a pluggable use of heuristics. We give further details of heuristics in section 4.2.

In a sense, SymTest prefers heuristics to backtracking, revealed by the fact that, on encountering infeasibility, SymTest first tries out all the heuristics it has. Only when none of them succeeds in giving a test sequence is backtracking given a chance. In this sense, backtracking is used by SymTest as a last resort.

3.2 Finding the Syntactic Test Path

Algorithm 2 findPath Algorithm

```
1: procedure FINDPATH(G, T, n, d)
2:     if T = ∅ or d = d_max then
3:         return EMPTYPATH
4:     end if
5:     A ← FINDLONGESTACYCLICPATH(G, T, n)
6:     return A ⌢ e_b ⌢ FINDPATH(G, T \ (T ∩ A), src(G), d + 1)
7: end procedure

8: procedure FINDLONGESTACYCLICPATH(G, T, n)
9:     if d = d_max then
10:        return EMPTYPATH
11:    end if
12:    acp ← ∅
13:    for all e ∈ outgoingEdges(n) do
14:        add e ⌢ FINDLONGESTACYCLICPATH(T \ e, hd(e)) to acp
15:    end for
16:    maxs ← all paths p ∈ acps such that ∄p' ∈ acps where p ≠ p' and |T ∩ p'| > |T ∩ p|
17:    return a path randomly selected from maxs
18: end procedure
```

Algorithm 2 shows the pseudo code of the FINDPATH procedure. It takes as input the control flow graph G of the program and gives a syntactic path through it which covers T.

In the process of searching out the shortest path, FINDPATH tries to minimise the number of times the path passes through the outer **while** loop of G. Theoretically, this problem is similar to the problem of finding the smallest set of chains in a partially ordered set which cover all the edges. The problem is a known to be tractable and efficient algorithms exist to solve it.

The algorithm shown in algorithm 2 is a variant we have implemented in the prototype tool. It does not give the most optimal solution in general case. However, for reducible graphs [28] – a restriction valid for a large majority of programs – it still gives the optimal set. In each iteration, the algorithm discovers a new acyclic path from the starting node s (at the top) to the target node t (at the bottom) by calling FINDLONGESTACYCLICPATH procedure. Each path covers some of the edges in T, the set of target edges. In the end of the execution of FINDPATH, it returns the optimal syntactic path covering all the edges in T. This path is obtained by concatenating all the acyclic paths computed, one during each iteration of the main while loop of G.

FINDLONGESTACYCLICPATH sets itself a goal of finding that acyclic path which contains the largest number of edges which originally are in the *targets* set, and which have not been covered in any of the previous iterations. In the current discussion, such a path is called the *longest acyclic path.* The logic of this algorithm is the following: For all nodes n'_i where (n, n'_i) is an edge, if acp_i is the longest acyclic paths starting at n'_i, then the longest acyclic path at n is the longest among the paths $(n, n'_i)^\frown acp_i$ ($^\frown$ is the concatenation operator). Here, we define the *length* of each path acp_i as the number of target edges in it.

For the class of acyclic core CFGs that can be represented using reducible graphs, the implementation of FINDPATH shown in algorithm 2 finds a syntactic path which covers all the target edges in shortest number of iterations. However, as explained earlier, the control flow path found by FINDPATH may not be a realistic path. If so, the fact would get revealed during the constraint generation step: the corresponding path predicate will turn out to be unsatisfiable. SymTest must find an alternative control flow path in such a case. Details of how this scenario is handled are given in section 4.

3.3 Constraint Generation

During test execution, the control will flow through the path discovered by FINDPATH algorithm if the inputs given to the program satisfy the *path predicate* corresponding to that control flow path. We use symbolic execution to compute this path predicate corresponding to the control flow path found above.

A *symbolic execution engine* program carries out *symbolic execution* on a program and generates a *symbolic execution tree.* In our case, since the control flow path is already determined by FINDPATH, the symbolic execution proceeds along only that path, outputting a *symbolic execution trace* (SET). Figure 4 illustrates symbolic execution. Symbolic execution of a program in figure 4(a) along the control flow path $e_1e_2e_3e_4e_5e_4e_5e_7$ results in a symbolic execution trace shown in figure 4(b).

From the symbolic execution trace, we compute the *path predicate* corresponding to this control flow path. Path predicate is obtained as a logical formula which is the conjunction of symbolic expressions corresponding to all the branch predicates encountered along the trace. In figure 4(b), the branch predicates have been highlighted in grey. Thus, the path predicate computed is: $(V_1 > 0) \wedge (2V_1 > V_1) \wedge (2V_1 + 3 > 2V_1) \wedge \neg(2V_1 + 6 > 4V_1)$.

3.4 Test Sequence Generation

Next, the above formula is given as an input to a theorem prover (an SMT solver, e.g. Yices). The solver returns concrete values of symbolic variable V_1. For example, the solver succeeds to find a satisfying valuations for the above formula as follows: $V_1 = 3$. This value is finally used as test input values for the corresponding input variables to constitute the test sequence. For example, with the above values, we get the following test sequence: $((a = 3))$.

4. HANDLING UNSATISFIABILITY

Figure 4(c) presents an example of how the control path computed by FINDPATH presented in section 3.2 could compute an unrealistic control flow path. The path predicate corresponding to it is: $(V_1 > 0) \wedge \neg(2V_1 > V_1)$. This is unsatisfiable. This fact gets revealed when we try to solve it using the SMT solver, which declares *UNSAT* (meaning that it is not able to find a satisfying valuation for this formula). Therefore, it is not possible to have a test sequence corresponding to this control flow path.

The problem of finding a satisfiable control flow path covering a set of targets is undecidable. So is the problem

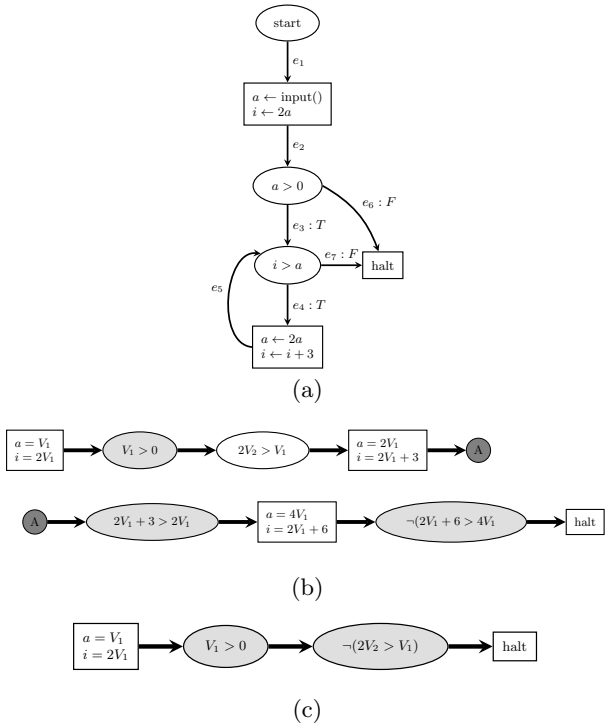

Figure 4: Symbolic execution: (a) A simple program; (b) Symbolic execution trace for path $e_1e_2e_3e_4e_5e_4e_5e_7$; (c) Symbolic execution trace for path $e_1e_2e_3e_7$

of finding satisfying assignments to the input variable that would carry the execution along a particular control flow path. When a computed control flow path turns out to be unsatisfiable, we are faced with the problem of finding an alternative path which is satisfiable. This is, due to identical reasons, undecidable in general.

4.1 Backtracking

Algorithm 3 The backtracking algorithm

```
 1: procedure BACKTRACK
 2:     (e_i, b_i) ← top(s)
 3:     if b_i = true then
 4:         POP(s)
 5:         PUSH(s, (e'_i, false))
 6:         return s
 7:     else
 8:         e ← POP(s)
 9:         if e is the top of the loop then
10:             d ← d − 1
11:         end if
12:         return BACKTRACK(s)
13:     end if
14: end procedure
```

SymTest systematically explores the neighbourhood of the longest satisfiable prefix (LSP) using backtracking. The backtracking algorithm is shown in algorithm 3. The algorithm uses a stack s to keep track of the LSP. Each element of s is a pair $E_i = (e_i, b_i)$ (counting upward starting with the bottom of the stack) where:

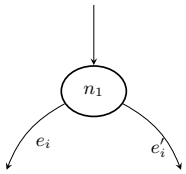

Figure 5: A decision node and its outgoing edges

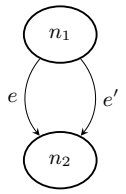

Figure 6: Two decision edges with common source and destination decision nodes

1. e_i is the ith **CFG edge** of the current LSP.
2. b_i is **boolean** indicating whether the other outgoing edge of e_i's decision node e'_i has been explored (see figure 5). $b_i = true$ implies that e'_i has not been explored; b_i implies that e'_i has been explored.

On being called, BACKTRACK examines the top element $E_t = (e_t, b_t)$ of the stack. If the $b_t = true$, BACKTRACK sets it to false and returns. If b_t is found to be false, then it pops off E_t and repeats the examination on the now top-of-the-stack element. This continues until either an element $E_i = (e_i, true)$ surfaces on the top of the stack, or the stack is empty.

4.2 Heuristics

4.2.1 Permute Edge

Consider a pair of edges (e, e') exists in the computed path such that they emerge from the same decision node, and they lead to the same decision node as shown in figure 6. Also assume that both e and e' are there in the syntactic path P computed by FINDPATH, and P turns out to be infeasible. Then, it may result in a satisfiable path by swapping the positions of e and e' in P.

4.2.2 Permute Paths

The path computed by FINDPATH algorithm is a concatenation of the set of acyclic paths $P = \{P_1, P_2, ..., P_n\}$. Any permutation of these paths is also an optimal syntactic path in G. Therefore, new syntactic paths can be generated by merely permuting the acyclic paths in the above set a 'reasonable' number of times (since the number of such permutation is $n!$).

4.3 Breaking the Test Sequence

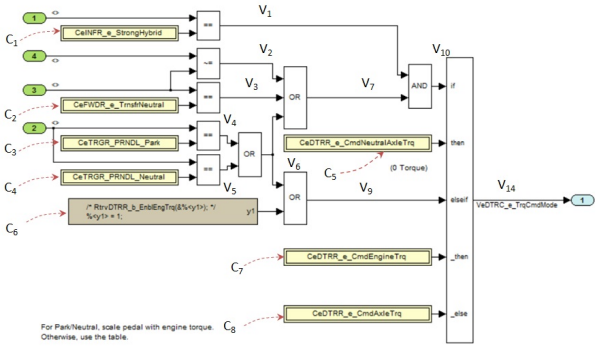

Figure 7: Simulink model from GM case study

Another method of handling unsatisfiability is breaking a single test sequence into multiple. In cases where infeasibility is caused due to loop carried data dependencies, this approach may result in satisfiability. Thus, a single test sequence now gets split into several, each corresponding to a separate test run. This is not semantically equivalent to the case of a single unbroken test sequence, a condition that may be required to be maintained in some cases. However, where a coverage using a single test simulation is not required, this may be an effective and acceptable way to achieve satisfiable and optimal test sequences.

A noteworthy point is that, in absence of loop carried data dependencies, the test vectors are equivalent to independent test cases and the test sequence generated reduces to a test suite. Therefore, in the case of generating independent test cases to achieve branch coverage on a program, SymTest method is applicable as is. The optimisation embodied in SymTest has an equivalent effect on the number of generated test cases in this case, as it has on the test sequence length in the case considered in this paper. Each test sequence in this case would then correspond to a separate test simulation.

The above heuristics are not the representative set, nor do we make any claims of their overall optimality. It is easily possible to think of test cases where they are effective. But we have not carried out detailed experiments to study their effectiveness on real life applications. These heuristics have been presented to serve as examples of how simple techniques may save us the effort of systematic but toilsome method of path exploration using backtracking. We believe that smart heuristics may significantly augment SymTest's power, a direction we intend to direct our research efforts. The bank of SymTest heuristics will benefit from contribution from other researchers. Therefore, SymTest follows an architecture that allows plugging in of heuristics in a seamless and transparent way. This is in the sense that SymTest's main algorithm does not distinguish between one heuristic and another and accesses them through a common interface. SymTest invokes all the heuristics in a particular order and returns the result on success by any of them.

5. EXAMPLE

In this section, we work out a complete example showing how a test sequence is generated from a Simulink model

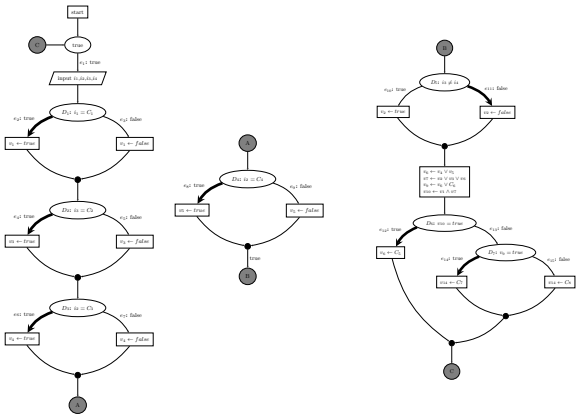

Figure 8: Control flow graph of the Simulink model in figure 7

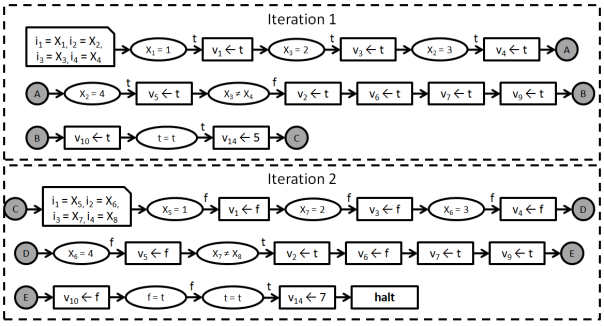

Figure 9: Control flow path computed by findPath for the program in figure 8

	Iteration 1	Iteration 2
i_1	X_1	X_5
i_2	X_2	X_6
i_3	X_3	X_7
i_4	X_4	X_8

Table 2: Assignment of symbolic values to input variables during symbolic execution

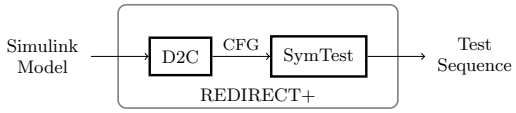

Figure 10: Test Generation for Simulink Models

using SymTest approach.

Figure 7 shows a Simulink model which is a part of a real-world embedded software used by General Motors in its cars. Figure 8 shows the CFG of this model. We have marked the target edges in bold lines. In the first iteration, FINDPATH selects the path p_1: e_1, e_2, e_4, e_6, e_8, e_{11}, e_{12}. In iteration 2, the path selected is p_2: e_1, e_3, e_5, e_7, e_{10}, e_{13}, e_{14}.

On running FINDPATH on this CFG, we get the following control flow path $P = p_1 \frown e_b \frown p_2 : e_1, e_2, e_4, e_6, e_8, e_{11}, e_{12}, e_b, e_1, e_3, e_5, e_7, e_{10}, e_{13}, e_{14}$. Omitting e_1, and replacing the above with corresponding branches, P becomes: D_1, D_2, D_3, D_4, $\neg D_5$, D_6, $\neg D_1$, $\neg D_2$, $\neg D_3$, $\neg D_4$, D_5, $\neg D_6$, D_7. Here, D_i means the true branch of D_i node; $\neg D_i$ means its false branch.

The path predicate for the above control flow path, obtained by symbolic execution along the same, is: $(X1 = 1) \wedge (X3 = 2) \wedge (X2 = 3) \wedge (X2 = 4) \wedge \neg(X3 \neq X4) \wedge (true = true) \wedge \neg(X5 = 1) \wedge \neg(X7 = 2) \wedge \neg(X6 = 3) \wedge \neg(X6 = 4) \wedge (X7 \neq X8) \wedge \neg(false = true) \wedge (true = true)$ where $X1$, $X2$, ..., $X8$ are the symbolic values assigned to the various input variables as shown in table 2. This constraint turns out to be unsatisfiable as $(X2 = 3) \wedge (X2 = 4)$ is unsatisfiable. $(X1 = 1) \wedge (X3 = 2) \wedge (X2 = 3)$ is the longest satisfiable prefix of this formula. In the DAG, the corresponding control flow path is e_1, e_2, e_4, e_6. So, using backtracking, for the first iteration of FINDPATH, we proceed along e_9. The resulting modified control flow path computed is: e_1, e_2, e_4, e_6, e_9, e_{11}, e_{12}, e_1, e_3, e_5, e_7, e_8, e_{10}, e_{13}, e_{14}. The resulting path predicate is: $(X1 = 1) \wedge (X3 = 2) \wedge (X2 = 3) \wedge \neg(X2 = 4) \wedge \neg(X3 \neq X4) \wedge \neg(X5 = 1) \wedge \neg(X7 = 2) \wedge \neg(X6 = 3) \wedge (X6 = 4) \wedge (X7 \neq X8)$ (tautologies omitted). This constraint is satisfiable. The values that Yices solver assigns to the symbolic values are: $X1 = 1$, $X2 = 3$, $X3 = 2$, $X4 = 2$, $X5 = 5$, $X6 = 4$, $X7 = 6$ and $X8 = 7$. The test sequence thus generated is: $((i_1 = 1, i_2 = 3, i_3 = 3, i_4 = 2), (i_1 = 5, i_2 = 4, i_3 = 6, i_4 = 7))$.

6. APPLICATION: TEST GENERATION FOR SIMULINK MODELS

SymTest has a general applicability in test generation for embedded software which can be modelled with an acyclic core CFG. A large majority of Matlab Simulink models, when translated to a control flow graph with an appropriate translation scheme, would be such systems. SymTest can then be used as is to generate test sequences for such systems. We employed the scheme shown in figure 10 to generate test sequences for Simulink models in many of our case-studies. We have developed a method that translates a Simulink model into its corresponding CFG. A lot of additional details is usually present in the code generated by an industrial strength code generator (e.g. Simulink RTW), presumably for the purpose of optimisation and portability. These additional details, however, tend to obfuscate (in a way) the essential structure of the code, rendering it unsuitable for our purpose of test generation. Our method generates generates a *minimalistic* CFG, in the sense that it does not have the additional details mentioned above. This method (shown the block named D2C), whose details are beyond the scope of this paper, is used to obtain a CFG representation of the Simulink model under test. Once the CFG is available, SymTest is used as is in a black box mode to generate the required test sequence.

7. EXPERIMENTAL EVALUATION

We have implemented SymTest as a prototype tool in Java[1]. We conducted experiments to compare SymTest's performance with a concolic testing tool. We used CREST for doing concolic testing. For our experiments, we used the following systems:

1. GM case study (anonymised)
2. Matlab automated cruise control model
3. Elevator controller model developed in our lab

All the above applications are embedded software models developed with MathWorks Simulink, and are parts of real embedded applications. Although, the models above are of moderate scale, they have some interesting characteristics like triggered subsystems and feedback paths. We translated these models into appropriate control flow models using D2C method. We generated tests with both SymTest and CREST. The test generation was done to represent two distinct scenarios:

1. *First time testing.* In this scenario, it is desirable to cover the complete model.
2. *Regression testing.* In this scenario, it is desirable to cover a subset of edges based on an impact analysis done after a change is made to the model.

To run CREST, we wrote the C code following the control flow structure of the models. An infinite main loop (typical of a reactive embedded system) would send CREST into a non-terminating execution. Therefore, we also restricted the number of iterations to $i = 4$ to allow CREST to backtrack.

We considered the number of iterations of the main loop as the common measure for the performance of the testing tool.

[1]SymTest tool and all case studies are available at `https://github.com/sujitkc/symtest`

Application	Branch	SymTest		CREST	
		L_{ts}	Coverage	N_{tc}	Coverage
Cruise Control	20	2	100%	12	70%
GM case study	32	2	100%	20	69%
Elevator Controller	28	3	100%	20	46%

Table 3: Test sequence length L_{ts} during first time testing

	Test Sequence Length	
Target Edge Set Size	$L_{ts} = 1$	$L_{ts} = 2$
$\|T\| = 3$	4	1
$\|T\| = 4$	3	2
$\|T\| = 5$	1	4
	8	7

Table 4: Test sequence length L_{ts} during regression testing.

In case of SymTest, this number directly translates to the test sequence length L_{ts}. For CREST, we considered N_{tc} as the equivalent measure, computed as a product of the number of test cases (shown as 'Iterations' in the CREST output) and the number of iterations i of the main loop in each test case. This is an approximate measure, but by trying out various values of i, we selected the one in which CREST gave the lowest value of N_{tc} (best case) for our case studies, i.e., $i = 4$.

The findings of our experiments for the first time testing scenario are summarised in table 3. It is seen that SymTest consistently succeeds to generate significantly shorter test sequences for all models. Each time, SymTest achieves 100% edge coverage. Further, SymTest generates *test sequences* as opposed to *test cases* generated by CREST. This means that, to use every test case generated by CREST to test the above Simulink models, one has to restart the simulation every time. This is not desirable in the context of testing of embedded systems. On the other hand, a test sequence generated by SymTest belongs to single simulation run. Each input vector in the test sequence takes care of data dependencies from earlier simulation steps. This matches real test scenarios more closely, and should be considered an important strength of SymTest.

For regression testing, we chose cruise control case study. We created 5 sets each of randomly selected 3, 4, and 5 edges. SymTest succeeded to provide 100% target coverage, 8 times out of 15, with a test sequence of length 1. SymTest seeks out the paths which cover the specific edges in the target set through a shorter test sequence. However, CREST is not able to leverage the fact that only a specific set, and not all, of the edges need to be covered: the concept of target edge set does not exist in the case of CREST. This establishes that SymTest does particularly well as a regression testing tool.

8. RELATED WORK

Symbolic execution, the fundamental technique on which SymTest is based, was proposed in 1976[21]. Since then, it has been widely studied and used for various verification and validation activities, particularly testing[21]. Symbolic execution tends to be a computationally expensive process which puts barriers to its use in industrial strength software systems. Much research has been directed to deal with this problem.

The problem of computing the smallest set of paths covering all edges of a DAG was reported in [9]. The problem is proved to be tractable and efficient algorithms have been reported[19].

Concolic testing was first reported in [27, 15]. It mixes random testing[10] and symbolic execution to compute test cases providing coverage over a program under test.

CBMC[7] employs bounded model checking technique to generate test sequences for ANSI-C programs. AutoMotGen[13] uses a similar approach to generate test cases for Matlab SL/SF models. Advantages are complete automation, coverage guarantees and unreachability detection capability. Limitations include scalability issues arising out of state space explosion, an inherent problem with model checking approach.

Hybrid concolic testing [22] reports a related technique which shows benefits over concolic testing particularly when the program under test is a reactive system. REDIRECT[26] employs an interleaved application of random-testing and constraint solving to generate test inputs providing coverage as specified. Random testing provides deep coverage through 'easy' targets; constraint solving provides wide coverage through 'hard' target. This method provides the advantages of scalability. The disadvantages of REDIRECT approach involve lack of guarantees on the optimality of the test data generated due to employment of random techniques.

There are available commercial tools e.g. Reactis[20] and EmbeddedTester[1] with coverage based test generation for Simulink[23]. There has been earlier research for coverage based test data generation for Simulink models. AutoMOTGen[13] takes the approach of translating the model under test into an equivalent transition system in SAL [24]. Thereby, it uses SAL-ATG to generate the test inputs. SAL-ATG[17] is SAL's automatic test generator which in turn uses bounded model checking [6] for test data generation. REDIRECT[26] follows a technique similar to hybrid concolic testing adapting it to Simulink Stateflow models and applying a number of novel heuristics to identify hard targets and probable solutions to them.

SymTest is a technique that uses symbolic execution to generate test inputs for a program given in the form of its control flow graph. A distinctive point of our work is stress is on test sequence optimisation, i.e. minimising the length of the generated test sequence. We have always found a unanimous agreement among our engineering colleagues that test data optimisation is an important problem from their viewpoint. However, to the best of our knowledge, the research in test

generation with explicit emphasis on test sequence (data) optimisation is not in plenty. An example is [2] which employs a heuristic technique based on bacteriological growth in optimisation of test cases. Regression test selection techniques [16] aim to reduce test execution time by selecting only those test cases which are likely to uncover faults induced by a modification of the software under test. In that sense, regression test selection may be viewed as a test optimisation step. A work that comes closest to our approach is given in[25]. This work uses a syntactic approach to find paths followed by using the CBMC bounded model checker. However, there is no attempt to optimise the test path even here. The algorithms presented in therein works on general CFGs, where as SymTest is applicable on CFGs with acyclic core.

Many of our case studies were done on Matlab Simulink models in GM. The Simulink to CFG translation scheme we use as a pre-processing step assumes an informal semantics for Simulink. Semantic issues of timing, types, composition and synchronisation etc. are important problems in their own right. There has been research reported in this direction[18], some in context of translators to other languages[5]. But, we have kept them beyond the scope of the current work.

9. DISCUSSION

SymTest shares its features with many related ideas in test generation. It uses symbolic execution and constraint solving as its solution engine. It explores neighbouring paths using backtracking in a spirit very similar to concolic testing. It uses a depth d_{max} to bound its search. This is analogous to the loop unrolling depth of bounded model checking. Also, it resorts to an exhaustive search of paths within a bounded depth, similar to bounded model checking.

9.1 Strengths

Shorter Test Sequences. SymTest claims its distinction on its attempt to generate shorter *test sequences*. These test sequences are different from the test inputs generated by concolic testing tools which are agnostic to the presence of an outer main loop characteristic of embedded applications. However, SymTest generates test inputs for multiple iterations of the outer loop, this being equivalent to multiple simulation steps in embedded systems. SymTest succeeds to generate optimal test sequences for several of the embedded systems that we tested using SymTest in our case studies.

Acyclic Core. The acyclic core feature may appear to be a restriction on the class of embedded systems that can be treated with SymTest approach. In reality, we have found it possible to mechanically translate a very wide variety of embedded systems modelled with Matlab Simulink (and similar languages like National Instruments LabView) into semantically equivalent acyclic core control flow graph (ACCFG). Models with triggered subsystems, feedback loops, heterogeneous sampling frequency, Stateflow charts, embedded analogue blocks etc. presented no fundamental hurdles to being translated into semantically equivalent ACCFGs. Further, we emphasise that there are several straightforward ways by which SymTest can be enhanced to work on non-acyclic core structures. For example, inner loops can be dealt with some heuristics like iterate once, unroll n times etc. We have not included them here for simplicity of presentation. Moreover, our current implementation does not have them as yet.

Target Set. Finally, the algorithm covers the set of target edges (and this could be easily extended to have edge tuples as target elements), and not all edges (unless specified). This makes SymTest approach particularly useful to regression testing scenario, where the tester does not wish to cover all edges or paths, but only selected ones. In such cases, SymTest clearly scores over all approaches based over concolic testing or random testing which provide no handle whatsoever to the user to specify which elements of the CFG they would like to cover during testing.

9.2 Weaknesses

On Infeasibility. SymTest's one weakness lies in the fact that, once a syntactic path is found infeasible, it must fall back on backtracking and other heuristics to explore nearby paths, whereby the theoretical guarantee of generating optimal length test sequences is lost. It should be noted that the problem of finding optimal paths is undecidable in general. SymTest's extensible architecture which allows interleaving other (external) heuristics with backtracking increases the probability of achieving good results with less computation. A few candidates are presented in CREST[3].

Implementation Related. A couple of shortcomings in our current implementation of SymTest are associated with our symbolic execution. We have indigenously developed SymTest's symbolic execution engine. Currently, its support for various language features is limited (e.g. various data types, function calls, pre-compiled binary etc.). Further, the entire symbolic execution trace is currently maintained in memory. This presents a scalability issue for long runs of symbolic execution. We are exploring the possibility of using already implemented symbolic execution tools like KLEE[4] etc. to ameliorate this limitation. Choice of SMT solvers is critical in determining the power of any symbolic execution tool. Our implementation uses Yices[11, 12] as its SMT solver. Yices' support to various data types and arithmetic operations is limited. Using a more powerful SMT solver like Z3[8], dReal[14] etc. is another step towards strengthening our approach in practice.

10. CONCLUDING REMARKS

In this paper, we have presented SymTest, a general solution to the problem of optimal test sequence generation for embedded software development. The method is fully automated and is designed to optimise the length of the test sequences generated. SymTest aggressively seeks to minimise test sequence length, and provides an extensible architecture to plug in external heuristics. When the set of target edges is a small subset of the total edge set – as in the case of regression testing –, it is important to search out the targets directly. Concolic testing does not provide a good enough start to this search. On the contrary SymTest significantly increases the probability of finding shorter test sequences by starting with the syntactically optimal path, and staying close to it in its subsequent backtracking search. Our experiments show that SymTest indeed succeeds to generate shorter test sequences than concolic testing approach for many embedded applications.

In future work, we intend to extend SymTest prototype tool to support a larger set of language constructs. Thereupon, we intend to do more rigorous experiments to evaluate SymTest's capability in optimising test sequences for industrial strength models and programs. We have designed SymTest so that externally developed heuristics can be plugged in seamlessly (conforming to the interface specification). We are currently working on devising more heuristics and evaluating their impact on SymTest's capability to generate test sequences for larger systems in shorter time.

11. REFERENCES

[1] B. E. S. AG. Embeddedtester, http://www.btc-es.de/.

[2] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Automatic test cases optimization using a bacteriological adaptation model: Application to .net components. In *Proceedings of the 17th IEEE international conference on Automated software engineering*, ASE '02, pages 253–, Washington, DC, USA, 2002. IEEE Computer Society.

[3] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[4] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[5] P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time simulink to lustre. In *In: Third International ACM Conference on Embedded Software, Lecture Notes in Computer Science*, pages 84–99. Springer, 2003.

[6] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19:7–34, July 2001.

[7] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[8] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):pp. 161–166, 1950.

[10] J. W. Duran and S. C. Ntafos. A Report on Random Testing. In *ICSE'81*, pages 179–183, 1981.

[11] B. Dutertre. Yicesä2.2. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 737–744, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[12] B. Dutertre and L. D. Moura. The yices smt solver. Technical report, 2006.

[13] A. A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, and K. C. Shashidhar. Automotgen: Automatic model oriented test generator for embedded control systems. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 204–208, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] S. Gao, S. Kong, and E. M. Clarke. dreal: An smt solver for nonlinear theories over the reals. In *Proceedings of the 24th International Conference on Automated Deduction*, CADE'13, pages 208–214, Berlin, Heidelberg, 2013. Springer-Verlag.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI 2005*, pages 213–223. ACM Press, June 2005.

[16] T. L. Graves, M. J. Harrold, J. M. Kim, A. Porter, and G. Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10:184–208, 2001.

[17] G. Hamon, G. E. Hamon, L. D. Moura, and J. Rushby. Generating efficient test sets with a model checker. In *In: 2nd International Conference on Software Engineering and Formal Methods*, pages 261–270. IEEE Press, 2004.

[18] G. Hamon and J. Rushby. An operational semantics for stateflow, 2004.

[19] J. E. Hopcroft and R. M. Karp. A n5/2 algorithm for maximum matchings in bipartite. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory (Swat 1971)*, SWAT '71, pages 122–125, Washington, DC, USA, 1971. IEEE Computer Society.

[20] R. S. Inc. Reactis, http://www.reactive-systems.com/.

[21] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[22] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[23] Mathworks. Simulink, http://www.mathworks.in/products/simulink/.

[24] L. D. Moura, S. Owre, H. RueSS, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. Sal 2, 2004.

[25] E. D. Rosa, E. Giunchiglia, M. Narizzano, G. Palma, and A. Puddu. Automatic generation of high quality test sets via cbmc. In M. Aderhold, S. Autexier, and H. Mantel, editors, *VERIFY-2010*, volume 3 of *EPiC Series*, pages 65–78. EasyChair, 2012.

[26] M. Satpathy, A. Yeolekar, and S. Ramesh. Randomized directed testing (redirect) for simulink/stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 217–226, New York, NY, USA, 2008. ACM.

[27] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30:263–272, September 2005.

[28] R. Tarjan. Testing flow graph reducibility. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 96–107, New York, NY, USA, 1973. ACM.