Search-Based Mutation Testing for *Simulink* Models

Yuan Zhan
Department of Computer Science
University of York
York, YO10 5DD, UK
+44-1904-432749
yuan@cs.york.ac.uk

John A. Clark
Department of Computer Science
University of York
York, YO10 5DD, UK
+44-1904-433379
jac@cs.york.ac.uk

ABSTRACT

The efficient and effective generation of test-data from high-level models is of crucial importance in advanced modern software engineering. Empirical studies have shown that mutation testing is highly effective. This paper describes how search-based automatic test-data generation methods can be used to find mutation adequate test-sets for *Matlab/Simulink* models.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *testing tools (e.g. data generator, coverage testing).*

General Terms

Verification.

Keywords

Matlab/Simulink, mutation testing, test-data generation, automation, heuristic search, simulated annealing.

1. INTRODUCTION

The modern aim of 'testing' is to discover faults at the earliest possible stage because the cost of fixing an error increases with the time between its introduction and detection. Thus high-level models have become the focus of much modern-day verification effort and research. *Matlab/Simulink* [27][28][29] is a widely used notation in the dynamic system development industry that allows models to be created and exercised. *Matlab/Simulink* models can be architectural level designs of software systems. The simulation facilities allow such models to be executed and observed. This property of *Simulink* turns out to be an advantage for effective dynamic testing. In this work, we focus on automatically generating mutation adequate test-data for testing *Matlab/Simulink* models. Other authors have recognized the practical significance of such modeling and the need to provide assurance information automatically, e.g. the worst-case execution times for such models [21]. Baresel et al. [13] proposed an innovative way of generating sequences of signals for testing *Simulink* models by building the overall signal from a series of simple signal types such as step, ramp and sine curves etc.. An input sequence can be made up of a number of sections; each section is defined by a choice of signal type from the above basic types, a length and an amplitude. This can significantly reduce the search space. They also applied this technique successfully to functional test-data generation – checking speed violation for a Distronic Cruise Control System.

Most test-data generation work has focused on satisfying particular *structural coverage criteria* (refer to [19] for definition). However, sometimes the execution of the underlying structure may not discover the error(s) in it. Mutation testing focuses on measuring the quality of a test-set according to its ability to detect specific faults. With faults that cannot be detected by the test-set at hand, we may want to generate targeted test-data that can discover them. This paper presents a method of generating such test-data to complement our previous *Simulink* structural test-data generation work [17].

2. SEARCH-BASED AUTOMATIC TEST-DATA GENERATION

Test-data generation can be dynamic or static, depending on whether the execution of the test object is involved or not. Search-based test-data generation is a dynamic approach. With the guidance information obtained from dynamically running or simulating the underlying test objects, it searches the input domains of the test objects for targeted test-data. This approach has been widely applied in structural testing [1][2][3][6][7][8][9] as well as functional testing [10][11][12] and non-functional testing (which largely focused on temporal testing) [14][15]. Most search-based test-data generation research had been carried out at the code level but Jones et al. [16] have generated test-data from Z specifications. In [17] we applied our search-based approach to the generation of test-sets achieving particular structural coverage measures of *Simulink* architectural models. In this paper, we extend our previous work to achieve mutation adequate test-sets.

The search-based test-data generation process is now described.

The satisfaction of a particular test requirement is couched as a sequence of one or more predicates over the behavior of the system before, during, or after execution. For example, a specific path will be taken when the corresponding set of branch conditions hold true during execution [17]. If `X` is in the range `0..25`, then a constraint error exception may be generated at a specific assignment statement `X=Y×Y`, when the healthiness precondition `Y×Y<=25` does not hold before execution of the statement [11]. A more detailed way of specifying the overflow might consist of a sequence of predicates defining a path that reaches the statement (with no exceptions along the way) together with `Y×Y>25` immediately before the statement. Causing a program to break its functional specification can be couched as satisfying the precondition before execution and not satisfying the post-condition at the end of execution [10].

We must be able to evaluate how close a program execution comes to satisfying a predicate at a point. For example, for a predicate `X>=50`, a value of `49` for `X` would be considered 'closer' to satisfying the predicate than would a value of `20`. Costs are associated with how far a predicate is from being satisfied – the closer to satisfaction the smaller the cost. A cost of zero is assigned when the predicate is satisfied. A typical cost function scheme is illustrated in Table 1. The value *K* in the table refers to a failure constant that is added to further punish test-data that causes a term to be untrue [2]. All cost function schemes used are based around similar notions. The table includes recent enhancements by Bottaci [18].

Table 1. Cost function encoding method.

Predicate	Value of Cost Function *F*
Boolean	if TRUE then 0, else K
$E_1 < E_2$	if $E_1 - E_2 < 0$ then 0, else $E_1 - E_2 + K$
$E_1 \le E_2$	if $E_1 - E_2 \le 0$ then 0, else $E_1 - E_2 + K$
$E_1 > E_2$	if $E_2 - E_1 < 0$ then 0, else $E_2 - E_1 + K$
$E_1 \ge E_2$	if $E_2 - E_1 \le 0$ then 0, else $E_2 - E_1 + K$
$E_1 = E_2$	if $Abs(E_1 - E_2) = 0$ then 0, else $Abs(E_1 - E_2) + K$
$E_1 \ne E_2$	if $Abs(E_1 - E_2) \ne 0$ then 0, else K
$E_1 \vee E_2$ (E_1 unsatisfied, E_2 unsatisfied)	(cost (E_1) × cost (E_2))/(cost (E_1) + cost (E_2))
$E_1 \vee E_2$ (E_1 unsatisfied, E_2 satisfied)	0
$E_1 \vee E_2$ (E_1 satisfied, E_2 unsatisfied)	0
$E_1 \vee E_2$ (E_1 satisfied, E_2 satisfied)	0
$E_1 \wedge E_2$ (E_1 unsatisfied, E_2 unsatisfied)	cost (E_1) + cost (E_2)
$E_1 \wedge E_2$ (E_1 unsatisfied, E_2 satisfied)	cost (E_1)
$E_1 \wedge E_2$ (E_1 satisfied, E_2 unsatisfied)	cost (E_2)
$E_1 \wedge E_2$ (E_1 satisfied, E_2 satisfied)	0

Search based testing combines the costs of the various relevant predicates to provide an overall cost for a particular execution. (We omit details here). The aim is to reduce the overall cost to zero. The test-data generation problem becomes a cost function minimization problem; a host of optimization techniques have been adopted, e.g. simulated annealing [10], genetic algorithms [6][7][8][9], tabu search [25], and ant colony optimization [26]. Details of heuristic search approaches can be found in [20]. A full account of test-data generation by heuristic search can be found in the McMinn's extensive survey [24].

3. MUTATION TESTING

It is natural to believe that the more errors a test-set can detect, the better the test-set is. Mutation testing (proposed by DeMillo et al. [4]) is based on this concept.

Mutation testing works as follows. A large number of simple faults, such as alterations to operators, constant values or variables are introduced into the program under test, one at a time. The resulting programs are called *mutants*. The goal is to generate a test-set that can distinguish each mutant from the original program by comparing the program outputs. If a mutant can be distinguished from the original program by at least one of the test cases in the test-set, we say the mutant is *killed*. Otherwise we say that the mutant is alive. Consider the statement `x:=y+z`; two mutants of this are `x:=y–z` and `x:=y×z`. In this instance, if the input variables are y and z, and the output is variable x, then an input case (`y=0`, `z=0`), for example, cannot kill either of the mutants as the output x will be the same for all three programs (one original and two mutants). However, a test input such as (`y=1`, `z=2`) can distinguish both mutants from the original.

Sometimes the mutant cannot be killed due to the *semantic equivalence* of the mutant and the original program. (They can *never* give different results.) Thus the adequacy of a test-set can be assessed by the following equation:

$$AdequacyScore = \frac{D}{M - E}$$

where ***D*** is the number of mutants that has been killed, ***M*** is the total number of mutants, and ***E*** is the number of semantically equivalent mutants [5].

Empirical studies have shown that mutation testing is highly effective [30][31]. Two major disadvantages of mutation testing are the huge number of mutants generated, which requires very significant computation (both for compilation of the mutants and the execution of test data on them), and the cost of determining any equivalent mutants, as this is usually done by hand.

In our approach, the way we mutate models – perturbing signals (as described in the next section) – reduces the number of mutants generated significantly compared to using the method of perturbing operations. Our experience also showed that very few equivalent mutants are generated for each model. Mutants that cannot be killed by targeted test-data generation must still be examined manually to determine equivalence.

4. *MATLAB/SIMULINK* AND MODEL MUTATION

Simulink [1] is a software package for modelling, simulating, and analysing system-level designs of dynamic systems. *Simulink* models/systems are made up of blocks connected by lines. Each block implements some function on its inputs and outputs the results. Outputs of blocks form inputs to other blocks (represented by lines joining the relevant input/output ports). Models can be hierarchical. Each block can be a subsystem comprising other

[1] Developed by the MathWorks Inc: http://www.mathworks.com.

blocks and lines. Figure 1 is an illustration of a simple *Simulink* model. There are many ways to translate *Simulink* models into code. One version of the translation is given in the Appendix section.

In *Simulink*, there are certain blocks that form branches. They are: 'For', 'If', 'Multiport Switch', 'Switch', 'SwitchCase' and 'While' block. 'For', 'If', 'SwitchCase' and 'While' blocks are provided by *Simulink* for the convenience of model construction from programs. But they are not generally used in constructing control systems. In particular, they are ruled out in Rolls-Royce Controls[2]. The 'Multiport Switch' block is a derivative of the 'Switch' block. Here we only address problems concerned with the 'Switch' blocks in the current work. There is a control parameter 'threshold' associated with each 'Switch' block. If the signal carried on the second input port of the 'Switch' block 'Vp' satisfies 'Vp ≥ threshold' then input port 1 is selected for output. Otherwise, input port 3 is selected.

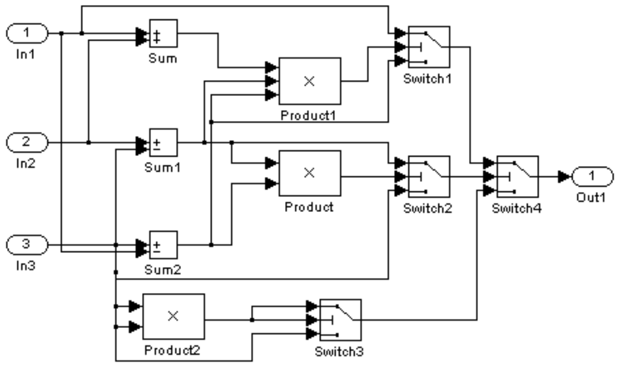

Figure 1 *Simulink* original model.

In *Simulink* all blocks execute at each time step. Thus, the traditional code-level concept of 'reaching' a block (i.e. causing it to execute) does not really occur. However, the computational outputs of some blocks may be ignored by those blocks for which they are inputs. In a switch block for example, the non-selected input is 'left hanging'. More appropriate structural coverage criteria can be defined in terms of selection of the inputs to blocks; a block could be thought of as fully exercised if each of its inputs is selected during the execution of at least one test, allowing that input to propagate further [17].

As described in the previous section, mutation testing focuses on measuring the quality of a test-set according to its ability to detect specific faults. To test *Simulink* models, we systematically introduce faults into the model and see how many of these are 'discovered' by the test-set. The more the faults that can be 'discovered', the better the test-set.

In our approach, errors are introduced to the system by perturbing the values of signals carried on wires/lines rather than the operation performed within blocks. For example, in the system illustrated in Figure 1, a mutant model can be created by inserting a mutation block 'AddMut' into the wires connecting block 'Sum' and block 'Product1' in the model, as illustrated in Figure 2. Such perturbation can be used to model initialization faults, assignment faults, condition check faults and even function/subsystem faults.

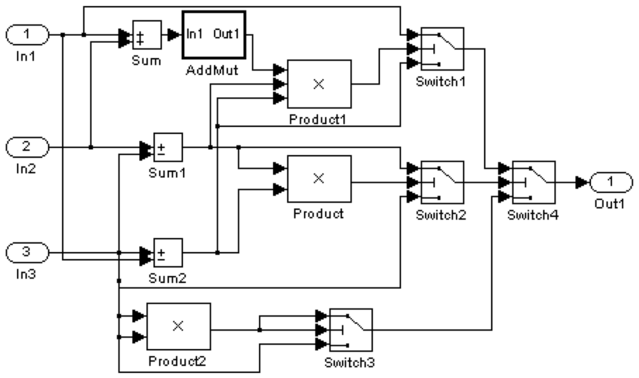

Figure 2 *Simulink* mutant model.

We defined three types of mutation operators: *Add*, *Multiply*, and *Assign*. These represent adding, multiplying, or assigning the signal carried on the input by/with a certain value. These mutation operators are built into a *Simulink* library, each as a subsystem that can be integrated with other models. A '*Mask Parameter*'[3] is associated with each mutation operator, which defines the 'certain value' as mentioned above. In our system, it is called '*Mutation Parameter*'. Given the mutation operator, the mutation parameter, and the fault injection position (which defines a signal connecting two ports between two blocks, of which, the necessary information includes Source Block Name, Source Port Number, Destination Block Name, and Destination Port Number), a mutant can be automatically generated by our system. By enumerating all signals within a model and applying appropriate[4] perturbation methods (a perturbation method is defined by the combination of a mutation operator and a mutation parameter) to them, faults are systematically introduced into models under test.

5. SEARCH-BASED MUTATION TESTING FOR *SIMULINK* MODELS

5.1 Converting a Mutation Testing Test-Data Generation Problem into an Optimization Problem

In mutation testing, we seek test-data that 'kills' the generated mutants (i.e. detects the faults hidden in the mutants). We assume that a fault can be detected by a test-datum if and only if the test-datum causes the final output vector of the mutant to be different from the output vector of the original model. If the system under

[2] This work is sponsored by the Rolls-Royce University Technology Center.

[3] *Simulink* allows users to build masked subsystems with one or more 'mask parameter(s)' left open for control by potential subsystem users.

[4] The purpose of perturbing signals is to imitate errors that might occur within system implementation. Different types of signal should be perturbed by different means. A Boolean signal only needs two types of perturbation (assign with '1/TRUE' and assign with '0/FALSE'). However, a real type of signal can be perturbed with all kinds of combinations of mutation operators and parameters.

test has more than one output variable/signal, a test-datum that causes differentiation of any one or more of the output variables/signals between the two models would be considered effective. Therefore, the evaluation of how well an underlying test-datum is satisfying a mutation testing requirement (i.e. fault detection requirement) should be based on how far the injected fault propagates within the system under test on any of the path/paths between the fault's introduction and the output. Similar to the way we couched structural test generation as a search problem [17], we assign a large value to test-data that are poor in propagating the fault, assign a small value to test-data that are good in propagating the fault but fail in fully propagating the fault, and assign zero to test-data that can reveal the fault in the output.

5.2 Cost Function Construction

To detect how far the fault has propagates we need to compare the runtime states of the original and of the mutant model. So the dynamic test-data generation process involves the execution of both of the models. Appropriate probes[5] must be inserted into both models to provide the necessary runtime information.

To meet the goal of having different outputs between the original model and the faulty model, we need to ensure two things happen:

1. The signal values at/after the point where fault is injected are different.

2. The difference ripples to the outputs.

The first requirement is normally easily achieved. Usually, unless the fault we inject is an ineffective fault (e.g. add 0 or multiply by 1), the value of the mutated signal tends to be different from that of the original one. There are some special occasions where the two values may be equal. For example, the original signal has a value of 0 and the mutation is to multiply the value by a certain value, say 100, or the original signal value is 1 and the mutation is to assign the value with 1. In these cases, we just need to tune the input vector to make the signal values at the fault injection point different from those specific values. Usually the goal can be achieved just by tuning the inputs randomly. Therefore, our strategy is to consider it as one approach level. The term of 'approach level' has been used in other work [24].

However, it is much more difficult to cause the input to make the difference at the fault-injection point affect the outputs. To fulfill this requirement, we need to trace down the structure of the model and make sure each point on the path from the fault-injection point through to the output differs between the two models. There may be a number of paths from the fault-injection point to any of the output ports of the system. In that case, we require at least one of them to propagate the difference (show the error). On each path, every block the signal passes through has the potential to disguising the fault and therefore it is sensible to break the fault propagation requirement down into a number of approach levels.

In *Simulink*, most functional blocks produce different outputs when the inputs change (e.g., mathematical blocks). Due to the special function of 'Switch' blocks (as introduced in section 4), changes (errors) are often masked by them for certain inputs. In order to provide more effective guidance to the targeted test-data search, we want to identify such positions where 'Switch' blocks might disguise the error, detect the information of their branching status, and apply such information in the test-data evaluation.

[5] Probe insertion is the activity of instrumenting a model so as to reveal certain internal system states during execution. For our *Simulink* models, a probe is implemented as an output block connected to the desired signal.

For 'Switch' blocks, the evaluation of a test-datum for detecting a particular error can be defined in the following way:

1. If there is a point in the mutant system where the value carried on the wire may be different from the corresponding point in the original system, and this point is connected to only one other block (non-Switch), as shown in Figure 3[6], then the cost will be $C = C_D + C_{OP} + C_R$, where C_D is the cost of causing this point to make a difference between the two models; C_{OP} is the cost of causing the difference to show at point P (in other words, to show after going through the operation of block 'OP'); and C_R is the cost of causing the difference to ripple after the 'OP' block.

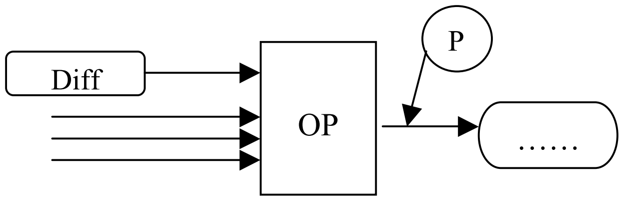

Figure 3

2. If there is a point in the mutant system where the value carried on the wire may be different from the corresponding point in the original system, and this point is connected to more than one other block, for example, the point is connected to two other blocks, as shown in Figure 4, then the cost will be $C = C_D + (C_{P1} \vee C_{P2})$[7], where C_D is the cost of causing this point to make a difference between the two models, $C_{P1} = C_{OP1} + C_{RP1}$ and $C_{P2} = C_{OP2} + C_{RP2}$. C_{OP1} represents the cost of causing the difference to show at point P1 (in other words, to show after going through the operation of block 'OP1'), and C_{RP1} represents the cost of causing the difference to ripple after the 'OP1' block. C_{OP2} and C_{RP2} are defined likewise.

[6] In the figure, the round-cornered rectangle labeled with a 'Diff' represents the point in the model where the value carried on the wire may be different between the two models; the rectangle labeled with an 'OP' represents an operational block; the circle libeled with a 'P' represents the point where a probe is inserted.

[7] Here the cost of $(C_{P1} \vee C_{P2})$ will be evaluated as the cost of either satisfying the predicate formula constructed at P1 or satisfying the predicate formula constructed at P2. Cost function evaluation of logical predicates is defined in section 2.

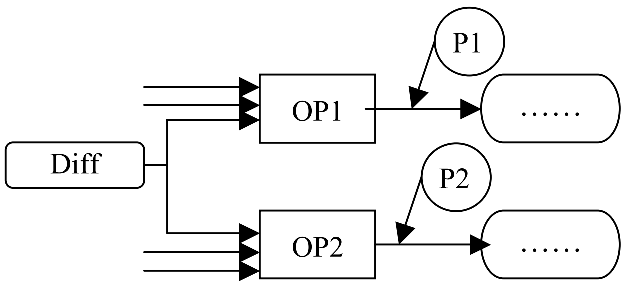

Figure 4

3. If there is a point in the mutant system where the value carried on the wire may be different from the corresponding point in the original system, and this point is connected to the first or third in-port of a 'Switch' block, as shown in Figure 5 and Figure 6, then the cost will be $C = C_D + C_{P1} + C_R$, where C_D is the cost of causing this point to make a difference between the two models, C_{P1} is the cost of causing the value at point P1 to satisfy (for the scenario in Figure 5) or dissatisfy (for the scenario in Figure 6) the first-input branching requirement[8] of the 'Switch' block and C_R is the cost of causing the difference to ripple after the 'Switch' block.

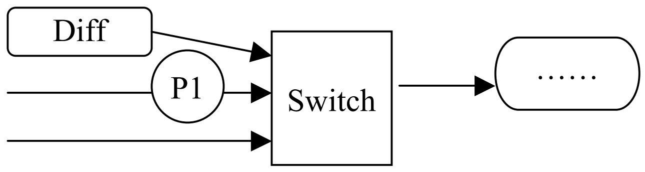

Figure 5

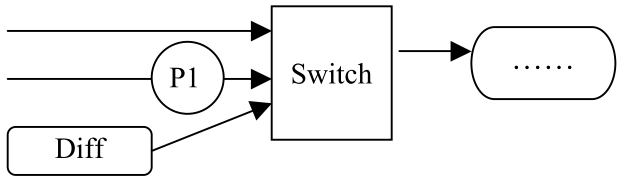

Figure 6

4. If there is a point in the mutant system where the value carried on the wire may be different from the corresponding point in the original system, and this point is connected to the second in-port of a 'Switch' block, as shown in Figure 7, then the cost will be $C = C_D + (C_{P1P2} + C_{P3}) \vee (C'_{P1P2} + C'_{P3}) + C_R$, where: C_D is the cost of causing values carried at this point to make a difference between the two models; C_{P1P2} is the cost of causing the value at point P1 in the mutant model to be different from the value at point P2 in the original model; C_{P3} is the cost of causing the value at P3 to satisfy the first-input branching requirement of the 'Switch' block in the mutant model but to dissatisfy the first-input branching requirement in the original model; C'_{P1P2} is the cost of causing the value at point P2 in the mutant model to be different from the value at point P1 in the original model; C'_{P3} is the cost of causing the value at P3 to dissatisfy the first-input branching requirement of the 'Switch' block in the mutant model but to satisfy the first-input branching requirement in the original model; and C_R is the cost of causing the difference to ripple after the 'Switch' block.

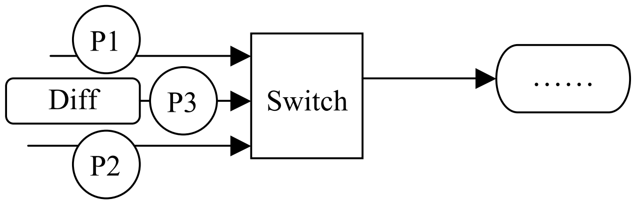

Figure 7

With the rules defined above, we will be able to evaluate the cost of moving from a test-datum to the targeted test-datum by applying these rules recursively. The starting point should be the point where a fault is injected and initially that point is the only 'Diff' point. Later on, the point where a 'C_R' is evaluated will be a new 'Diff' point. In the rules above, the basic evaluations are C_{OP}, C_{OP1}, C_{OP2}, C_{P1}, C_{P1P2}, C_{P3}, C'_{P1P2}, and C'_{P3}. These cost evaluations are usually interpreted as the cost of fulfilling a relational predicate or a logical combination of a number of relational predicates. Detailed cost function evaluation of relational and logical predicates as illustrated in Table 1 is adopted.

For example, based on these rules, the cost function of the problem as illustrated in Figure 1 and Figure 2 can be built as below:

Probe insertion (Due to the space limit, we cannot provide a picture. Readers are advised to draw by hand on Figure 2 according to the following instructions to acquire a better understanding):

P1: between AddMut(out1) and Product1(in1) for the mutant; and between Sum(out1) and Product1(in1) for the original model.

P2: between Product1(out1) and Switch1(in2) for both models.

P3: between In1(out1) and Switch1(in1) for both models.

P4: between Sum2(out1) and Switch1(in3) for both models.

P5: between Switch2(out1) and Switch4(in2) for both models.

Given the notion that 'V' represents value of probe, 'TH' represents threshold parameter of a switch block, and 'Or' and 'Mu' in subscript font represent values obtained from original model and mutant model respectively, the cost function is therefore:

$C = C_{P1} + C_{P2} + (C_{P3P4} + C_{P2}) \vee (C'_{P3P4} + C'_{P2}) + C_{P5}$

[8] First-input branching requirement is the branching requirement defined for the underlying 'Switch' block to pass its first input through.

$C_{P1} = \text{cost}(V_{P1Or} \neq V_{P1Mu})$ …rule1

$C_{P2} = \text{cost}(V_{P2Or} \neq V_{P2Mu})$ …rule1

$C_{P3P4} = \text{cost}(V_{P3Or} \neq V_{P4Mu})$ …rule4

$C_{P2} = \text{cost}(\ (V_{P2Or} \geq TH_{Switch1}) \wedge (V_{P2Mu} < TH_{Switch1})\ )$ …rule4

$C'_{P3P4} = \text{cost}(V_{P4Or} \neq V_{P3Mu})$ …rule4

$C'_{P2} = \text{cost}(\ (V_{P2Or} < TH_{Switch1}) \wedge (V_{P2Mu} \geq TH_{Switch1})\ )$ …rule4

$C_{P5} = \text{cost}(V_{P5Or} \geq TH_{Switch4})$ …rule3

The above cost function construction ideas are only general principles. In real applications, the evaluation of C_{OP}, C_{OP1} and C_{OP2} is not that easy. According to the cost function evaluation method defined in Table 1, the cost of causing $E_1 \neq E_2$ is either a K or 0. Currently we assign a value of '10' to K. However, such an evaluation may not represent the hardness of satisfying the predicate accurately. This problem will be discussed in our future work. Another problem is that whatever a value of K we decide on, it does not give much guidance to the search when two different inputs both get 'trapped' by block 'OP', or 'OP1' or 'OP2'. To address this problem, we should assess these costs according to the functionality carried out by the block 'OP', 'OP1' or 'OP2'. If this block is a subsystem, then detailed analysis into the block needs to be carried out in order to give a meaningful evaluation (the analysis can be performed by recursively applying the cost evaluation rules given above). For a basic block[9], if this block is a logical block[10], to give a good cost evaluation that can reflect the test-datum quality, we need to chain-back to obtain the information of what contributes to the evaluation of the Boolean inputs of the logical block and form the cost function with that information. Similarly if the basic block is a relational block, we need to find out its input values and relational parameter ('=', '≠', '>', '≥', '<', or '≤') so as to give more subtle evaluation. Currently in the prototype testing system we do not address this kind of situation and we deal with this kind of block in the same way as dealing with other basic blocks.

5.3 Targeted Test-Data Search

In this work, we have used the well-established technique of simulated annealing [22] to search for the desired test-data. Simulated annealing is a global optimization heuristic that is based on the *local* descent search strategy. Interested readers are referred to [23], [22] and [20] for more details about the annealing algorithm. In our application a move effectively perturbs the value of one of the inputs in the current test sequence by a value less than or equal to 1 percent of the range of the input. We applied a geometric cooling rate of 0.9. The number of attempted moves at each temperature was 500, with a maximum of 100 iterations (temperature reductions) and a maximum number of 30 consecutive unproductive iterations (i.e. with no move being accepted). These parameters may be thought to be on the 'small' side, but the computational expense of simulation requires us to make pragmatic choices.

[9] A basic block is a non-subsystem block.

[10] A logical block is a block that carries out logical calculations, provided by *Matlab/Simulink*.

6. CASE STUDY

We propose a testing strategy that combines random testing and targeted heuristic search based testing. This is because random test-sets are usually a cheap way of achieving a moderate coverage. For easy problems, it usually consumes less computation compared to the search-based test-data generation approach we provide. In our experiments, we start with generating a moderate amount (say 10,000) of test inputs randomly and run each of them on both the original model and every mutant model, check if the mutant can be killed by at least one of the random test-data. For those mutants that cannot be 'killed', we use heuristic search to target each mutant-killing aim and generate desired test-data.

Table 2 shows the description of some hard mutant-killing oriented test-data generation problems (none of the testing aims involved in these testing problems could be met by the structural adequate test-sets we generated) and Table 3 shows the corresponding comparison result of costs (in terms of time and number of cases tried) in solving them between using simulated annealing search and random test-data search. In Table 2, each test-data generation task is specified with the name of the model under test and the fault description (including mutation operator, mutation parameter, and fault-injection position, which in turn is made up of details of source block name, source port number, destination block name, and destination port number). Model 'RandMdl' is the model in Figure 1. 'Quadratic' is a model that has similar size and function as 'RandMdl'. 'DetectDuplexFaults' is a real model from industry. The result in Table 3 is based on the average of 30 individual runs of the program. All the test generation tasks demonstrated here are selected because a random test-set of 10,000 test cases failed in generating satisfactory tests for these aims.

Here we give an illustration of what the data in the tables represent. The first row of data means: in the 'Quadratic' model, we inject a fault on the wire connecting block 'Product2' port '1' and block 'Switch2' port '2'. The fault adds '1' to the value carried on this wire. The simulated annealing search based test-data generation tool consumed 15,708.4 tries on average (the result is based on 30 individual runs, and all runs were successful) to find the desired test-data. The random test-data generation would have cost more than 85,789.4 tries on average (the result is also based on 30 individual runs, but only 6 of them produced successful outcomes) to find the appropriate test-data. The simulated annealing (SA) runs were allowed up to 100 external loops and 500 internal loops. The simulated annealing search usually spends a few thousands tries in searching for a feasible *initial temperature*. Therefore, if a SA search fails, the cost of cases tried would be more than 50,000. On the other hand, the search time cost of each try is different between SA search and random search. SA search costs more in the overheads of computing the moves, evaluating feasibility of test-data, etc. We allow the random test-data generation up to 100,000 tries of different test-inputs (so that the time allowance of each search attempt for the random approach is no less than that is allowed for the SA search). For some models, each run produced a successful outcome with this limit. In others, only some of the runs did so. For failed runs, we prefix the (Search Time) or (Case No Tried) with a '>' to symbolize the real cost of generating such a test input should be larger than this figure. The numbers in parenthesis are the number of successful runs against the number of total runs.

As can be seen, the search-based test-data generation approach enlarges the fault detection capability of test-sets whilst saving computation cost in achieving goals compared to random testing.

Table 2 Problem description.

Prob-lem	Model Name	Mut Op	Mut Para	Src Block	Src Port	Dst Block	Dst Port
1	Quad-ratic	Add	1	Pro-duct2	1	Switch2	2
2	Quad-ratic	Add	1	Sum	1	Product	1
3	RandMdl	Add	1	Pro-duct	1	Switch2	2
4	RandMdl	Add	1	Pro-duct1	1	Switch1	2
5	RandMdl	Add	-1	Pro-duct2	1	Switch3	2
6	DetectDu-plexFaults	Add	-1	DTC6	1	Relation-al Op	2

Table 3 Test-data search cost comparison.

Problem	SA Case No	Random Case No	SA Time (Seconds)	Random Time (Seconds)
1	15,708.4 (30/30)	>85,789.4 (6/30)	324.8 (30/30)	>1,692 (6/30)
2	10,529.8 (30/30)	>88,031.7 (7/30)	335.5 (30/30)	>2,649.8 (7/30)
3	8,008.1 (30/30)	>26,219.5 (28/30)	100.49 (30/30)	>380.33 (28/30)
4	>20,513.9 (21/30)	>76,829.1 (11/30)	>355.3 (21/30)	>1,239.3 (11/30)
5	>22,518.1 (25/30)	>33,805.2 (22/30)	>844.4 (25/30)	>1,195.9 (22/30)
6	760.3 (30/30)	25100 (30/30)	27.5 (30/30)	843.7 (30/30)

7. CONCLUSION AND FUTURE WORK

Evidence has shown structural coverage sufficient tests cannot guarantee errors will be detected. To address this problem, we propose an approach to generate mutation adequate test-sets. In our system, mutants can be generated automatically and systematically. In order to efficiently generate a sufficient set of test-data that can kill all the mutants, we use a two-prong approach. We randomly generate a large set of test-data, detect their mutant-killing ability, and then minimize the test-set whilst retaining its overall mutant-killing ability. For mutants that cannot be killed by the random test-set, we provide an effective means of automatically generating individual test-data for fulfilling individual mutant-killing aims. In this way we use targeted test-data generation to complement random test generation so as to achieve mutation adequacy.

Testing and analysis at high-level stages has become a crucial part of effective software development. Here, we have chosen to work on *Matlab/Simulink* models. We believe that the conceptual test-generation framework could extend to other architectural or modelling notations provided that the notation selected supports execution or simulation.

The automatic test-data generation method we present is conceptually extensible. We intend to extend the test-data generation approach to other types of test-data generation, such as exception testing, safety assertion testing etc.

8. ACKNOWLEDGMENTS

Our thanks to Rolls-Royce for sponsoring this research.

9. REFERENCES

[1] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering, 16(8): 870-879*, 1990.

[2] N. Tracey, J. Clark, K. Mander, and J. McDermid. An Automated Framework for Structural Test-Data Generation. *Int'l Conf. on Automated Software Engineering, pages 285-288*, 1998.

[3] J. Wegener, K. Buhr, and H. Pohlheim. Automatic Test Data Generation for Structural Testing of Embedded Software Systems by Evolutionary Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), pages 1233-1240*.

[4] R. DeMillo, R Lipton, and F. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE computer, 11: 34-41*, 1978.

[5] Jeffrey Voas and Gary McGraw: *Software Fault Injection: Innoculating Programs Against Errors*. By John Wiley & Sons, 1997.

[6] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gal, S. Katsikas and K. Karapoulios. Application of Genetic Algorithms to Software Testing. In *Int'l Conf. on Software Engineering and its Applications, pages 625-636*, 1992.

[7] A. Watkins. The Automatic Generation of Test Data Using Genetic Algorithms. In *Proceedings of the Fourth Software Quality Conference, pages 300-309*, 1995.

[8] R. Pargas, M. Harrold, and R. Peck. Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification and Reliability. 9(4): 263-282*, 1999.

[9] B. Jones, H. Sthamer, and D. Eyres. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal, 11(5): 299-306*, 1996.

[10] N. Tracey, J. Clark, and K. Mander. Automated Program Flaw Finding Using Simulated Annealing. *ACM/SIGSOFT Symposium on Software Testing and Analysis (ISSTA 1998), pages 73-81*. 1998.

[11] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated Test Data Generation for Exception Conditions. *Software – Practice and Experience, 30(1): 61-79*, 2000.

[12] O. Buehler and J. Wegener. Evolutionary Functional Testing of an Automated Parking System. In International Conference on Computer, Communication and Control Technologies (CCCT'03) and The 9th International Conference on Information Systems Analysis and Synthesis, (ISAS'03), Orlando, Florida, USA, 2003.

[13] A. Baresel, H. Pohlheim, and S. Sadeghipour. Structural and Functional Sequence Test of Dynamic and State-Based Software with Evolutionary Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), LNCS vol. 2724, pages 2428-2441*

[14] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer and B. Jones. Systematic Testing of Real-Time Systems. *Proceedings of the 4th European Conference on Software Testing, Analysis & Review (EuroSTAR '1996)*, Amsterdam, Netherlands, December 1996.

[15] P. Puschner and R. Nossal. Testing the Results of Static Worst-Case Execution-Time Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium, pages 134-143*, 1998.

[16] B. Jones, H. Sthamer, X. Yang, and D. Eyres. The Automatic Generation of Software Test Data Sets Using Adaptive Search Techniques. In *Proceedings of the 3rd International Conference on Software Quality Management, pages 435-444*, 1995.

[17] Y. Zhan, and J. Clark. Search Based Automatic Test-Data Generation at an Architectural Level. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004), LNCS vol. 3103, pages 1413-1426*.

[18] Leonardo Bottaci. Predicate Expression Cost Functions to Guide Evolutionary Search for Test Data. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), LNCS vol. 2724, pages 2455-2464*.

[19] Hong Zhu, Patrick A. V. Hall and John H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys, Vol. 29(4): 366-427*. December 1997.

[20] C. R. Reeves (Ed.). *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Publications, Oxford, 1993.

[21] R. Kirner, R. Lang, G. Freiberger and P. Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. 14th Euromicro International Conference on Real-Time Systems, ECRTS'02, Vienna, Austria, 2002.

[22] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. *Science, 220(4598): 671-680*, May 1983.

[23] N. Metropolis, A. W. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculation by Fast Computing Machine. *Journal of Chem. Phys., 21:1087-1091*, 1953.

[24] P. McMinn. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability, 14(2), pages 105-156*, June 2004.

[25] Eugenia Díaz, Javier Tuya, Raquel Blanco. Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search. In 18th IEEE International Conference on Automated Software Engineering. Montreal, Canada, Oct. 2003.

[26] P. McMinn and M. Holcombe. The State Problem for Evolutionary Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), LNCS vol. 2724, pages 2488-2500.*

[27] The MathWorks. *Using Simulink – Model-Based and System-Based Design*. The MathWorks., Inc. 2002.

[28] The MathWorks. http://www.mathworks.com/products/simulink

[29] Juan Carlos Cockburn. Matlab/Simulink internal resource. http://www.eng.fsu.edu/~cockburn/matlab/matlab_help.html

[30] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang. An Experimental Evaluation of Data Flow and Mutation Testing. In *Software – Practice and Experience, 26(2), 165-176*, 1996.

[31] P. G. Frankl, S. N. Weiss, and C. Hu. All-Uses vs. Mutation Testing: An Experimental Comparison of Effectiveness. *Journal of Systems and Software, 38(3), 235-253*, 1997.

10. APPENDIX

Pseudo C code equivalence of the system described in Figure 1 is:

```
double  Out1  =  calculate(In1:double,  In2:
double, In3: double);
tmpSum = In1+In2;
tmpSum1 = In2–In3;
tmpSum2 = In3–In1;
tmpProduct = tmpSum1×tmpSum2;
tmpProduct1 = tmpSum×tmpSum1×tmpSum2;
tmpProduct2 = In3×In3;
if tmpProduct1 >= thresholdSwitch1
       tmpSwitch1 = In1;
else   tmpSwitch1 = tmpSum2;     end;
if tmpProduct >= thresholdSwitch2
       tmpSwitch2 = tmpSum1;
else   tmpSwitch2 = In3;   end;
if tmpProduct2 >= thresholdSwitch3
       tmpSwitch3 = tmpProduct2;
else   tmpSwitch3 = In3;   end;
if tmpSwitch2 >= thresholdSwitch4
       Out1 = tmpSwitch1;
else   Out1 = tmpSwitch3;  end
```