Randomized Directed Testing (REDIRECT) for Simulink/Stateflow Models

Manoranjan Satpathy
GM India Science Lab
manoranjan.satpathy@gm.com

Anand Yeolekar
GM India Science Lab
anand.yeolekar@gm.com

S. Ramesh
GM India Science Lab
ramesh.s@gm.com

ABSTRACT

The Simulink/Stateflow (SL/SF) environment from Mathworks is becoming the *de facto* standard in industry for model based development of embedded control systems. Many commercial tools are available in the market for test case generation from SL/SF designs; however, we have observed that these tools do not achieve satisfactory coverage in cases when designs involve nonlinear blocks and Stateflow blocks occur deep inside the Simulink blocks.

The recent past has seen the emergence of several novel techniques for testing large C, C++ and Java programs; prominent among them are directed automated random testing (DART), hybrid concolic testing and feedback-directed random testing. We believe that some of these techniques could be lifted to testing of SL/SF based designs; REDIRECT (RandomizEd DIRECted Testing), the proposed testing method of this paper, is an attempt towards this direction. Specifically, REDIRECT uses a careful combination of the above techniques, and in addition, the method uses a set of pattern-guided heuristics for tackling nonlinear blocks. A prototype tool has been developed and the tool has been applied to many industrial strength case studies. Our experiments indicate that a careful choice of heuristics and certain combinations of random and directed testing achieve better coverages as compared to the existing commercial tools. [1]

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms

Design, Experiments, Measurement, Verification

[1]The opinions expressed in this paper are those of the authors only and do not reflect the opinion of the Organization

EMSOFT'08, October 19–24, 2008, Atlanta, Georgia, USA.

Keywords

Model Based Testing, Simulink/Stateflow Models, Hybrid Systems, Coverage

1. INTRODUCTION

The Simulink/Stateflow (SL/SF) environment from The Mathworks [11] has become the *lingua franca* in industry for model based development of embedded control systems, especially in the automotive and the aerospace domains. This environment supports hierarchical development of complex controller designs and provides a rich set of high level and customizable computational and control blocks suitable for hybrid control systems. A wide variety of application specific block-sets available with the SL/SF environment enable easy development of control systems in various domains. SL/SF models help in early design exploration, simulation and automatic code generation for different hardware/software platforms.

Our focus in this work is *model based testing* using SL/SF models. In model based testing, the models form the basis for generating test cases which can be used to check whether an implementation conforms to its model. The model elements provide additional notions of coverage over model elements that help in measuring the test adequacy, which is central to any testing activity. Model coverage along with coverage over code gives an added confidence.

Many methods have been proposed for automatic generation of tests from state transition graph models. These methods usually explore the transition graphs either randomly or use systematic graph exploration techniques or a combination of these two. Random generation has been the preferred method when the model is large or infinite as in the case of SL/SF models. While searching the infinite number of paths in the model graph, coverage measures on the model space, like state and transition coverage, are used to prune the number of paths explored. One issue in exploration is that many of the paths may be invalid. There are constraints associated with paths which may not be satisfiable. So constraint solving is an important task during exploration of paths. In the SL/SF context, coverage of the states and the transitions of a Stateflow block could be highly non-trivial because the Stateflow block may be deeply embedded within the Simulink blocks. Inputs may have to pass through a complex network of Simulink blocks to reach the Stateflow block. Furthermore, such Simulink blocks may be non-linear which in turn may give rise to non-linear constraints for which no satisfactory methods exist for solving these constraints.

Many commercial tools have appeared in the market for test case generation from SL/SF designs. Some of the prominent tools are: *Reactis* from Reactis System Inc. [16], *STB* from the TNI Software [21], `BEACON Tester` from Applied Dynamics International [1], `T-VEC tester` from the T-VEC technologies [22] and recently, the *Design Verifier* from `The Mathworks` [12]. Given an SL/SF design model and coverage goals over the model elements, all these tools generate test vectors meeting the coverage goals. These tools use powerful random or constraint solving techniques or a combination of both. Our experience with these tools shows that they do not produce satisfactory coverage in cases when the designs involve non-linear blocks and the Stateflow blocks occur deep inside the Simulink blocks. Coverage of the model elements in such Stateflow blocks requires careful and systematic selection of input values. The primary motivation of this work is to go well beyond the current techniques in constraint solving and directed exploration of the SL/SF model elements.

In the recent past, code based testing has received renewed interest and attention with the emergence of several novel techniques for testing large C, C++ and Java programs. Prominent among such techniques are: *directed automated random testing* (DART)[6], *hybrid concolic testing* [10] and *feedback-directed random testing* [15]. DART mixes concrete execution with symbolic execution and generates constraints for uncovered paths from the constraints of already covered paths. Hybrid concolic testing interleaves random testing and DART to cover model elements both depthwise and breadth-wise. Feedback directed random testing uses feedback from concrete executions to generate new input sequences. These techniques have worked remarkably well for C, C++ and Java programs. We strongly believe that some of these techniques could be lifted to testing of SL/SF designs and this paper is our initial attempt towards this purpose.

REDIRECT (RandomizEd DIRECt Testing), as we call our method, uses a careful combination of the following techniques: (a) Testing by use of random input sequences, (b) Directed traversal from a point already reached due to simulation, (c) Backtracking which means going backward in a traversed path to cover some more model elements, and (d) feedback-based random testing in which feedback from a part of the current run is used in obtaining the future part of the run. An important highlight is that REDIRECT uses a set of pattern-guided heuristics for tackling the non-linear blocks. We have developed a prototype tool based on REDIRECT and the tool has been applied to many industrial strength case studies. Our experiments indicate that a careful choice of heuristics and certain combinations of random and directed testing can achieve better coverages as compared to the existing commercial tools.

In summary, the main contributions of this paper are:

- A flexible mix of random and systematic testing through the four components – random, directed, backtracking and feedback-based testing – of the REDIRECT approach achieves an improved coverage over SL/SF models.
- Our method uses a set of smart heuristics to achieve better coverage of the model elements even when the transition constraints or the Simulink block functions are non-linear.
- A prototype tool based on REDIRECT has been developed and experimented with real designs.

The organization of the paper is as follows. Section 2 discusses the related work. Section 3 describes in detail the test case generation algorithm and the heuristics to handle non-linear constraints. In Section 4, we discuss our prototype implementation and our experimental results. Section 5 presents an analysis of our approach, and finally, Section 6 concludes the paper.

2. RELATED WORK

The ETS 300 406 [4], the ETSI standard for *Methods for Testing and Specification*, defines: *conformance testing* is concerned with the assessment of the extent to which an implementation or a system conforms to its specification. Since exhaustive testing is impractical on technical and economic grounds, conformance testing cannot guarantee absolute conformance to a specification. This approach to testing is usually associated with a *testing criterion* which is a set of requirements on test data which reflects a notion of adequacy on the testing of a system [18]. A test adequacy criterion determines whether sufficient testing has already been done, and in addition, it provides measurements to obtain the degree of adequacy obtained after testing stops [23]. A *test oracle* is a mechanism to determine correctness of test executions. A *test driver* (or a test harness) is a tool which activates a system, provides test inputs and reports test results.

The *Reactis tester* from *Reactive Systems* [3] generates test cases automatically from SL/SF models. The technique is primarily random simulation; inputs are selected by Monte Carlo methods. It also uses the technique of guided simulation [3, 20] under which after a simulation, output values are examined and then subsequent input values are chosen so as to guide the simulation to uncovered parts. Reactis tester can generate test cases based on many coverage criteria over the Simulink and Stateflow blocks including the MC/DC (Modified Condition/Decision) coverage [14]. For random and guided simulation, Reactis takes three parameters: (i) number of tests meaning the number of random simulations, (ii) number of steps for each random simulation, and (iii) number of targeted steps for guided simulation. A test case can be viewed as a matrix in which each row corresponds to the sequence of values of an inport or an outport and each column corresponds to a single simulation step. After running a test case, the model is brought back to its initial state so that another test case can be run [20]. STB (Safety Test Builder) from TNI Software [21] is another successful testing tool for Simulink/Stateflow models. Test inputs are primarily random with some heuristics. STB uses almost the same coverage objectives as that of Reactis including the MC/DC over Simulink and Stateflow. BEACON tester from Applied Dynamics International [1] and T-VEC tester T-VEC technologies [22] are two other commercial testing tools for SL/SF models. From the very little documentation that is available, it appears that both have probably the same capability as those of Reactis and STB.

Gadkari et al. [5] have discussed test case generation from SL/SF models by using model checking. Models restricted to a subset of SL/SF are translated to formal models in the SAL [19] formal language. Keeping a coverage criterion over

the SL/SF model in mind, the SAL model is so instrumented with *trap variables* [7] that reachability of a trap variable implies reachability of a model element; traces leading to trap variables then become the test cases. The ATG tool does not handle non-linear constraints with the exception of multiplication; and further, the authors point out that it would face scalability issues in an industrial setting. When we view test case generation possibilities, pure random testing is at one extreme and the model checking approach is at the other extreme. The REDIRECT method is somewhere in between as it combines random and systematic exploration.

Recently, The Mathworks has introduced Simulink Design Verifier [12] whose functions are test case generation, proving of model properties and generation of counterexamples. In this tool, model coverage objectives include decision, condition and MC/DC. Custom test objectives can be directly defined over SL/SF blocks by using design verification blocks. This tool can also show unreachability of certain model elements. Not much details about the performance of Design Verifier are available in the literature as it is a recent tool.

Godefroid et al. in [6] discuss the DART (Directed Automated Random Testing) approach for C programs. The power of DART comes from a combination of random inputs, constraint solving and directed coverage. Given a C program, first a random input vector is supplied to execute the program and the symbolic constraint associated with the covered path is extracted. This symbolic constraint is altered to obtain the path constraint of a path adjacent to the earlier path. If this path constraint can be solved, then we have the input vector for the new path. Provided constraints are solvable, by systematically altering the constraints one can cover all the paths of finite length. Let X and Z be the inputs to a program and $f(X) = Z$ be a control condition in the program. Let f be a library function whose internal details are not known. Execute this code fragment by giving random inputs to X and Z, and let control take the right path. But now the concrete value of f(X) is known with the current input. The same value of $f(X)$ will be the input of Z in the next iteration – value of X will remain unchanged – and thus we can traverse the left path. The constraint solver faces problems when the constraints are non-linear. When non-linearity is sparse then heuristics are available to find path solutions with a certain probability. However, this probability decreases sharply when a path constraint has too many non-linear constraints.

Approaches like DART are unlikely to reach deep states within the state space of the given program. This is because maintaining and solving symbolic constraints along execution paths become more and more expensive as the length of execution grows. The *hybrid concolic testing* [10] approach addresses this issue by interleaving DART and random testing. Thus random testing takes care of the depth in the state space whereas DART takes care of breadth. The authors of this technique point out that this approach is more suitable for reactive programs which periodically get inputs from environment.

Pacheco et al. [15] have discussed random testing with feedback for unit testing of object oriented programs. This approach incrementally builds test sequences from a set of previously generated test sequences which are empty to begin with. The process first selects a method $m(T_1, \ldots T_k)$ at random. Next, depending on parameter types $T_1, \ldots T_k$, a matching prefix sequence is obtained from the set of sequences available which becomes the preamble of the above method. This new sequence is executed to check its legality in the context of the given program. The authors of this method have observed that the results obtained by this approach are comparable to those of systematic testing.

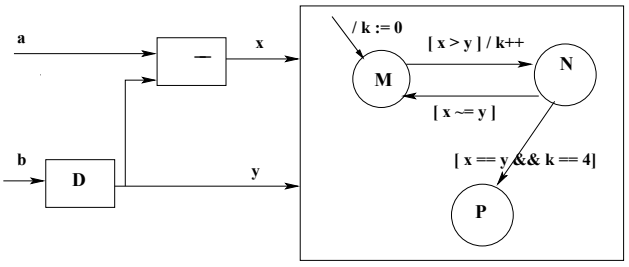

Figure 1: A simple SL/SF model

Alur et al.[2] have presented an analysis technique for linear hybrid systems which combines numerical simulation with symbolic analysis. Given a simulation trace of an SL/SF model, using polyhedra based backward symbolic analysis, a region of initial state space inducing equivalent behaviors in the model is identified. Further simulation is carried out on the model, starting from initial states outside of this region and the above step is repeated a finite number of times to have adequate coverage over the initial states. Another method for initial state coverage is proposed by Agung Julius et al. [9] using bisimulation metrics. Our method is distinct from these approaches as we focus on structural coverage over the models. Another important difference is that our models could have varying inputs at different simulation steps.

3. THE REDIRECT APPROACH

We will introduce the `REDIRECT` approach with a motivating example. Refer to the SL/SF diagram of Figure 1 in which the Stateflow part consists of three states and the Simulink part consists of a delay block and a block for an arithmetic operation. Here, state M is the initial state of the Stateflow block. We will explain the simulation behavior of this SL/SF model.

At each time step (either user-given or automatically derived) the model executes all of its blocks (including the Stateflow block) according to a pre-defined order. In this example, the *delay* block D executes first, followed by the subtraction block, and in the end the Stateflow block (or the chart) is executed. Assuming that the current *active state* is M, the guard of transition MN is checked to see if it holds. If so, the action $k{+}{+}$ associated with the transition is executed and state N becomes *active*. Since there are no more active states – this can occur in case of concurrent Stateflow charts – the Stateflow block completes its current processing step. As all the blocks of the SL/SF model have been processed, the model is now ready to accept its next input vector and proceed to the next simulation step. This cycle repeats till the simulation duration ends.

Let us consider the reachability of the state marked **P**. The equations for **x** and **y** in the above figure are: $x_n = a_n - y_n$ and $y_n = b_{n-1}$ for $n \geq 1$, with n being the n-th sampling step. y_1 gets the initial value of the delay block which is user defined. Assume the Stateflow diagram is unfolded from the initial state up to a certain depth (refer to Figure 2).

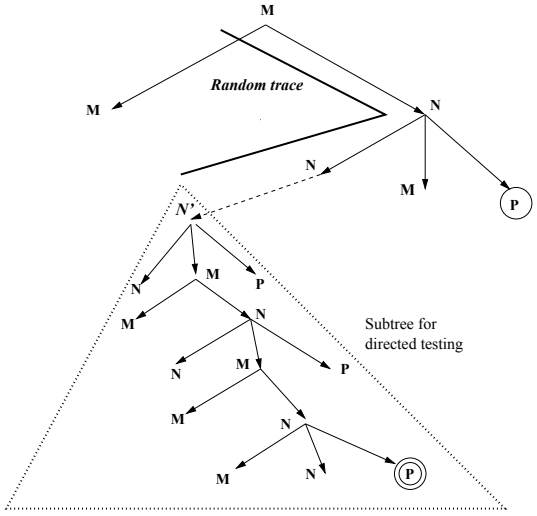

Figure 2: Coverage by random and directed testing

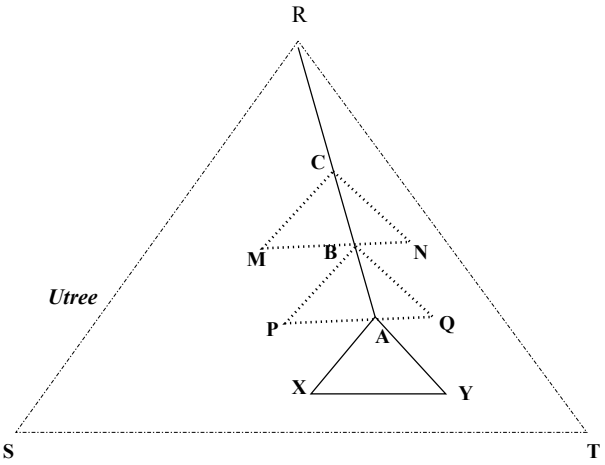

Figure 3: The unfolded tree and the coverage approaches

Each transition can be associated with a constraint, and the conjunction of all constraints in a path is termed as the *path constraint*. For instance, the path constraint of the transitions leading to **P** at level 2 (shown within a circle) is:

$$x_2 = y_2 \wedge k_2 = 4 \wedge x_2 = a_2 - y_2 \wedge y_2 = b_1 \wedge k_2 = k_1 + 1 \wedge x_1 > y_1 \ldots$$

This constraint was fed to the constraint solver *yices* [19], and the constraint was found to be unsatisfiable implying that this **P** is not reachable. On the other hand, the path constraint associated with the **P** at level 8 and shown within double circles in the figure is:

$$x_8 = y_8 \wedge k_8 = 4 \wedge x_8 = a_8 - y_8 \wedge y_8 = b_7 \wedge k_8 = k_7 + 1 \ldots$$

This constraint was fed to *yices*, and we got the solution: $[(1, 0.5), (0, -0.5), (0, 0.5), (0, -1), (-1, 0.5), (0, -1), (-1, 0), (0, 0)]$. Thus **P** was reached by pure directed testing.

The REDIRECT tool reached **P** by using a combination of random and directed testing. The random input sequence $[(3, 2), (4, 0), (2, -1), (-1, 3), (6, 4)]$ was applied and it took control to state N; this path has been shown by the bold line from root to node N' in Figure 2. With respect to N', directed testing was applied over the subtree of depth 5, shown as the triangle with dotted lines. It was found that, all **P**s occurring at depth less than 5 in the subtree were unreachable. However, the path constraint for reaching **P**, shown within double circles, was solved by *yices*, and we got the solution: $[(9, -0.5), (0, 0.5), (0, -0.5), (0, 0), (0, 0)]$.

With the aim of reaching the state marked **P**, we ran Reactis multiple times with very high parameter values; one such run had parameters: no. of tests = 1000; no. of steps/test = 1000 and no. of target steps = 100000. However, none of the runs could reach **P**. Furthermore, using Reactis, we could achieve 50% MC/DC coverage of the Stateflow block, whereas, with REDIRECT, we could achieve 100% MC/DC coverage.

3.1 Test case generation algorithm

In this paper, we limit coverage objectives to covering the states and transitions in a Stateflow. We wish to highlight the power of our method by considering these simpler objectives; however, more complex objectives (e.g. transition-pair coverage) can be incorporated to the method and to the tool without much of difficulty.

The first three components of REDIRECT approach – Random Testing, Directed Testing and Backtracking – can be best seen from the diagram of Figure 3. The method assumes that the states of the Stateflow model are unfolded up to a pre-defined depth to form a tree with its root representing the initial state. Henceforth, this unfolded tree will be known as `Utree`. We assume that hierarchical and parallel Stateflow components are flattened, and thereafter, the `Utree` is constructed. Note that the `Utree` is never constructed in full; relevant subtrees within it are constructed dynamically on a need basis. In the figure, the region RST represents the `Utree`. A random input sequence over the given SL/SF model will give rise to a trace which will be a path in `Utree` (assume that the trace does not cross the boundary RST.) In the figure, R and A are respectively the start and the end points of such a trace. The trace RA may have covered some of the model elements within the coverage objectives, and therefore the latter needs to be modified after this phase. Generation of random traces may be performed a number of times. These steps constitute the Random testing component of REDIRECT.

A subtree of a certain depth with the end-point of the simulation trace as the root – A in the figure – is marked within the `Utree`. All the targets (transition edges) within this subtree are collected and are marked for coverage. In the figure, AXY is such a subtree. A Stateflow edge may have many occurrences in the subtree but all of them may not be feasible. Path constraint of a target edge – in relation to the root of the subtree – is derived and it is fed to the constraint solver to get a solution. If the path constraint is linear then a constraint solver can possibly find a solution. However, if the constraint is non-linear, its pattern is identified and the heuristics associated with this pattern in a library are used to generate input sequences with the aim of covering the said target. This phase constitutes the `Directed Testing` component of `REDIRECT`. The choice of the depth of this subtree is a crucial factor; either it can be supplied as an input parameter or it can be decided dynamically.

In the `Backtracking phase`, our method backtracks along the simulation trace to continue directed testing. In the

figure, B is the state after backtracking, and directed testing continues from there by constructing the subtree BPQ. Once this is over, there can be further backtracking, and in the figure, C is the next backtracking point. A heuristic can decide on how many steps to backtrack and how many times to backtrack.

```
Inputs:
 M : Given SL/SF model.
 Utree: Unfolded Stateflow states as a tree
 MaxDepth: max depth of Utree.
 CO: Coverage objective.
 bsteps: max steps to go back while backtracking.
 tdepth: From a subtree with end(trace) as root, targets
 are selected. tdepth  is the max depth of this subtree.
Step1:im←instrument_model(M);
      ran_att← 0; //random attempts
      itrace ← null; // incremental trace
Step2:iseq←generate_ran_inp_seq(im);
      ran_att←ran_att +1;
Step3:trace←simulate(im,root(Utree),iseq);
      out_test_sequence(trace);
Step4: // # is the concatenation operator
      CO←update_coverage(CO,trace#itrace);
      itrace← null;
      if (coverage_done(CO,ran_att)=true)
        STOP;
Step5:tar←new_target(Utree,trace,CO,tdepth);
      if tar=null {
        trace ← backtrack(trace,bsteps);
        if trace=null goto Step 2
        else goto Step 5;
      }
Step6:constr←get_constraint(Utree,trace,tar);
      if is_non-linear(constr,tar,im)=true {
        itrace←nl_solver(tar,constr,trace,im);
        if (itrace=null) goto Step 5;
        else {
          out_test_sequence(trace#itrace);
          goto Step 4;}
      }
Step7:sol←solve_constr(constr,trace,Utree);
      if sol ≠ null {
        out_test_sequence(trace#sol);
        itrace ← sol;
        goto Step 4;
      } else goto Step 5
```

Figure 4: Test case generation Algorithm

The test case generation algorithm under `REDIRECT` which uses the above three techniques is given in Figure 4. `MaxDepth` is the depth of the `Utree` and `CO` is the coverage objective. `tdepth` is depth of the subtree which is constructed for directed testing, and `bsteps` is the number of edges to go back along the simulation trace for backtracking. The entities like `tdepth, bsteps` etc. are supplied as input parameters; however, a tool, for efficiency reasons, may compute these parameters dynamically.

In the algorithm, routine `instrument_model()` in **Step 1** instruments the model; instrumentation is required to capture all information associated with a simulation run, this will be discussed later. In Steps 2 and 3, the model is simulated by generating a random input sequence. After every simulation trace, the coverage objective needs to be modified and this is performed by the routine `update_coverage()` in **Step 4**. The routine `new_target()` in **Step 5** produces a new target within the subtree meant for directed testing. Routine `backtrack()` produces a truncated trace to be used for further directed testing and it returns `null` if no further backtracking is possible. When the constraint is non-linear, the routine `nl_solver()` in **Step 6** tries to cover the target; more about non-linearity will be discussed later. The routine `solve_constr()` in **Step 7** solves a linear path constraint and produces a solution if the constraint is solvable; otherwise, it returns `null`. Directed testing always occurs with respect to a reference point (root of a subtree). During this phase, the simulation trace augmented with the non-null solution produced by the constraint solver, is in effect a trace to reach the current target from the root of the `Utree`. Therefore a test case can be generated from the simulation trace and the non-null solution obtained; this task is performed by the routine `out_test_sequence()` of **Step 7**. Similarly, if the **nl_solver()** covers a target, a test sequence is also generated (refer to **Step 6**). Depending on the coverage achieved, and the number of random sequences generated, a decision is taken whether to continue or stop further testing; this task is performed by the routine `coverage_done()` (in **Step 4**).

```
nl_solver(tar1,constr1,trace1,im1) {
 /* constr1: non-linear constraint of transition tar1*/
 /* trace1: simulation trace: root to source of tar1*/
 /* im1: instrumented SL/SF model */
 /* Max_Iter: max trials to cover a non-linear target*/
no_iterations ← 0;
trace_list ← [trace1]; /* singleton list */
while (no_of_iterations ≤ Max_Iter) {
 iseq ← heu_inp_seq(trace_list,constr1,im1);
 htrace  ← simulate(im1,end(trace1),iseq);
 if (already_covered(tar1,htrace) = true)
   return (htrace);
 else {
   trace_list←htrace::trace_list/*list constructor*/
   no_iterations ← no_iterations +1;
 }
}
return (null);
}
```

Figure 5: Algorithm for handling non-linear constraints

Let R, D and B respectively stand for Random testing, Directed testing and backtracking; the algorithm above has considered the ordering in which each run is of the form $R(DB)^+$, where + stands for one or more times. This means, for each run, we perform one R followed by $(DB)^+$, and there can be several such runs. However, our method can easily be extended to other orderings like $(R|D|B)^+$ and $R(DB)^+(R_m(DB)^+)^+$. The former means, for each run, select any of the testing approaches non-deterministically and perform it, and then make another choice to continue the cycle a number of times. The latter, shown in Figure 6, means perform R followed by $(DB)^+$ but store in memory a reachable point in the very first D. Then from this point in

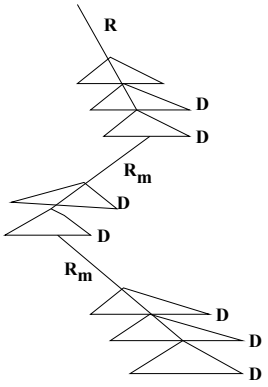

Figure 6: A possible ordering

memory, perform a random simulation (or simulation by an input sequence generated by a heuristic) which we denote by R_m, and thereafter perform a $(DB)^+$ again storing a reachable point in the first D of the current $(DB)^+$ in memory. Next, continue with the $R_m(DB)^+$.

3.2 Handling non-linear constraints

During directed traversal (Step 6 in the algorithm), `nl-solver()` of Figure 5 tries to cover a target with non-linear constraints. The algorithm refers to a library called the *pattern guided heuristic library* (PGHL) which contains a set of patterns and each pattern is associated with a sequence of ordered heuristics which will be used to generate the next portion of the input sequence (with respect to the current state) with the aim of covering the target. In addition, this phase uses feedback-based testing techniques. Feedback can be of two kinds. Backward feedback tries to get information from previous runs and uses it to cover a target, whereas in forward feedback, we go for one or more dry runs to obtain information which could be used to cover the desired target; more about dry runs will be explained later.

The routine `heu_inp_seq()` either consults the PGHL or attempts a dry run to obtain the next portion of the input sequence. Each time an input sequence is obtained, incremental simulation occurs with the end of the current trace as the starting point. This goes on till the number of iterations reaches the max limit (`Max_Iter` in the algorithm) or till the target is reached, whichever is earlier. When the `nl_solver()` fails to cover a non-linear constraint, it returns `null`.

3.3 Pattern Guided Heuristic Library (PGHL)

While attempting to cover a target, REDIRECT attempts to match non-linear constraints syntactically with some predefined patterns. Patterns commonly encountered in SL/SF designs are stored in a table and heuristics associated with each pattern help in generating input vectors. A single pattern may match multiple heuristics; several heuristics may be combined to generate the next input sequence. Some of such heuristics have been shown in Table 1. Since we are dealing with domains like automotive and aerospace, most of the inputs are associated with physical phenomena, and therefore can be assumed to be continuous and monotonic for some duration. The key to non-linear constraint satisfaction using these heuristics is to obtain monotonic behavior of the constraints and quantities involved.

The table entries need some explanation. The first column contains the nature of the pattern. The second column presents a typical example of the pattern; subscripts nl and l respectively denote a non-linear function and a linear function respectively. No subscript means the function could be anything. Usually, we assume a non-linear function to be uninterpreted and only take its black box view. The third column outlines the heuristic to handle such a situation. The first three columns in essence represent the PGHL. The fourth column states the form of actual constraint that we have encountered in our case studies. The last column tells how many times the same heuristic was applied to reach a solution. While generating an input sequence, we obtain a sequence of a certain length. Next, suppose we discover that the input sequence converges towards the solution but does not meet it; in such a case, we need to generate an input sequence of greater length, and we may have to do it repeatedly to reach the constraint. The feedback from the earlier runs – we call them dry runs – are used to obtain the next input sequence. The last column tells about this number of iterations in the case studies.

4. TOOL IMPLEMENTATION AND CASE STUDIES

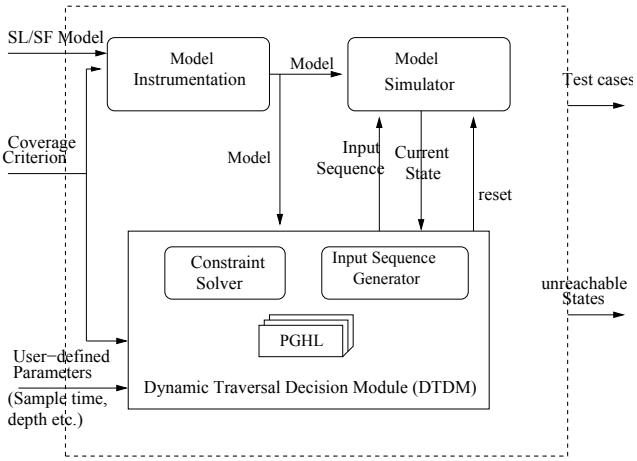

Figure 7: The Tool Architecture

4.1 The tool architecture

The architecture of the `REDIRECT` tool can be seen from Figure 7. The tool takes a SL/SF model, a coverage criterion and some user-defined parameters like sampling time, the subtree depth for directed testing etc. The model is first instrumented and fed to the model simulator and also to the Dynamic Traversal Decision Module (DTDM). Instrumentation mainly consists of translating the Simulink functions or Stateflow guards to elements in a constraint language, and to capture all information associated with simulation runs.

The DTDM is the major component of this tool. It has three sub-components: a constraint solver, an input sequence generator and a pattern-guided heuristic Library (PGHL). The DTDM feeds an input sequence (either random or heuristic driven) to the simulator which simulates the instrumented model accordingly. After the end of the simulation, the simulation trace is supplied back to the DTDM. Based on the

Pattern	Example	Approach	Where encountered	no. of iterations
Simple non-linear constraints	$(x * y) \circ k$ $(x/y) \circ k$ $\circ$ is a relational operator x, y are inputs	Hold x constant. Get i/p sequence of size s by varying y in one direction. Change s or direction if necessary.	in some contrived examples	
Common i/p on both side of constraint	$f_{nl}(x, y) \circ g(y)$ $f_{nl}(x, y, x) \circ g(x, z)$	Hold common inputs constant. vary other i/p in both directions.	$f_1(brake, throttle)$ $< g_1(throttle)$	15
rate of growth on both sides	$f_{nl}(.) \circ g(.)$	Estimate slopes from prior simulations of f & g and estimate time steps to satisfy constraint	$f_2(brake, throttle)$ $> g_2(brake)$	8
i/p generation with concrete values	$f_{nl}(x, y) \circ g_l(z)$ f non-linear; g linear	Take concrete value of of $f_{nl}(x, y)$ from simulation to get $C \circ g_l(z)$; solve for z. Assumption: $f(x, y)$ does not change much for some duration	$after(TWAIT, tick)$ $\&\&speed \geq up\_th$ speed is non-linear, take its concrete value; up_th is linear (lookup table entry)	2
i/p generation from history	$f_{nl}(x, y) \geq g(x, u, v)$ (similarly for $f_{nl}(..) \leq g(..)$)	From previous simulations, identify i/p sequence with f increasing & g decreasing; append same i/p seq to current run.	$f_3(brake, throttle)$ $> g_3(brake)$	4
Sampling step abstraction (e.g. integrator blocks); constraint length is too long	$x_{n+1} = x_n + \delta * I_n$ δ: discrete step chosen by user.	sample system behavior coarsely; replace δ by $k * \delta$; obtain missing inputs by interpolation	*after(700,wakeup)*	2

Table 1: Pattern-heuristic mapping in PGHL

feedback received, the DTDM may output a test vector, or, it may decide on the next input sequence, and if necessary, resets the model for a fresh traversal. The heuristic library is used when the constraint is non-linear and the solver is not able to solve it.

4.2 Tool implementation

We will first discuss the implementation of instrumentation steps. For the Simulink part of the model, we attach observers to the outputs (or inputs) of state-based blocks, such as integrators and delays. This is performed by attaching *To-Workspace blocks* to the Simulink model and enabling data-recording for each discrete simulation step. This is implemented by a Matlab script which uses Matlab `api` calls for attaching new blocks and setting block properties through call-back functions. For Stateflow blocks, additional Matlab code is appended to every state *entry, during* and *exit actions*, and to *condition/transition actions* of every transition; this is again achieved by parsing the Stateflow charts using Matlab `api` function calls. As the simulation progresses, a data structure in the workspace, usually a cell array, records inputs given to the model, data values of the state-based variables and activity within the Stateflow chart. This can be analyzed post-simulation by the test generation algorithm.

Next, we parse the model – by a parser similar to the one used in [5] – for generating constraints in the language of *yices* [19] which is our constraint solver. Blocks in Simulink are translated to their semantic-equivalent constraints. For instance, in Figure 1, the delay block is translated into $y_n = b_{n-1}$ and the transition guard $[x > y]/k++$ is translated into $x_n > y_n \wedge k_{n+1} = k_n + 1 \wedge x_n = a_n - y_n \wedge y_n = b_{n-1}$ with $n > 0$.We thus have timed copies of the variables corresponding to the discrete time steps. Where a semantic-equivalent linear translation is unavailable – such as multiplication/division, transfer function, library calls, masks, etc – we substitute those with uninterpreted function calls with inputs as parameters (black-box view). During actual constraint-solving process, the calls, if required, are replaced with their concrete values obtained from simulation. Since, we usually require a multi-step solution, a Matlab script controls the generation of these constraints for a given number of time-steps.

For the simulation component, we use the Matlab simulation engine [11], invoked using the **sim** command. It accepts an instrumented model and an input sequence, and simulates the model to produce a trace.

The DTDM stores previous simulation runs to learn about the black-box (i.e. non-linear) function behavior. The PGHL stores a set of ordered heuristics against each pattern; each heuristic enables the generation of an input sequence. Consider a constraint pattern of the form: $f(x, y) > g(y)$ with f non-linear and g linear. The first heuristic for this pattern suggests to keep y – the common variable – constant and to vary x. The first learning input sequence is generated as follows: given current x (x_c), maximum value of x (x_{max}), current y (y_c), incremental step for x (δx) and input sequence length ($hlength$), the input values are:
for $j = 1 \ldots hlength$, $y[j] \leftarrow y_c$
for $j = 1 \ldots hlength$, $x[j] \leftarrow min(x_c + (j-1) * \delta x, x_{max})$

This is the input sequence from the current state. If incremental simulation of this input sequence converges towards the solution, then either we hit the target or if necessary, we next generate a longer input sequence. In case of divergence, input sequence is generated in which x is gradually decreased. If this heuristic fails, then DTDM applies the next heuristic, and this continues till the target is hit or the process is aborted.

4.3 Case Studies

In addition to performing studies over a number of smaller models, we have evaluated our REDIRECT tool over five SL/SF models: four in the automotive domain and one in the aerospace domain. Table 2 tabulates the coverage results both from Reactis and REDIRECT. Reactis was run with very high parameter values and the table entries show the best results which were compared with the best results of REDIRECT. The first column contains the names of the applications and the sizes of their SL/SF models. Note from the table that REDIRECT always achieves same or higher state/transition coverage in relation to the Reactis results. Though Reactis was given the opportunity with high parameter values (refer to column 1 under the heading Reactis), it missed some targets deep down in the tree. REDIRECT achieves better coverage because complicated targets, in particular targets with non-linear constraints, benefit a lot from the use of heuristics and *feedback-based testing*. Coverage metrics in the *ATC* and the *Power Window Controller* applications demonstrate that. The last row shows the results of the *Missile Guidance controller* (MGC) model which contains some non-trivial Simulink blocks like continuous transfer functions and continuous integrators. Reactis does not support such Simulink blocks as of now, and therefore, could not handle this model as such. However, REDIRECT did not face any problem while handling this model since such blocks were treated as uninterpreted functions.

In the following, we will discuss a case study in detail and present short descriptions of the rest. All such case studies are available as demo examples in Matlab 7.0.

4.3.1 Automatic Transmission Controller (ATC)

The ATC system is a controller for a 4-gear automatic transmission based on two primary inputs: brake and throttle. As shown in Figure 8, this system has five subsystems: Vehicle, Engine, Gear-shift logic, Threshold calculation and Transmission. The gear-shift logic is modeled in Stateflow (Figure 9) and the rest are modeled in Simulink. The Stateflow model has nine states altogether and is hierarchical. It has also parallel-AND and exclusive-OR modes. The Simulink blocks contain integrators, gains, look-up tables and feedback loops. The constraints of the transitions in the Stateflow model have non-linear components.

The controller implements logic for shifting gears based on upshift and downshift thresholds, specified as lookup tables. A gear is shifted upwards (downwards) if current speed continues to violate threshold for a given amount of time as indicated by the two transitions from **upshifting** (**downshifting**) to **steady_state**. The engine subsystem models the engine behavior which takes the throttle value as input and outputs the `rpm`. The transmission subsystem takes gear value and the `rpm` as inputs and outputs a torque estimate to the vehicle subsystem. The vehicle subsystem calculates the speed based on the brake applied and the torque.

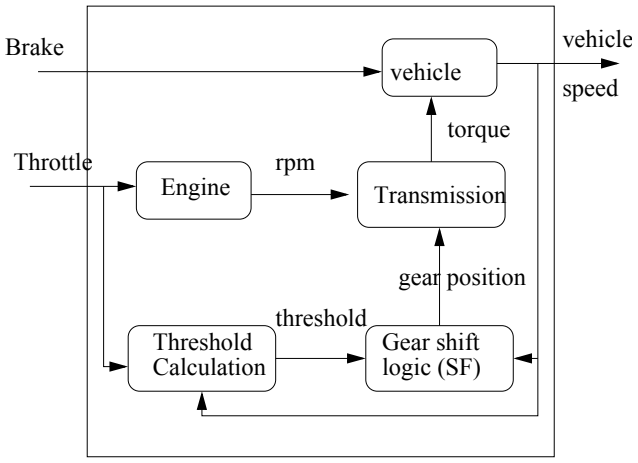

Figure 8: The Automatic Transmission Controller

Reactis was not able to fully cover the Stateflow as shown in Table 2. REDIRECT applied random testing, directed testing and four heuristics from the PGHL and achieved full coverage. We will illustrate the application of one such heuristic here. For the transition guard $[speed > up\_th]$, *speed* is a non-linear function of *brake* and *throttle*, and *up_th* is linear in *throttle*. Thus the heuristic of second entry in Table 1 is applicable here. *throttle* being the common variable was fixed and *brake* was varied. It was observed that *speed* increased monotonically with increasing value of *brake*. The generated input sequence covered the transition. For the transition $[after(TWAIT, tick)\&\&speed > up\_th]$, current concrete value of *speed* was used to find a solution for *throttle*. Thus heuristic of the fourth entry in Table 1 was used, and the input sequence generated by this heuristic covered the above transition.

4.3.2 Other case studies

The *Power Window controller* (PWC) (this is a variant of the PWC in Matlab demo) is for controlling the glass movements in the vehicle windows. The input to the system is *driver_command*, which may be 1 for moving the window up, 0 for downward movement and 2 for neutral. The controller outputs appropriate values to the plant, modeled using a discrete integrator. An internal variable *hit* tracks when the window hits the top or the bottom threshold. A special state *CycleDone* is entered on completing a movement from the top to bottom position or vice-versa. Reactis could not achieve full coverage; REDIRECT covered it by using directed testing, followed by backtracking and then another round of directed testing.

The *Cruise Controller* (CC) allows a car to cruise automatically at a set speed. The system senses the brake, accelerator pedal, and other cruise activation controls from the environment. The controller can be in active mode only if no brake, accelerator or cancel command is issued and the current speed is above a threshold. Reactis could achieve full coverage for this example. REDIRECT achieved full coverage by using a combination of random and directed testing.

The *Adaptive Cruise Controller* (ACC) is responsible for determining the host vehicle's speed on a continual basis. It takes driver's input and environment information which

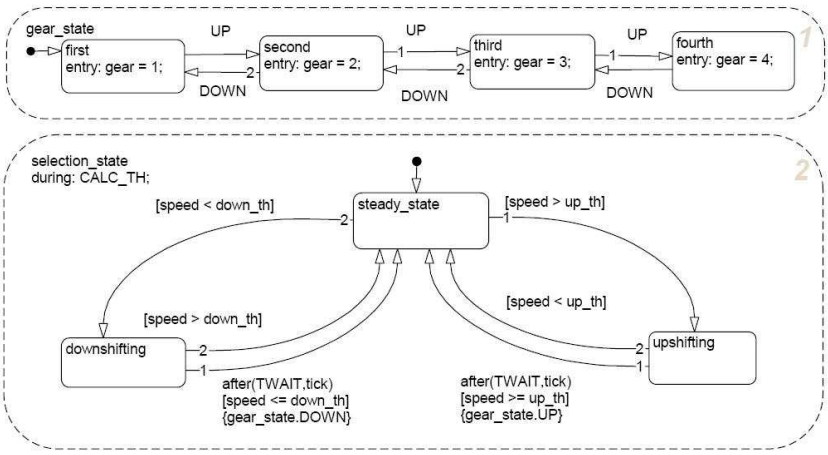

Figure 9: SF chart of ATC

includes current speed, leader's presence/absence and the gap between the host and the leading vehicles. A combination of directed testing, random testing and backtracking was used by REDIRECT to cover all but two states. All heuristics failed to cover these two states and the associated transitions.

The *missile Guidance Controller* (MGC) takes the target and missile positions and direction as input and decides by what amount the trajectory should change. The Stateflow block (two parallel Stateflow charts) defines all mode transitions during normal and unusual situations. This model has a large number of non-linear blocks which mainly model the non-linear rigid body dynamics. Let us consider a particular constraint which contains the predicate $after(700, wakeup)$; this is to be satisfied to cover a transition. The number of inputs required is at least 700, which may involve a lot of computation for the heuristic input generator (or the constraint solver, in case of linear constraints). So we have applied the sampling time abstraction technique (last row in Table 1) two times to solve this constraint. We consider inputs at wider sampling intervals and in the process, generate a much shorter input sequence. The missing intermediate input values are next obtained automatically by interpolation.

5. DISCUSSION

- From our experiments, we can infer that a careful combination of random testing, directed testing, backtracking and feedback based testing is a powerful technique for testing of Simulink/Stateflow models. In other words, we have integrated interesting aspects of DART, hybrid concolic testing and feedback based testing into REDIRECT. Since directed testing is localized – we generate constraints for transitions in a subtree of relatively smaller depth – scalability is never an issue. Some ideas about feedback based testing from previous runs has been mentioned in [15]; however, REDIRECT considers both backward and forward feedbacks for covering targets.

- If a state has many outgoing transitions, they can be assigned probabilities of occurrence; for example, a target with an equality is less probable to occur in a random simulation. When a transition probability is less, then directed testing is the choice, and when probability is more then random testing is the choice. Thus probabilities can also be associated with a path. Keeping these probabilities in mind, for a model, a plan can be made about the way to cover the coverage objectives. Thus the ordering $(R|D|B)^+$ can further be refined to exploit this fact. Furthermore, probability analysis of paths can determine the depths of the subtrees which would be used for directed testing.

- Handling non-linear constraints both in Stateflow and in Simulink – by use of smart heuristics from the heuristic library and feedback based techniques – is a major contribution of our approach. Our experiments suggest that Reactis does not perform well when faced with non-linearity. One important aspect of Reactis coverage is worth mentioning. For the ATC case study, we have run Reactis 25 times to generate test cases. On two occasions, the coverage was 100% but for the remaining 23 times the coverage was low. This is because of the role of Random testing on Reactis coverage; every time the coverage measures can be different (In Table 2 we have shown the average measures). This corroborates the observations made by Miller et al. [13]. From our experiments, we can infer that REDIRECT avoids this unreliability in quality but reaps the benefits of random testing.

- Currently, our method flattens parallel Stateflow charts for covering the states and transitions. This blows up the number of states and transitions. We are exploring the use of special traversal techniques that explore the states concurrently in different charts to avoid flattening. This is a part of our future work.

6. CONCLUSIONS AND FUTURE WORK

Random and Systematic testing techniques have their own strengths and weaknesses. We have used an effective mix of randomness and directed coverage in our REDIRECT approach and developed a prototype tool. We have performed several case studies over medium-sized SL/SF models and achieved better structural coverage on these case

Applications	Reactis					REDIRECT			
	params	% State cover	% Tr. cover	No. of testcases	Avg length of testcases	% State cover	% Tr. cover	No. of testcases	Avg length of testcases
Cruise Controller (5 states, 7 Tr. 23 SL blocks)	1K, 1K 20K	100	100	31	2.5	100	100	1	57
ATC (9 states, 14 Tr. 26 SL blocks)	1K, 1K, 1000K	83	71	4	460	100	100	3	336
ACC (10 states, 16 Tr. 65 SL blocks)	1K, 1K, 200K	90	75	10	354	90	75	2	44
Power Window controller (5 states, 11 Tr. 4 SL blocks)	1K, 1K, 100K	80	81	9	9	100	100	2	65
MGC (7 states, 9 Tr., 140 SL blocks)	×	×	×	×	×	100	100	3	100

Table 2: Coverage measures of SL/SF models (4 automotive and 1 aerospace)

studies than some commercial tools. Handling non-linear constraints using heuristic and feedback techniques is an important contribution of our research.

7. REFERENCES

[1] Applied Dynamics International. BEACON for Simulink/Stateflow, http://www.adi.com

[2] R. Alur, A.Kanade, S.Ramesh, and K.C. Shashidhar. Symbolic Analysis for Improving Coverage of Simulink/Stateflow Models, In EMSOFT (this Proceedings), ACM, 2008.

[3] R. Cleaveland, S.A. Smolka, and S.T. Sims. An Instrumentation-Based Approach to Controller Model Validation, In *Automotive Software Workshop, San Diego*, Available at http://aswsd.ucsd.edu/2006/pdfs/smolka-vm-slides.pdf

[4] ETSI. ETS 300 406: Methods for Testing and Specification (MTS); Protocol and profile conformance testing specifications; Standardization Methodology, *European Telecommunication Standard*, 1995.

[5] 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 Proc. of the CAV'08, LNCS Volume 5123, pages 204-208, 2008.

[6] P. Godefroid, N. Klarlund, and K. DART: Directed Automated Random Testing, In *Proc. of the PLDI'05*, Chicago, pp. 213-223, 2005.

[7] G. Hamon, L. de Moura, and J. Rushby. Automated Test Generation with SAL, *CSL Technical Note*, January 2005.

[8] ISO. OSI Conformance Testing Methodology and Framework– ISO 9646, 1998.

[9] A. Agung Julius, G. Fainekos, M. Anand, I. Lee, and G.J. Pappas. Robust test generation and coverage for hybrid systems, In *LNCS Volume 4416*, Springer, pages 329-342, 2007.

[10] R. Majumdar, and K. Sen. Hybrid Concolic Testing, In *Proc. of the ICSE*, Minneapolis, pages 416-426, 2007.

[11] The Mathworks, http://www.mathworks.com

[12] The Mathworks, Simulink Design verifier, http://www.mathworks.com

[13] S.P Miller, E.A. Anderson, L.G. Wagner, M.W. Whalen, and M.P.E. Heimdahl. Formal Verification of Flight Control Software, In *Proc. of the AIAA Guidance, Navigation and Control Conference and Exhibit*, San Francisco, pages 1-16, August 2005.

[14] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating Test Data from State Based Specifications, *Software Testing, Verification and Reliability*, John Wiley, 13(1): 25-53, March 2003.

[15] C. Pacheco, S.K. Lahiri, M. Ernst, T. Ball. Feedback-directed Random Test Generation, In *Proc. of the ICSE*, Minneapolis, pages75-84, 2007.

[16] Reactis: http://www.reactive-systems.com

[17] Reactive Systems. Model Based Testing and Validation with Reactis, Reactive Systems Inc., http://www.reactive-systems.com

[18] D.J. Richardson, A. Leif Aha, T.O. O'Malley. Specification-based Test Oracles for Reactive Systems, In Proc. of ICSE, Melbourne, pages 105-118, 1992.

[19] SRI International. SAL home page http://sal.csl.sri.com

[20] S. Sims, and D.C DuVarney. Experience Report: The Reactis Validation Tool, In *Proc. of the ACM International Conference on Functional Programming*, Freiburg, pages 137-139, October 2007.

[21] STB. Safety Test Builder, Automatic Test Generation for Simulink/Stateflow, TIN Software.

[22] T-Vec. T-Vec Tester for Simulink, http://www.t-vec.com

[23] H. Zhu, P.A.V. Hall, and J.H.R. May. Software Unit Test Coverage and Adequacy *ACM Computing Surveys*, 29(4):366-427, 1997.