A search-based framework for automatic testing of *MATLAB/Simulink* models

Yuan Zhan *, John A. Clark

Department of Computer Science, University of York, York YO10 5DD, UK

Available online 9 June 2007

Abstract

Search-based test-data generation has proved successful for code-level testing but almost no search-based work has been carried out at higher levels of abstraction. In this paper the application of such approaches at the higher levels of abstraction offered by *MATLAB/ Simulink* models is investigated and a wide-ranging framework for test-data generation and management is presented. Model-level analogues of code-level structural coverage criteria are presented and search-based approaches to achieving them are described. The paper also describes the first search-based approach to the generation of mutant-killing test data, addressing a fundamental limitation of mutation testing. Some problems remain whatever the level of abstraction considered. In particular, complexity introduced by the presence of persistent state when generating test sequences is as much a challenge at the *Simulink* model level as it has been found to be at the code level. The framework addresses this problem. Finally, a flexible approach to test sub-set extraction is presented, allowing testing resources to be deployed effectively and efficiently.
© 2007 Elsevier Inc. All rights reserved.

Keywords: *MATLAB/Simulink*; Test-data generation; Automation; Structural coverage; Mutation testing; State problem; Tracing and deducing; Test-set reduction

1. Introduction

Software testing is an expensive procedure. It typically consumes more than 50% of the total development budget (Beizer, 1990). Failure to detect errors can result in significant financial loss or even disaster in the case of safety critical systems. It is desirable, therefore, to seek techniques that will achieve testing rigour (i.e., be effective) at an acceptable cost (i.e., be efficient).

The modern aim of 'testing' is to discover faults at the earliest possible stage as 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 is a software package for modelling, simulating and analysing system-level designs of dynamic systems. Many designers choose to model designs using *Simulink* and generate code automatically from them. These models are sometimes considered by industry as architectural level designs of software systems. *Simulink* has been widely used in designing large-scale embedded systems, including engine controllers. A prominent feature of such systems is that they are of high complexity and with a need for high integrity. Thus, the verification and validation processes for such systems need to be stringent. The facilities provided by the *Simulink* tools *allow* models to be executed (simulated) and observed. This is advantageous for effective dynamic testing. A *Simulink* model can serve many purposes in testing: as a model from which test data can be generated, as reference model for test coverage, as a source for the generation of test oracles, and as a test object in its own right.

Test-data generation is one of the most tedious tasks in the software testing process. As system size grows, manual test-data generation places a great strain on resources (both mental resources and budget). This problem becomes

---

* Corresponding author.
E-mail addresses: yyzhan@yahoo.com (Y. Zhan), jac@cs.york.ac.uk (J.A. Clark).

especially serious when developers need to achieve high confidence in the system's operation. Automated test-data generation is *one* way forward. Automated test-data generation is perhaps the "Holy Grail" of software testing and new approaches are needed if there is to be a step change in achievement.

Search-based test-data generation techniques have been widely investigated for use in code-level testing. Testing of various functional and non-functional properties has been targeted, as has fulfilling certain structural coverage criteria. Almost no search-based test-data generation work has been carried out above the code level. The framework presented in this paper interprets code-level use of such techniques in the context of *Simulink* models and provides significant extensions to the search-based paradigm.

A model-level analogue of code branch coverage is presented and a search-based solution is described. This is a natural extension of existing code-level work. The most stringent low-level adequacy criterion is provided by mutation testing. Unfortunately, finding test data that kill mutants is expensive. Mutation testing is, however, very powerful. Mutation adequacy typically subsumes many structural coverage criteria and may also detect faults that structural testing has little chance of discovering. In the case of safety critical systems, it may well be worth the cost of devoting significant effort in generating mutation-adequate test-sets to test the system and gain more confidence in the software product, especially when the test-data generation process can be automated. The framework adopts a search-based approach to mutant-killing test-data generation. (There would appear to be no other search-based approach to mutant killing.)

The presence of persistent state has been shown to be problematic at the code level and such difficulties remain when *Simulink* models are to be tested. In such cases, sequences of inputs that can put the model under test into particular states are needed to enable the underlying test goals to be achieved. Simple search guidance appears to be insufficient and results in a 'flat' cost-function landscape. Our work uses a conditional back-propagation approach (which we term *Tracing and Deducing*) over *Simulink* models to provide more sophisticated guidance.

Although the emphasis is placed on *test-data* generation (i.e., inputs), the final aim is to create suitable sets of *test cases* (i.e., we need to calculate the expected *results*). This may be entirely manual and consequently very expensive. Choosing which set of tests is an important issue. Testers may seek the most effective test-set given a particular budget, or seek the cheapest test-set achieving a specific aim. The framework provides a template for describing various test aims and uses search-based approach to extract an optimal sub-set for the specified criteria.

Preliminary experimentation is carried out to show that the proposed techniques are suited to their tasks and that the testing framework effectively and efficiently generates test-sets for *Simulink* models.

Some work relevant to that presented herein has previously been published in the following papers: (Zhan and Clark, 2004; Zhan and Clark, 2005, 2006). The following sections in this paper differ from or expand upon those other papers. Section 4 in this paper extends the work in Zhan and Clark (2004) by investigating branch coverage achieved by the proposed search technique whilst (Zhan and Clark, 2004) focused on path coverage. Section 5 in this paper re-modelled the logic behind the techniques presented in Zhan and Clark (2005) and renovated the experiments. Section 6 here is an extract of Zhan and Clark (2006). This work also aims to integrate the different topics covered in those papers into an encompassing framework.

2. Background

This section covers three topics. Firstly we will give an introduction to *MATLAB/Simulink*, and provide an interpretation of coverage measurements for *Simulink*. Then we review the application of search-based techniques in software testing. Finally, an overview of random testing will be presented.

2.1. MATLAB/Simulink and test coverage

2.1.1. Simulink introduction

Simulink is a software package for modelling, simulating and analysing system-level designs of dynamic systems. Its developer **The MathWorks** (The MathWorks) describes it as "*a platform for multidomain simulation and Model-Based Design for dynamic systems. It provides an interactive graphical environment and a customisable set of block libraries, and can be extended for specialized applications.*" In modern dynamic systems development, the code may be generated automatically from *Simulink* model designs.

Simulink models 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. Models can be hierarchical. Each block can be a subsystem comprising other blocks and lines. This feature allows *Simulink* to handle complexity. Fig. 1 is a simple *Simulink* model. It determines whether an equation of form $ax^2 + bx + c = 0$ is a quadratic equation (i.e.,
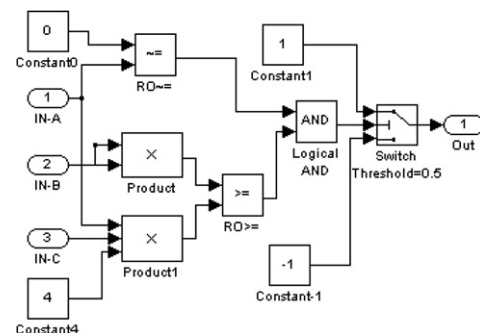

Fig. 1. A simple *Simulink* model.

$a \neq 0$) and if it has real-valued solution(s) (i.e., $b^2 - 4ac \geq 0$). If true, '1' is output; otherwise, '-1' is output. 'IN-A', 'IN-B' and 'IN-C' are three input blocks, receiving input values of 'a', 'b' and 'c' from the user. Block 'Out' provides the output.

A Switch block is commonly used for implementing branching in *Simulink*. There is a control parameter 'Threshold' associated with each Switch block. If the signal carried on its second-input port 'V_{SW2}' satisfies '$V_{SW2} \geq Thres$' then input 1 is selected as the output. Otherwise, input 3 is selected. Fig. 1 contains a Switch block on the right, with input 1 having constant value 1 and input 3 having constant value –1.

A LogicalOperator block (e.g., block 'Logical AND' in Fig. 1) has two parameters: an operator (which can be 'AND', 'OR', 'NAND', 'NOR', 'XOR', or 'NOT') and an input-number (which specifies the number of required or allowed inputs).

A RelationalOperator block (e.g., block 'RO~=' and 'RO>=' in Fig. 1) has two inputs. There is a parameter defining the desired relation between the two inputs. If the relation is TRUE, the output will be '1'; otherwise, the output will be '0'. There are six options for the relational parameter: ==, ~=, <, <=, >= and >.

Simulink is generally used for designing embedded systems. Such systems typically maintain state. The systems have continuous inputs and outputs and the execution step is controlled by some timer trigger, e.g., a step size can be 1 ms. Therefore, for the model in Fig. 1, the input to the system over n time steps should be a sequence $\langle(\text{IN-A}_1, \text{IN-B}_1, \text{IN-C}_1), (\text{IN-A}_2, \text{IN-B}_2, \text{IN-C}_2), \ldots, (\text{IN-A}_n, \text{IN-B}_n, \text{IN-C}_n)\rangle$, and the corresponding output should also be a sequence $\langle\text{Out}_1, \text{Out}_2, \ldots, \text{Out}_n\rangle$.

The executable feature of *Simulink* allows dynamic testing to be carried out at the software *design* stage. Reduced cost results from the earlier detection of errors. We are aware that automatic code generation tools are available and test-data can be generated using traditional techniques from the resulting code without much delay. However, industrial users still prefer to manually code their critical software applications for efficiency and scheduling reasons. In such cases, testing for *Simulink* models is desirable. On the other hand, test data generated from the *Simulink* models are not limited to testing *Simulink* models only. They can be also used to test the conformance between the models and the code or even testing the code alone, i.e., we can use Simulink simulation facilities as a test oracle.

To enable *Simulink* models to be thoroughly tested, stringent coverage metrics need to be applied. The interpretation of code-level structural coverage and mutation coverage for *Simulink* is now discussed.

2.1.2. Structural coverage for Simulink

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, some blocks have conditional selection behaviours, which is analogous to the behaviours caused by branches in code. Therefore, analogous code-level structural coverage (Zhu et al., 1997) can be defined. In this paper, we adopt the *Branch Coverage* definition by *Reactis Tester* (Reactive Systems Inc.). The **Branch Coverage** criterion requires all conditional behaviours of blocks, provided the block has conditional behaviours, to be executed at least once. For example, a logical operator has two conditional behaviours: 'TRUE' and 'FALSE'. Branch coverage requires that each behaviour be exhibited by at least one test execution.

Based on the definition of *Branch Coverage*, a more stringent coverage criterion like **Path Coverage** can be defined as: all combinations of various identified branch choices are exercised. Many other coverage criteria for *Simulink* models have been defined by *Simulink Verification & Validation* (also provided by **The MathWorks**) and *Reactis Tester*, such as, decision coverage, condition coverage, MC/DC coverage, etc. This work will focus on the generation of adequate *branch-coverage* tests. In *Reactis Tester*, blocks considered for the *branch coverage* analysis purpose include: DeadZone, LogicalOperator, MinMax, MultiportSwitch, RelationalOperator, Saturation and Switch. Section 4 will investigate test-data generation techniques for achieving branch coverage for *Simulink*.

2.1.3. Mutation coverage for Simulink

Mutation testing (Voas and McGraw, 1997) measures the quality of a test-set according to its ability to detect specific faults. It 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. 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 (the original and the two mutants). However, an input such as (y = 1, z = 2) can distinguish both from the original.

Sometimes the mutant cannot be killed due to the *semantic equivalence* of the mutant and the original program. (No input exists that can distinguish the programs.) 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 killed by the test-set, M is the total number of mutants, and E is the number of
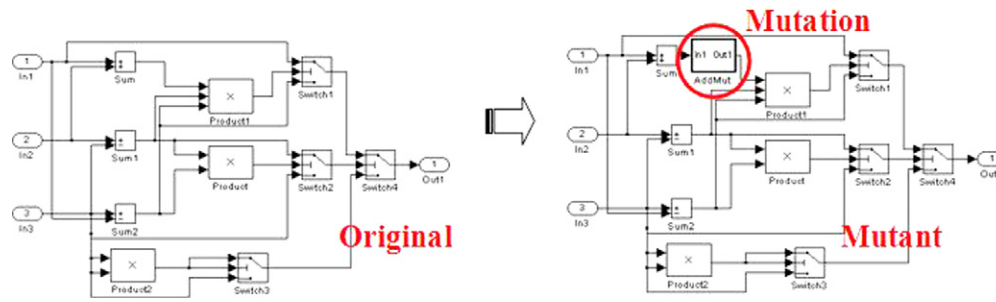
Fig. 2. An example of model mutation.

mutants being semantically equivalent with the original program ($M - E \geqslant 1$).

Empirical studies have shown that mutation testing is highly effective and mutation adequacy typically subsumes many other kinds of coverage criteria, such as statement and branch coverage. However, the technique may be computationally expensive (both for compilation of large numbers of mutants and for the execution of test data on them), and very substantial manual costs may be incurred determining mutant equivalence. Finding test data to kill mutants may be a highly laborious (and typically manual) task. Successful automation of this aspect would be a major facilitator of mutation testing in its own right, but also has the potential to reduce the mutation equivalence problem since in many cases distinguishing data may actually exist, even if it is hard to find manually.

To test *Simulink* models, one can systematically introduce faults into the model and check how many of them can be 'discovered' by the test-set. The more the faults that can be 'discovered', the better will be the test-set. Therefore, *Mutation Coverage* can be defined as causing each conjectured fault to be detected by at least one test. This work is only concerned with *strong* mutation, which requires faults to be detected by the final output, but we see no reason why *firm* and *weak* mutation cannot be interpreted and used for Simulink mutation.

There are many ways in which one can perturb the functionality of *Simulink* models. For example, one can change the function a block carries out (e.g., change a `Sum` block to a `Product` block); one can twist the inputs of blocks (e.g., swap the first and the third input of a `Switch` block); one can alter parameters of block settings (e.g., a wrong 'Gain' parameter is assigned to a `Gain` block); and so on.

In our prototype framework, errors are introduced into the system by perturbing the values of signals carried on lines. For example, Fig. 2 illustrates an original model and its mutant model, which is created by inserting a mutation block 'AddMut'[1] into the line connecting block 'Sum' and 'Product1' in the model. Such perturbation is used because it is easy to implement, and it can be used to model many kinds of faults, such as initialisation faults, assign-

ment faults, condition check faults and even function/sub-system faults (which comprise faults injected by perturbing the functionality of blocks/sub-models).

Three types of *Mutation Operators* are defined: *Add*, *Multiply* and *Assign*. These represent adding, multiplying, or assigning the signal carried on the input by/with a certain value. In each perturbation, a parameter is associated with the mutation operator, which defines the 'certain value'. In this work, it is called the '*Mutation Parameter*'. An automatic mutant generation program has been implemented. Given the mutation operator, the mutation parameter and the fault-injection position (which defines a signal connecting two ports between two blocks), a mutant can be automatically generated by the program. By enumerating all signals within a model and applying appropriate[2] perturbation methods to them, faults are systematically introduced into models under test. For example, for the original model illustrated in Fig. 2, if one applies all different combinations of all three types of *mutation operators* and three *mutation parameters* $(1, 0, -1)$, ruling out inappropriate and obviously ineffective perturbations, 175 mutants can be generated by the automatic fault-injecting software that has been developed. A preliminary experiment has been conducted on 5 model cases (these models are also used in case study in Section 5) to compare the effectiveness of the mutation coverage specified here with that of the structural coverage specified in the Section 2.1.2. The experiment checks the structural coverage achieved by a minimised mutation-adequate test-set and the mutation coverage achieved by a minimised structural-adequate test-set. The result is: the former one achieves full coverage while the latter one does not. Figures indicating the coverage percentage for the latter case can be found in Table 4 in Section 5 too.

To measure if a fault can be detected, tests are run against both the original model and the mutant model. The fault can be detected if different outcomes are produced. If not, one may want to generate more effective

---

[1] The block 'AddMut' changes the value of its input by adding a certain value to it.

[2] The purpose of perturbing signals is to imitate errors that might occur within system implementation. Different types of signals should be perturbed by different means. A Boolean signal needs no more than 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-data that can cause the outcomes to differ. Section 5 will be devoted to investigating the technology for such test-data generation.

2.2. Search-based software testing

Search-based software testing, especially search-based test-data generation has received the most attention from researchers in the search-based software engineering domain. In addition, search-based techniques have also been used in regression test-set selection.

2.2.1. Test-data generation

Test-data generation can be dynamic or static, depending on whether the test object is executed or not. The static approach determines conditions (in the case of path coverage, the conditions would be path-traversal conditions) that need to be satisfied for the underlying test-data generation aim to be met, and then uses various means (e.g., linear programming, constraint solving) to derive desired test data from them. The constraints are typically generated using symbolic execution (Clarke, 1976). The technique, however, has not seen widespread application due to technical difficulties in handling particular language features, such as loops, arrays, pointers and memory allocation. The dynamic approach requires the execution of the software under test to determine the desired test data. Search-based test-data generation is a dynamic technique.

In search-based test-data generation approaches the satisfaction of a particular test requirement is couched as a sequence of one or more predicates over the behaviour 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 (Zhan and Clark, 2004).

One can determine how 'close' particular inputs come to satisfying these constraints (represented by predicates). Those inputs that are 'far from satisfying' the constraints may be associated with high costs, those that 'nearly satisfy' them can be associated with low costs and those that satisfy them can be associated with zero cost. Thus, the test-data generation problem is couched as a numerical minimisation problem. The idea here is that the input space can be viewed as the domain of a cost function that defines how far any input is from satisfying the constraints (test requirement). By searching the input space using optimisation techniques, the approach aims to 'home in' on test data that solve the problem at hand.

Crucial to the optimisation approach is the derivation of an appropriate cost function. A typical scheme is illustrated in Table 1 (mostly taken from Tracey (2000)). 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. The table includes enhancements (the evaluation scheme for predicate $E_1 \vee E_2$) by Bottaci (2003).

Using the cost encoding scheme illustrated above, the costs of satisfying various relevant predicates can be com-

Table 1
Commonly used cost encoding scheme

Predicate to satisfy	Value of cost function *Cost*
Boolean	if TRUE then 0, else K
$E_1 < E_2$	if $E_1 - E_2 < 0$ then 0, else $E_1 - E_2 + K$
$E_1 \leqslant E_2$	if $E_1 - E_2 \leqslant 0$ then 0, else $E_1 - E_2 + K$
$E_1 > E_2$	if $E_2 - E_1 < 0$ then 0, else $E_2 - E_1 + K$
$E_1 \geqslant E_2$	if $E_2 - E_1 \leqslant 0$ then 0, else $E_2 - E_1 + K$
$E_1 = E_2$	if $Abs(E_1 - E_2) = 0$ then 0, else $Abs(E_1 - E_2) + K$
$E_1 \neq E_2$	if $Abs(E_1 - E_2) \neq 0$ then 0, else K
$E_1 \vee E_2$ (E_1 unsatisfied, E_2 unsatisfied)	$(cost(E_1) \times cost(E_2))/(cost(E_1) + cost(E_2))$
$E_1 \vee E_2$ (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

bined to provide an overall cost for a particular execution. The aim is to reduce the overall cost to zero. The test-data generation problem becomes a cost function minimisation problem, for which a host of optimisation techniques have been adopted, e.g., simulated annealing (Tracey et al., 1998), genetic algorithms (Jones et al., 1996), tabu search (Díaz et al., 2003), and ant colony optimisation (McMinn and Holcombe, 2003). Details of heuristic search techniques can be found in Reeves (1993).

The search-based approach has been widely applied in structural testing (Jones et al., 1996; Korel, 1990; Tracey et al., 1998; Wegener et al., 2002) as well as functional testing (Tracey et al., 1998; Buehler and Wegener, 2003) and non-functional testing (which has largely focused on temporal testing) (Wegener et al., 1996; Puschner and Nossal, 1998). The above works are all carried out at the code level. Various problems encountered have been addressed, such as the flag problem (Bottaci, 2002; Harman et al., 2004) and the state problem (McMinn, 2005). Jones et al. (1995) have attempted test-data generation from Z specifications. A full account of test-data generation by heuristic search can be found in McMinn's extensive survey (McMinn, 2004).

2.2.2. Test-set selection

Test-set selection/reduction is usually used in regression testing, with the purpose of validating modified software to detect whether new errors have been introduced into previously tested code. The techniques attempt to reduce costs by selecting and running only a sub-set of tests in the test object's existing test suite. Test-set selection techniques

mainly fall into two categories: techniques for determining the error-detection ability and techniques for optimising test-sets (i.e., minimising the cost). The first category has received the most attention because techniques for it vary with programming languages, test requirements, etc. For example, with the prevalent use of Java language, corresponding regression testing test-selection techniques are emerging (Harrold et al., 2001). Techniques for the second category make use of search-based optimisation techniques, such as genetic algorithms and simulated annealing (Baradhi and Mansour, 1997; Mansour and Fakih, 1997). Again crucial to the optimisation approach is the derivation of an appropriate cost function. The construction of the cost function requires coverage ability information on the constituent tests to be available.

2.3. Random testing

Random testing has two main advantages compared to systematic testing: (1) test-data are easy to generate and (2) it allows statistical prediction of significance of testing (Hamlet, 1994). As far as the second advantage is concerned, however significant the random testing is reported as, the assurance can never be 100%. It is still desired to obtain confidence through some well-defined measurements, which lead to the attempt of satisfaction of various coverage criteria.

Once there is a goal of test-data generation, the first advantageous property of random testing – 'easy to generate' – can be exploited. Studies (Duran and Ntafos, 1984) indicated that random testing is more cost effective for many programs. Some very subtle errors can be discovered with low cost. And it is also not inferior in coverage than other carefully designed testing techniques. The efficiency problem of random test-sets can be addressed by test-set reduction. Evidence has also shown that test-set extraction has little or no reduction in the error-detection ability of the test-sets (Wong et al., 1995).

On the other hand, random testing has been reported as an ineffective approach for many testing problems, e.g., testing interrupt-driven embedded software (Regehr, 2005), preamble generation for state transition based testing (Colin and Legeard, 2004), etc.

Effectively combining random testing with other testing techniques may yield more powerful and cost-effective testing strategies.

3. Framework overview

We propose a testing framework that can facilitate the automatic generation of effective and efficient test-sets. The framework structure is illustrated in Fig. 3.

The framework provides four main functions: random test-data generation, coverage measurement, targeted test-data generation and test-set reduction. The intended test-set generation process of using the framework is described
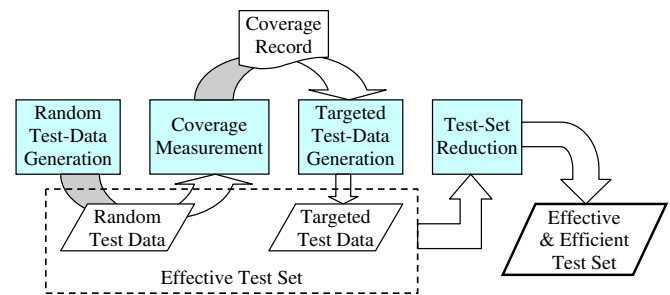

Fig. 3. Structure of the automatic test-set generation framework.

as follows. Given a model under test and a set of coverage requirements, the framework uses random test-data generation to generate a set of test data. The test data are executed and the coverage achieved by the set is recorded. Those requirements that have not been satisfied can be identified and be targeted in turn by the search-based test-data generation approach. These two sets of test data are combined to form an effective test-set. However, there is usually redundancy involved. A test-set reduction tool is used to produce an effective and efficient test-set.

The technical difficulty of the framework lies in the targeted test-data generation and test-set reduction.[3] The rest of the paper will provide solutions for addressing these problems. Two types of targeted test-data generation problems are considered, generating structural-coverage (in particular, branch coverage) test data and generating mutant-killing (in particular, mutants obtained by signal perturbation) test data. The problem caused by persistent state within the test-data generation is also addressed.

4. Structural testing

As discussed in Section 2.2.1, crucial to the search is the guidance, i.e., cost function construction. To construct a cost function, one needs to determine the strategy for evaluating test data. This involves two tasks: deriving the constraints that need to be satisfied to achieve the underlying test coverage requirement, and encoding the constraints as a cost function that gives appropriate evaluation to test data. The first task is what we need to handle here since existing technology (as illustrated in Table 1) can be adopted directly to implement the second task.

Once the cost function is built, heuristic search techniques are applied to locate the desired test data.

So in this section, we will provide our solutions to constraint derivation and the heuristic search application in the *Simulink*-based test-data generation context. An experimental study will be carried out to test the effectiveness of our techniques.

---

[3] It is not difficult to remove some redundancy from a random test-set. However, to decide the minimum test-set is a combinatorial difficult problem.

4.1. Constraints derivation

A single *Simulink branch-coverage* requirement requires the selected conditional behaviour of the underlying block to be exhibited as the output of the block. Three of the blocks that have conditional behaviours are addressed for the test-data generation process in this work. They are: `LogicalOperator`, `RelationalOperator` and `Switch`. Extensions to address the other blocks can be made accordingly by identifying the corresponding conditions for executing the different behaviours of the blocks. All three types of blocks under consideration have two conditional behaviours. For `LogicalOperator` and `RelationalOperator`, they are: being evaluated to 'TRUE' or 'FALSE'. For the `Switch` block, they are selecting the first input to pass through or selecting the third input to pass through. For simplicity, the two conditional behaviours are noted as 'TRUE' branch and 'FALSE' branch respectively for all three types of blocks. For `Switch` blocks, if the first input is selected, the 'TRUE' branch is considered to be executed; otherwise, the 'FALSE' branch is deemed to have been chosen. How each of the three blocks is handled for constraint derivation will be discussed below.

4.1.1. 'RelationalOperator' block

The branching requirement of a `RelationalOperator` block can be fulfilled by satisfying or dissatisfying the relational predicate proposed by the block. For example, in Fig. 1, if the 'TRUE' branch of block 'RO>=' is required to be covered, the relational predicate of $Out_{product} \geq Out_{product1}$ needs to be satisfied. The cost function can therefore be derived according to rules in Table 1. To obtain the runtime value of $Out_{product}$ and $Out_{product1}$, probes need to be inserted into the model at the corresponding points as illustrated in Fig. 4 ('Probe1' and 'Probe2'). Probes are inserted by connecting the signal to an `Out` block for observation.

4.1.2. 'LogicalOperator' block

In the prototype tool that has been implemented, three types of logical operators are addressed for `LogicalOperator` blocks: 'AND', 'OR' and 'NOT'. Extension to cover other types of logical operators can be implemented accordingly. To fulfill a branching requirement for a `LogicalOperator` block, it would seem natural to derive the satisfiability constraint based on the logical computation the block executes. For example, in Fig. 1 if the 'TRUE' branch of block 'LogicalAND' is required to be covered, the logical predicate of $Out_{RO\sim=} \wedge Out_{RO>=}$ needs to be satisfied. The costs of $Out_{RO\sim=}$ and $Out_{RO>=}$ need to be evaluated as individual predicates according to the cost evaluation methods in Table 1. Since they are Boolean values, according to the first rule in Table 1, the cost of a Boolean value will be evaluated to '0' or 'K'. Such cost evaluation does not have gradients (the cost function landscape will be two plateaus) and therefore cannot provide effective guidance to the search.
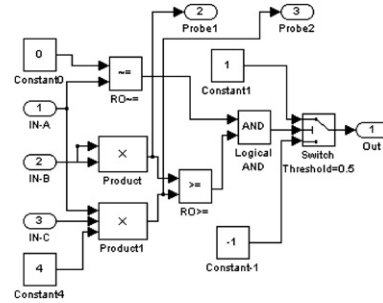

Fig. 4. Probe insertion for covering a branch of block 'RO>='.

In order to obtain a discriminating cost function, the Boolean signals involved in the target predicate need to be back-propagated within the model, so that equivalent relational predicates representing the Boolean values can be derived. In this way, the above branch coverage problem can be interpreted as satisfying the following predicate:

$$(V_{probe1} \neq V_{probe2}) \wedge (V_{probe3} \geq V_{probe4})$$

V_{probei} represents the signal value at the respective probe insertion point i, as illustrated in Fig. 5. Such constraints enable more navigable cost landscapes to be produced according to Table 1.

In some cases, multiple back propagating processes may be required. For instance, when one or more of the inputs of the underlying `LogicalOperator` block to be covered is by itself the output of another `LogicalOperator` block, the new `LogicalOperator` block needs to be back-propagated. Certain logical/relational predicate calculations may also be required in deriving the cost function. E.g. when the 'FALSE' branch is required to be covered for the 'logicalAND' block in Fig. 1.

4.1.3. 'Switch' block

To achieve branch coverage of a `Switch` block, one must execute the system with a test-datum that causes its switching condition to be 'TRUE' (i.e., $V_{SW2} \geq$ Thres), and with a test-datum that causes it to be 'FALSE' (i.e., $V_{SW2} <$ Thres). 'V_{SW2}' represents the runtime value of the second input of the `Switch` block. 'Thres' represents the 'Threshold' parameter of the `Switch` block. Usually when 'V_{SW2}' varies significantly for different inputs, the cost func-
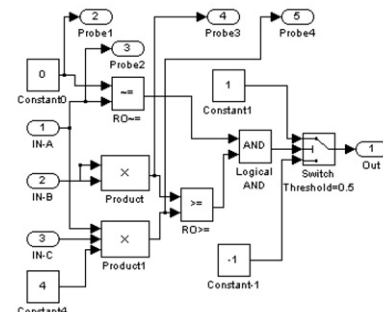

Fig. 5. Probe insertion for covering a branch of block 'LogicalAND'.

tion created would provide useful guidance to the search. However, a `Switch` block is often used as a test point for 'TRUE' or 'FALSE' (i.e., its second input is a Boolean value and Thres equals to '0.5'), like the one used in Fig. 1. In this case, a similar approach to the way `LogicalOperator` blocks are handled is more appropriate. Boolean signals involved in evaluating the cost function are traced back until a discriminating cost function can be constructed. In this manner, the following predicate can be derived for covering the 'FALSE' branch of the `Switch` block in Fig. 1:

$$(V_{probe1} = V_{probe2}) \vee (V_{probe3} < V_{probe4})$$

Here, the same probe deployment as in Fig. 5 is applied.

4.2. Search application

In this work, the well established technique of simulated annealing (SA) (Reeves, 1993) search is used. Simulated annealing is a global optimisation heuristic that is based on the *local* descent search strategy. Some researchers may consider it as a *local* search technique that allows escape from local optima. The annealing algorithm for test-data generation can be found in our previous publication (Zhan and Clark, 2004).

Simulated annealing is sensitive to its parameter settings, such as neighbourhood definition (move strategy) and cooling schedule (including four integral elements: initial temperature, final temperature, temperature decrement rate and length of Markov Chains). Different problems have different preferences. Based on information from the literature and preliminary experimentation, we decided to use the following set of parameters within this paper:

Neighbourhood definition: a *fixed-move-strategy*[4] with a parameter of `0.02`;
Initial temperature: a temperature enabling an initial move *acceptance ratio* of `70%`;
Final temperature: the temperature reached after executing 300 temperature reductions;
Geometric temperature decrease rate: `0.8`;
Markov Chains length: `100`. (The number of trial moves at a particular temperature.)

These parameters may be thought to be on the 'small' side, but the computational expense of simulation requires us to make pragmatic choices. (*The MathWorks* provides a *Simulink Accelerator* tool, which can accelerate the model simulation through two approaches: to optimise the customer model designs, and to generate equivalent code from the model, compile and execute the code. When such a tool is used, a different set of parameters may turn out to be more suitable.)

Hill climbing and genetic algorithms (GA) have also been widely used in automatic test-data generation. As a local search technique, hill climbing was not selected due to its native local search feature. It is desirable that the search is not limited to providing only local optima. GA is a population-based approach. According to preliminary experiments on simulation costs, GA is not very suitable for *Simulink* based test-data generation due to the prohibitive simulation cost involved in each test-data evaluation.

4.3. Experimental study 1

An experimental study is performed to examine the search-based test-data generation technique for *Simulink* explained above. Eight test objects are used. Of them, the first three have been designed by the author to present significant difficulty to the test-data generation technique. The next three are borrowed from McMinn's PhD thesis (McMinn, 2005), where they are used as experimental cases for test-data generation through evolutionary search. The remaining two are industrial, from an aero engine controller project.

Experiments in this paper were performed on a number of different computers. Machines of different speeds have been normalised to the speed of the fastest machine, which is an [Intel Pentium M 1.6 GHz, 512M RAM] laptop computer. Basic random test-data generation and simulation was used to gauge the normalisation factors in each case.

The simulated-annealing search-based approach as well as two other test-data generation approaches (random generation and *Reactis Tester*) are examined for comparison. They are referred to as SA-STD (it represents SA standard, as opposed to the SA-T&D approach used later in Section 6), RAND, and Reactis.

The experiment consists of two parts. The first part examines the coverage achievement of each approach. The goal is to achieve full *branch coverage*. Coverage for branches is attempted one at a time. For the SA-STD approach, in each attempt of covering one branch, according to the SA cooling schedule, the search will claim failure after `30,100` (of which `100` is used in determining the initial temperature) tries. For RAND and Reactis, each approach is allowed `100,000` attempts (evaluations of test-data) to achieve an individual branch coverage goal.[5] It is expected to show that the search-based approach can achieve higher coverage with fewer attempts. The second part of the experiment compares the time cost and the number-of-evaluations of different approaches in

---

[4] In this work, neighbours are generated by randomly choosing an input vector in the current test-datum, then randomly choosing an element of that vector and changing that element's value by a randomly chosen amount less than an identified maximum value. A *fixed-move-strategy* means that the 'identified maximum value' is fixed during the search. The parameter associated with the move strategy represents the ratio of the 'identified maximum value' to the range of the selected element. Detailed description about neighbourhood definition can be found in Zhan's thesis (Zhan, 2005).

[5] Since Reactis attempts to cover all branches at a time (no individual branch can be targeted), it is allowed (number_of_branches× 100,000) attempts each time to cover a whole model.

Table 2
Comparison of structural coverage achievements

Model name	No. of branches	–	Coverage		
			RAND (%)	Reactis (%)	SA-STD (%)
Tiny	8	Average	75	75	100
		Maximum	75	75	100
Quadratic	6	Average	83.3	83.3	96.7
		Maximum	83.3	83.3	100
Inputs-Check	8	Average	75	100	87.5
		Maximum	75	100	100
Post-Code-V.	76	Average	84.2	84.2	91.7
		Maximum	84.2	84.2	96.1
Sort-Code-V.	56	Average	76.4	75	86.8
		Maximum	78.6	75	91.1
Smoke-Detector	34	Average	88.2	88.2	90
		Maximum	88.2	88.2	94.1
Sys-Fuel-Dip-Ing-Req	20	Average	65	65	93
		Maximum	65	65	95
Calc-Start-Progress	50	Average	92.6	98	99.8
		Maximum	98	98	100

generating an input sequence to cover an individual branch using the above setting. Since *Reactis Tester* cannot be set to target one individual branch at a time, it is not included in this part of experiment.

In the experiment, each search was repeated 10 times. When RAND and SA-STD were used, it was ensured that the fixed sequence lengths for generating the appropriate test-data were long enough for all branches to be covered. *Reactis Tester* can automatically adjust the sequence length during the search.

Table 2 shows the results of the first part of experiment. It gives the average success-rate of the 10 runs and the maximum coverage achieved within the 10 runs. The results are as expected: SA-STD achieved higher coverage even with fewer attempts allowed.

Table 3 details the results of the second part of the experiment. For each model, a specific difficult branch coverage requirement was subjected to experiment. In the table, an estimate of random test-data generation probability is provided for each of the problems. Some of the runs may be unsuccessful, given the limit of tries being imposed. Therefore, it is reasonable to assume that the real cost is larger than that obtained in this experiment (hence the prefix of '>'). As can be seen, the SA-STD approach achieved a higher success rate as well as using fewer test-data evaluations (consuming less time).

4.4. Summary

This section showed how search-based test-data generation techniques can be applied to testing *Simulink* models. A method for adapting the cost function construction method to structural test-data generation for *Simulink* was proposed. The application of the simulated-annealing search technique was discussed. An experimental study was reported which demonstrated that the search-based test-data generation techniques can be successfully applied

to *Simulink* model testing. It has been shown that the search-based approach outperforms two other approaches – random testing and *Reactis Tester* (which uses constraint solving) on several difficult cases.

5. Mutation testing

In this section we will introduce a search-based technique for automatically determining data to kill the generated mutants (mutants are generated in the manner as described in Section 2.1.3). The implementation serves as a 'proof-of-concept'. Inherently difficult generic mutation issues, such as the detection of equivalent mutants, are beyond the scope of this work.

The application of the search technique should remain the same as for structural testing. Therefore, this section will focus on describing the cost-function construction method. A case study report follows.

5.1. Cost function construction

In mutation testing, one seeks to 'kill' the generated mutants; i.e., reveal the faults hidden in the mutants. The aim is to find a test-datum that causes different values to be exhibited on at least one element of the outputs between the original and mutant systems. There may be multiple paths that an error can propagate along, depending on the structure of the model. An error may or may not propagate on those paths depending on the given input and the block. An example of error propagation paths is given below.

In Fig. 6, assume that the constant value in the block Constant is erroneously set. There are six paths the error can propagate through to the output. They are: path 1, Sum – Product – Switch (inport2) – Out1; path 2, Sum – Product – Switch (inport3) – Out1; path 3, Sum – Switch1 – Out2; path 4, Sum – Product1 – Switch1

Table 3
Comparison of costs in generating individual branch-coverage test data

| Problem no.	Model name	Random generation probability	Test method	Success rate (%)	Search cost	
					Av Td evals	Av time (s)
1	Tiny	<2.64e−3	RAND	0	>100,000	>753
			SA-STD	100	1545	14
2	Quadratic	1.4142e−6	RAND	0	>100,000	>548
			SA-STD	80	>17,924	>126
3	Inputs-Check	≈1.98e−13	RAND	0	>100,000	>2457
			SA-STD	50	>24,616	>789
4	Post -Code-V.	≈1.887e−7	RAND	0	>100,000	>12,101
			SA-STD	100	10,605	1386
5	Sort -Code-V.	≈3.99e−16	RAND	0	>100,000	>6347
			SA-STD	100	6650	712
6	Smoke-Detector	1.07e−7	RAND	0	>100,000	>2017
			SA-STD	30	>25,709	>603
7	Sys-Fuel-Dip-Ing-Req	–	RAND	0	>100,000	>2714
			SA-STD	80	>13,213	>287
8	Calc-Start-Progress	5.44e−8	RAND	0	>100,000	>1404
			SA-STD	90	>14,825	>185
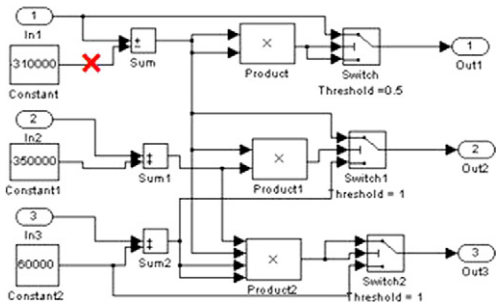
Fig. 6. Simulink model of 'Quadratic'.

– Out2; path 5, Sum – Product2 – Switch2 (inport1) – Out3 and path 6, Sum – Product2 – Switch2 (inport2) – Out3. On each of the paths, the error may be masked by any of the blocks it travels through. For example, the error may be masked by Switch1 in the third path if in both models (the original and the mutant) the outputs of Product1 are evaluated to be less than the 'Threshold' of Switch1 (which is '1'); the error may also be masked by Product1 in the fourth path if the output of Sum1 is evaluated as '0'.

The evaluation of how well an underlying test-datum satisfies a mutation testing requirement (i.e., being able to reveal the fault) should be based on how far the fault propagates within the system under test on any of the path/paths between the fault's introduction and the output. Any further propagation in any path should be rewarded. Any full propagation of a path would be considered as a successful solution. Similar to the cost function construction principles used in the last section, a large cost is assigned to test-data that are poor in propagating the fault, a small cost is assigned to test-data that are good in propagating the fault but fail in fully propagating the fault, and zero is assigned to test-data that can reveal the fault in the output.

To detect how far a fault has propagated, runtime values of some internal variables within both models need to be obtained for comparison. So the dynamic test-data generation process involves the execution of both models. Probes are inserted in both models.

To meet the goal of having different outputs between the two models, the choice of the input needs to ensure that two things happen:

1. The signal values at (after) the point where fault is injected are different.
2. The difference ripples to the outputs.

One way to design the cost function is to add the costs of satisfying the above two requirements together:

$$C = C_{\text{Diff}} + C_{\text{Ripple}} \tag{1}$$

The first requirement is normally easily achieved. Usually, unless an equivalent mutant is created (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 signal the value of 1. Usually a slight tune of the input vector can change the condition and cause the signal values at the fault-injection point to differ from those specific values. The cost of satisfying this requirement can be defined as

$$C_{\text{Diff}} = \begin{cases} K, & \text{not satisfied} \\ 0, & \text{satisfied} \end{cases} \tag{2}$$

To detect whether this requirement is satisfied or not, a probe needs to be inserted *at* the fault-injection point for the original model and *after* the fault-injection point for the mutant model. Runtime signal values can then be

monitored to decide whether they are identical and the result can be used for evaluating C_{Diff}.

To fulfill the second requirement the structure of the models between the fault-injection point and the output needs to be traversed and checked against the error propagation condition. As shown previously, there may be a number of paths from the fault-injection point to any of the output ports of the system; it is required that at least one of them can propagate the difference (show the error). On each path, every block the erroneous signal passes through has the potential of masking the fault and therefore it makes sense to break the fault propagation requirement down into a number of integral requirements. The cost of propagating the fault can then be defined as

$$C_{\text{Ripple}} = Cost(Rpath_1 \vee Rpath_2 \vee \cdots \vee Rpath_m) \quad (3)$$

where

$$Cost(Rpath_i) = Cost(Rblock_{i,1} \wedge Rblock_{i,2} \wedge \cdots \wedge Rblock_{i,n})$$
$$= \sum_{block=1}^{n} Cost(Rblock_{i,block}) \quad (4)$$

In the above equations, function '*Cost*' is defined by the same cost function encoding method as given in Table 1. '$Rpath_i$' represents the requirement of rippling the error through the ith path, assuming there are altogether m paths that the error may propagate through. '$Rblock_{i,j}$' represents the requirement of rippling the particular error[6] through the jth block on the ith path, assuming there are altogether n blocks between the fault-injection point and the output on the path.

Now the problem is reduced to evaluating each '$Cost(Rblock_{i,j})$'.

Without understanding the detailed functionality of the block, the best evaluation of the cost of enabling it to ripple an error would be[7]:

$$Cost(Rblock_{i,j}) = \begin{cases} K, & \text{not rippled} \\ 0, & \text{rippled} \end{cases} \quad (5)$$

In *Simulink*, most functional blocks produce different outputs when the input changes (e.g., mathematical blocks). Usually, such blocks only mask errors on a very specific set of input values. Once the block masks the error, a small adjustment to the input vector can enable the propagation of the error. The above cost function evaluation can be applied to such blocks.

There are some blocks that may often mask errors, such as the branching blocks. For example, for an 'AND' `LogicalOperator`, if one of its inputs is erroneous and the other input is 'FALSE', the error would be masked as the output will always be 'FALSE'. Likewise, for a `Switch`

block, if the first input is erroneous and the second input enables the third input to be output in both models, the error would also be masked. Sometimes, however, the measures necessary to avoid errors being masked by these blocks can be logically deduced. Such deduced predicates can then be encoded into cost functions for evaluating costs of propagating errors through these blocks (i.e., evaluating '$Cost(Rblock_{i,j})$'). This kind of cost function can provide more effective guidance to the test-data search. Detailed cost function construction for these blocks will be discussed below. Again, the three typical branching blocks – `LogicalOperator`, `RelationalOperator` and `Switch` – are concerned.

5.1.1. '*LogicalOperator*' block

To enable an error to propagate through a `LogicalOperator` block, it is desirable that the output of the block takes different values between the original model and the mutant model. That is to satisfy the following predicate:

$$((ValO_{OUT} == 1) \wedge (ValM_{OUT} == 0))$$
$$\vee ((ValO_{OUT} == 0) \wedge (ValM_{OUT} == 1))$$

In the above predicate, '*Val*' stands for 'Value', '*O*' stands for the original model, '*M*' stands for the mutant model and '*OUT*' stands for the output of the block, which is also where the probe is inserted (as illustrated in Fig. 7, in the figure, 'P' represents both the output of the `LogicalOperator` block and the probe).

The cost function can be therefore defined as

$$Cost(Rblock_{LOP}) = Cost(((ValO_{OUT} == 1)$$
$$\wedge (ValM_{OUT} == 0))$$
$$\vee ((ValO_{OUT} == 0)$$
$$\wedge (ValM_{OUT} == 1)))$$

Some refinement to the above cost function (to make it more effective) can be made if particular information can be deduced. For example, if the logical operator is 'AND' and if it can be deduced that all the other inputs of the block are correct (except the erroneous one), in order to enable the error to propagate, all inputs but the erroneous one should be 'TRUE'. As illustrated in Fig. 8, the probes ('P1' and 'P2') should be inserted into all input signals except the erroneous one. The cost function will be:

$$Cost(Rblock_{AND}) = Cost((Val_{P1} == 1)$$
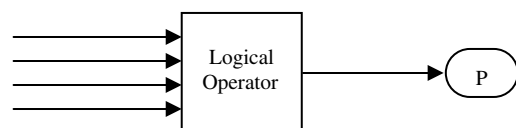$$\wedge (Val_{P2} == 1) \wedge \cdots)$$

---

[6] There may be more than one erroneous input for the underlying block. The particular error refers to the particular erroneous signal on the underling path.

[7] Similar to the way C_{Diff} is evaluated, one or more probes are inserted at the output(s) of the block to monitor if the error has been rippled.
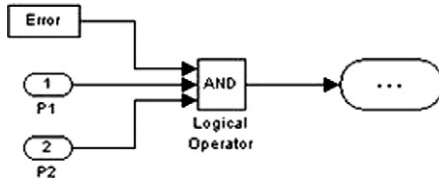

Fig. 7. Error propagation through a `LogicalOperator` block.

Fig. 8. Error propagation through a logical 'AND' block.

Fig. 9. Error propagation through a relational operator '>=' block.

In the equation, '...' represents cases that the `Logical-Operator` block has more inputs.

If the logical operator is 'NOT', the error can always remain after passing through the block. Therefore the cost will be zero.

5.1.2. 'RelationalOperator' block

If the relational operator is '>=', to propagate an error requires the following to be satisfied (no matter which input is the erroneous input):

$$((ValO_{P1} >= ValO_{P2}) \wedge (ValM_{P1} < ValM_{P2}))$$
$$\vee ((ValO_{P1} < ValO_{P2}) \wedge (ValM_{P1} >= ValM_{P1}))$$

where 'Pi' represents the ith probe. Probe insertion is illustrated in Fig. 9.

The cost function for enabling an error to propagate through a '$>=$' block is defined as

$$Cost(Rblock_{>=}) = Cost(((ValO_{P1} >= ValO_{P2})$$
$$\wedge (ValM_{P1} < ValM_{P2}))$$
$$\vee ((ValO_{P1} < ValO_{P2})$$
$$\wedge (ValM_{P1} >= ValM_{P1})))$$

If it can be deduced that for the mutant model, one of the inputs of the `RelationalOperator` block behaves erroneously and the other one behaves correctly, and if the value superiority[8] can be also deduced, then the cost function can be simplified. For example, if it can be known that $ValO_{P1} > ValM_{P1}$ and $ValO_{P2} == ValM_{P2}$, the cost function for the '>=' `RelationalOperator` block can be simplified as

$$Cost(Rblock_{>=}) = Cost((ValO_{P1} >= ValO_{P2})$$
$$\wedge (ValM_{P1} < ValM_{P2}))$$

Such simplification can effectively speed up the search for targeted test-data.

The cost function definition for the other `RelationalOperator` blocks can be similarly derived.

5.1.3. 'Switch' block

If the first input of a `Switch` block is erroneous, to enable the error to propagate through it is required that the second input of the block in both models enables the
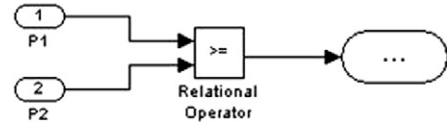
first input to channel through. The cost function can be defined as

$$Cost(Rblock_{Switch}) = Cost((ValO_{P1} >= Thres_{Switch})$$
$$\wedge (ValM_{P1} >= Thres_{Switch}))$$

'$Thres_{Switch}$' represents the 'Threshold' parameter of the `Switch` block. The probe insertion method is illustrated in Fig. 10.

If it can be deduced that the second input of the block cannot be erroneous, which indicates that $ValO_{P1} == ValM_{P1}$, the cost function can be simplified as

$$Cost(Rblock_{Switch}) = Cost(ValO_{P1} >= Thres_{Switch}) \tag{6}$$

If the third input of a `Switch` is erroneous, the cost function can be constructed accordingly.

If the second input of a `Switch` block is erroneous, the following cost function can be constructed (The probe insertion method is illustrated in Fig. 11.):

$$Cost(Rblock_{Switch}) = Cost(((ValO_{P2} >= Thres_{Switch})$$
$$\wedge (ValM_{P2} < Thres_{Switch})$$
$$\wedge (ValO_{P1} \sim= ValM_{P3}))$$
$$\vee ((ValO_{P2} < Thres_{Switch})$$
$$\wedge (ValM_{P2} >= Thres_{Switch})$$
$$\wedge (ValO_{P3} \sim= ValM_{P1}))) \tag{7}$$

Simplification can also be applied. For example, if it is deduced that $ValO_{P2} > ValM_{P2}$, the cost function can be reduced to:

$$Cost(Rblock_{Switch}) = Cost((ValO_{P2} >= Thres_{Switch})$$
$$\wedge (ValM_{P2} < Thres_{Switch})$$
$$\wedge (ValO_{P1} \sim= ValM_{P3}))$$

5.1.4. A cost function evaluation example

The case to be considered is the problem given in Fig. 2. Eq. (1) will be used as the cost function: $C = C_{Diff} + C_{Ripple}$.

To evaluate C_{Diff}, a probe (P1) is inserted to monitor whether the runtime values at the error injection point are identical between the two models. Eq. (2) will be used for the cost computation.
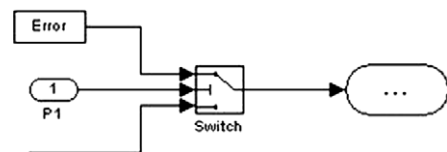

Fig. 10. Error propagation through a `Switch` block with the first input being erroneous.

---

[8] The value superiority tells which value is larger. For example, if the mutation method is 'increase the value by 1', then the value in the mutant should be inferred to be larger than that in the original.
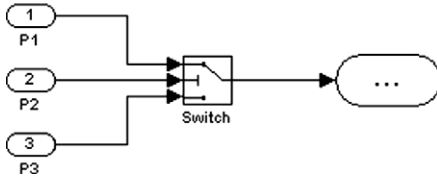
Fig. 11. Error propagation through a `Switch` block with the second input being erroneous.

C_{Ripple} can be evaluated by Eq. (3): $C_{\mathrm{Ripple}} = Cost(Rpath_1 \vee Rpath_2 \vee \cdots \vee Rpath_m)$.

There is only one path that the error may propagate through, which is `Product1 – Switch1 – Switch4 – Out1`.

Therefore $C_{\mathrm{Ripple}} = Cost(Rpath_1)$, where $Cost(Rpath_1) = \sum_{block=1}^{n} Cost(Rblock_{1,block})$, according to Eq. (4).

In this case, $n = 3$, block(1) = `Product1`, block(2) = `Switch1`, block(3) = `Switch4`.[9]

To evaluate $Cost(Rblock_{1,1})$, a probe (P2) needs to be inserted into the output of the block. Eq. (5) will be used for the cost computation.

To evaluate $Cost(Rblock_{1,2})$, probes need to be inserted into all three inputs of the block. Since a probe (P2) already exists for the second input, P3 and P4 are inserted for the first and third inputs. Eq. (7) will be used for the cost computation.

To evaluate $Cost(Rblock_{1,3})$, a probe (P5) needs to be inserted into the second input of the block. Since it can be deduced that the second input of the block cannot be erroneous, Eq. (6) will be used for the cost computation.

Probe insertions are illustrated in Fig. 12.

5.2. Experimental study 2

An experimental study is performed to examine the search-based mutant-killing test-data generation technique for *Simulink*. Five models are used. The first two are designed by the author and the other three are from industry, an aero engine controller project. Three of the models have been used in experimental study 1.

The same experimental setting as for experimental study 1 is used here. Mutants are generated using the three types of signal *mutation operators* ('*Add*', '*Multiply*' and '*Assign*') as defined in Section 2.1.3. The *Mutation parameters* applied are '1', '0' and '-1'. Inappropriate perturbations are ruled out.[10]

Two types of comparisons are carried out, the overall mutation coverage (i.e., mutation score) and the performance of generating individual mutant-killing test-datum. Since there is no existing tool for this kind of test-data generation for *Simulink*, the only approach that can be compared to is the random generation approach.
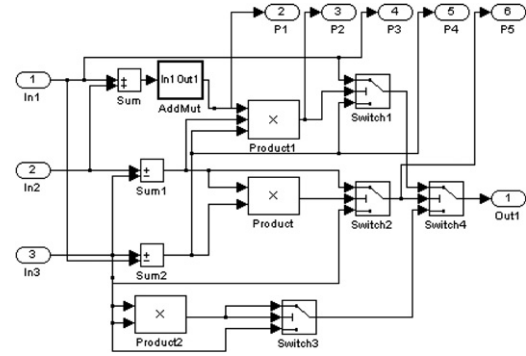

Fig. 12. Probe insertions for a cost evaluation example.

In the first comparison, the three types of test-sets being compared are: firstly, randomly generated sets of `100,000` test-data; secondly, those same test-sets with the addition of supplementary test-data that enables full structural coverage to be achieved (in this experiment, it is branch coverage); and thirdly, randomly generated test-sets supplemented as above now including mutation-adequate test-data generated by the technique addressed in this section. Mutation scores achieved by the three types of test-sets are compared. It is expected to show that structural-adequate test-sets may not be mutation adequate, and that the mutation-based test-data generation technique is effective in providing the desired test-data.

The second comparison compares the solving success rate and computational cost of the two approaches (random approach and search-based approach) on some individual mutant-killing test-data generation problems. The cases selected are mostly those mutants that cannot be killed by `100,000` random test-data. It is expected to demonstrate that the search-based approach outperforms the random approach in such difficult test-data generation cases, i.e., costs less and succeeds more often.

The results of the first comparison are presented in Table 4. Full structural coverage of model 'Detect-Duplex-Faults' and 'Cockpit-Speed-Trimming' is easily achieved by random testing. Therefore, no supplemental structural test-data are added to the second sets and hence there is no improvement in the mutation coverage achieved by the second sets. As can be seen, full structural coverage generally does not achieve full mutation coverage. The random test-sets by themselves have even lower mutation coverage. Full mutation coverage can be achieved using the mutation-adequate test-data generation technique for the cases investigated.[11]

Table 5 details the results of the second part of the experiment. The eight problems represent some difficult[12]

---

[9] The `Out1` block is not considered because it will never mask the error.
[10] For example, assigning values that are not '0' or '1' to Boolean signals is considered to be inappropriate.

[11] When the search-based test-data generation failed to find data to kill particular mutants, a manual examination of the model under test was carried out. Examinations showed that such mutants were all equivalent mutants.
[12] A mutant-killing problem is considered to be difficult if random test-data generation fails to generate adequate test data for it after trying 100,000 times for at least once within 10 tries.

Table 4
Comparison of mutation score

Model name	No. of non-equivalent mutants	–	Mutation score		
			RAND (%)	RAND + Structural (%)	RAND + Structural + Mutation (%)
Tiny	118 (All Non-Equiv.)	Average	39.8	89.8	100
		Maximum	39.8	89.8	100
Quadratic	161 (All Non-Equiv.)	Average	86.3	94.4	98.1
		Maximum	86.3	94.4	100
Detect-Duplex-Faults	210 (3 Equivalent)	Average	99.9	99.9	100
		Maximum	100	100	100
Cockpit-Speed-Trimming	233 (4 Equivalent)	Average	93.9	93.9	100
		Maximum	93.9	93.9	100
Calc-Start-Progress	337 (10 Equivalent)	Average	91.1	98.5	100
		Maximum	91.1	98.5	100

Table 5
Comparison of costs in generating individual mutant-killing test data

Problem no.	Model name	Test method	Success rate (%)	Search cost	
				Av Td evals	Av time (s)
1	Tiny	RAND	0	>100,000	>1538.1
		SA-MUT	100	2382	41.2
2	Tiny	RAND	0	>100,000	>1549.6
		SA-MUT	100	9164	166.0
3	Quadratic	RAND	0	>100,000	>1193.8
		SA-MUT	20	>29,428	>432.6
4	Quadratic	RAND	0	>100,000	>1161.2
		SA-MUT	60	>21,595	>338.3
5	Detect-Duplex-Faults	RAND	90	>75,678	>1729.6
		SA-MUT	100	4328.3	99.7
6	Cockpit-Speed-Trimming	RAND	0	>100,000	>1983.2
		SA-MUT	100	627	14.3
7	Cockpit-Speed-Trimming	RAND	0	>100,000	>1996.7
		SA-MUT	100	1030.2	44.9
8	Calc-Start-Progress	RAND	0	>100,000	>2117.4
		SA-MUT	100	2178	58.5

mutant-killing test-data generation problems encountered in the first part of the experiment. The results showed that compared to the random approach, the search-based approach can frequently achieve a higher success rate as well as using fewer test-data evaluations in achieving the goal.

5.3. Summary

The execution of structural elements of a system does not guarantee the detection of errors. To gain more confidence in the software being produced, higher mutation coverage is desired in testing. This section proposed a search-based test-data generation method for generating mutation-adequate test-data for *MATLAB/Simulink* models. An experimental study was carried out. The results further proved that structural-adequate test-sets may let many kinds of faults go undetected, and also demonstrated that our mutation-adequate test-data generation technique can successfully generate desired test-data.

6. The state problem

Despite the successful use of search-based test-data generation described in the previous two sections, certain features of systems can hinder the test-data search, e.g., the flag problem (Bottaci, 2002) and the ***state problem*** (McMinn, 2005). Embedded systems, such as engine controllers, typically make extensive use of state to record real-time information. Such systems usually require test data to be sequences of inputs that can put the system into a certain state, in order to exercise particular elements. Thus, the generation of such input sequences becomes difficult. Usually a coarse objective landscape is yielded and the test-data search easily gets stuck.

The Sort-Code-Verification problem used in McMinn's thesis (McMinn, 2005) is an example. (This problem has also been used in the experimental study 1 in this paper.) The problem description is as follows:

The system validates a UK bank sort code of the form 'XX – XX – XX' where 'X' is an integer digit. A line feed charac-

ter is also expected at the end of the input. Unicode characters are submitted to the system one at a time. The system needs to keep track of how far through the validation process the system is. One of three constant integer values is returned. *RESULT_ENTER_NEW_CHAR* signals the system is ready to accept a new character; *RESULT_VALID* signals the previously entered sequence is valid and that the system is ready to read in another sort code; or *RESULT_INVALID*, which signals that the last character was invalid.

Fig. 13 is the *Simulink* description of the problem.

In the model, the states are maintained by the `Unit-Delay` block passing the information in the previous step back into the system for use by the new step (clock cycle). Assume that the test aim is to cause the output of block 'pos9+lf' to be TRUE. This requires a valid sort code to be input. To satisfy such a test aim manually, the following constraints can be derived: the targeted test-datum should consist of at least 9 steps; and the shortest sequential input should be: digit, digit, dash, digit, digit, dash, digit, digit and line feed. Standard cost function design will usually generate two constraints: the value of 'In1' is equal to 12 and the runtime value of the output of block 'pos' equals to 9. Such constraints do not tell what the minimum number of steps needed to be executed is. The cost function landscape of the second constraint will be 9 plateaus; each indicates one of the states of the system. Therefore the search will obtain little guidance from these constraints and result in failure. The kind of guidance the search really needs is something like the constraints that are derived manually.

We therefore will introduce a technique called ***tracing and deducing*** (T&D), which uses more detailed constraint information to replace the rough guidance. Such detailed constraints allow a more easily navigable landscape to be created. The rest of this section will describe the T&D

method and report its effectiveness according to experimentation.

6.1. The tracing and deducing method

The *tracing and deducing* method seeks to provide more navigable guidance to the search. Suppose that a testing goal is given. This would typically be a requirement to satisfy some predicate P over one or more signal values at a certain step in the execution of the system, e.g., a branch condition. As 'proof-of-concept' work, we restrict ourselves to the branch-coverage test-data generation domain. Then this predicate P must be a single predicate over a single signal. In other words, for a branching block of type `LogicalOperator` or `RelationalOperator`, P would specify the expected output value of the underlying block, to be either '0' or '1'; for a branching block of type `Switch`, P would specify the expected second-input value of the block to be either '⩾' or '<' the 'Threshold' of the `Switch` block. In the above interpretation, this single signal can always be identified as the output of some block B. By back-propagating the signal through block B, the predicate can be replaced by a new predicate or a conjunction or disjunction of multiple predicates. The new predicate(s) is (are) over the input signal(s) of block B. We call a replacement like this one 'application' of the deducing process. The newly generated predicates can be deduced in the same way. By tracing back recursively like this, more useful guidance to the targeted test-data search can be derived. The call of the deducing process is controlled by the tracing procedure, which determines when to stop. A simplification process is also used with the *tracing and deducing* process in order to keep the complexity of predicates under control. Details about applying the T&D method and some relevant issues will be discussed below.
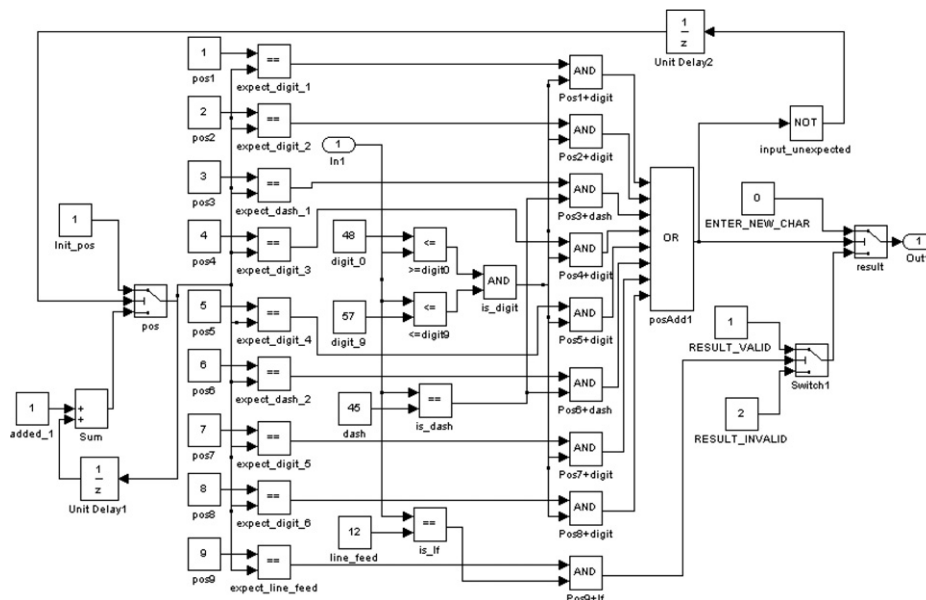

Fig. 13. Simulink model of 'SortCodeVerification'.

6.1.1. Number of steps to execute

For a system that maintains states, there is no straightforward way to know the minimum number of steps necessary to execute to enable certain coverage or certain behaviour. Usually automatic test-data generation tools (e.g., our two tools described in Section 4 and 5 in this paper and the tools by McMinn (2005)) do not provide facilities for identifying such number; rather, given an arbitrary number of steps to run (usually by the tool user), the tool either produce satisfactory test-data or fail (the failure reason – due to too small number of steps used or due to the search – remains unknown). When using the T&D method, given the number of steps to run, the solver will assume that the test goal is met on the final step (such decision gives the highest opportunity for the T&D process to succeed). Sometimes the simplification procedure in the T&D method can determine contradiction, indicating that the number of steps to execute is too small. In this case, an augmented number of steps should be tried. On the contrary, if the number is too big, the T&D method will result in providing unnecessarily complicated constraints for guiding the search (which may also fail the search).

6.1.2. Notions and information storage rules

A branch-coverage testing goal can be interpreted into a predicate defining the value range of a particular signal value at the final step of the execution. For example, if the goal is to have the output of block 'pos3+dash' to be 'TRUE', the goal predicate will be: the output value of block 'pos3+dash' on the Nth step is 'TRUE' (N is the number of steps to execute). Starting from this initial predicate (constraint), new constraint(s) can be deduced. All constraints should concern values of signals at a certain time unit. Each signal has a unique integer label (determined by its source block name and source port number). The following notation is used in denoting constraints. The value of the 5th step of signal 32 will be denoted as '$P_{32}(5)$'. 'P' represents term 'Probe', which indicates that we have to insert a probe into that signal to detect its value. In a similar way, a constant value of 58 will be denoted as '$C(58)$', where 'C' represents 'Constant'.

Constraints are always in the form of a relational predicate or various logical combinations of relational predicates, such as

$$P_{32}(5) >= C(58);$$
$$\{P_{32}(5) >= C(58) \lor P_{32}(5) <= C(47)\} \land P_{32}(5) \neq C(12).$$

The constraints are recorded in a tree-like structure, which is called an ***objective-tree***. There are two types of nodes in an *objective-tree*: ***Predicate-Node*** and ***Or-Node***. A *Predicate-Node* records the information of an atomic constraint, which is a relational predicate, and its 'next' domain, which is a pointer pointing to the node that has a conjunctive 'AND' relation with it. An *Or-Node* can have multiple children; each is denoted by a pointer to the corresponding *Child-Node*, which will, in turn, be a *Predicate-Node* or an *Or-Node*. The relation between the children of an *Or-Node* is disjunctive 'OR'. An *Or-Node* also has a 'next' domain, pointing to the node that has an 'AND' relation with all its children. If the 'next' domain is '0', it means no more predicates will be included in this conjunctive relation. An *Or-Node* is denoted by a triangle, with arrows coming out of its left corner representing the child pointers and one arrow originating from the middle of its bottom side representing the next pointer; a *Predicate-Node* is denoted by a rectangle with one arrow originating from the middle of its bottom side representing the next pointer. For both types of nodes, the next pointer may or may not exist.

For example, predicate $\{P_{32}(5) >= C(58) \lor P_{32}(5) <= C(47)\} \land P_{32}(5) \neq C(12)$ can be denoted as shown in Fig. 14. In the figure, the numerals, such as 1–4, are node numbers. Node 1 is an *Or-Node*, having two children – node 3 and node 4. The 'next' domain of node 1 points to node 2.

6.1.3. Deducing rules

The deducing activity is the process of refining predicates. If a predicate is deduced, a new predicate or a few new predicates will be generated to replace the original one. Some deduction rules for a selected range of blocks have been implemented into a prototype tool. However, the deduction rules may not be restricted to these. More rules for other types of blocks can be added to the set.

In the prototype tool, for a predicate of form '$P_x(y)$ *rel* $C(z)$', it can be deduced when the source block of probe 'x' is: Switch, or LogicalOperator, or RelationalOperator, or UnitDelay, or Sum and it has only one non-constant input, or Product and it has only one non-constant input. Also, only predicates with one variable(probe) are deduced. Detailed deducing rules are given below.

6.1.3.1. 'Switch' block.
If the source block is Switch, as illustrated in Fig. 15:

There are two ways the predicate '$P_x(y)$ *rel* $C(z)$' can be satisfied: the second input of Switch is greater than or equal to the 'Threshold' parameter and the first input satisfies the predicate requirement as x does; or, the second input of Switch is less than the 'Threshold' parameter and the third input satisfies the predicate requirement as x does. The step number of the signals in the newly deduced
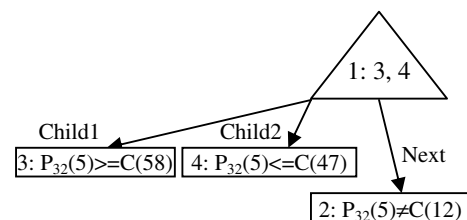

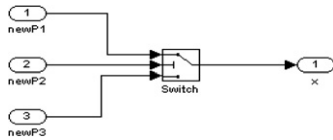Fig. 14. An example of objective-tree representation.

Fig. 15. Deducing process for Switch block.

constraints should be the same as in the original predicate. The implementation is as described below.

Four new nodes are created: newNode1, newNode2, newNode3 and newNode4. The 'Threshold' ('thres') of the Switch block will be detected.

NewNode1 will be recorded as predicate '$P_{newP2}(y) >= C(thres)$'. Its 'next' domain will be 'newNode3'.

NewNode2 will be recorded as predicate '$P_{newP2}(y) < C(thres)$'. Its 'next' domain will be 'newNode4'.

NewNode3 will be recorded as predicate '$P_{newP1}(y)$ rel C(z)'. Its 'next' domain will be '0'.

NewNode4 will be recorded as predicate '$P_{newP3}(y)$ rel C(z)'. Its 'next' domain will be '0'.

The tree structure transformation is illustrated in Fig. 16.

6.1.3.2. 'LogicalOperator' block.
If the source block of signal x is LogicalOperator or RelationalOperator, since the output of such blocks must be either '0' or '1', the original predicate node '$P_x(y)$ rel C(z)' needs to be interpreted. For example, if rel is '>' and z is '0.5', the predicate will be interpreted into '$P_x(y) == C(1)$'; if rel is '<' and z is '0.3', the predicate will be interpreted into '$P_x(y) == C(0)$'; if rel is '<' and z is '0', the predicate will be deduced to 'FALSE' since that is impossible; if rel is '≠' and z is '5', the predicate will be deduced to 'TRUE' since it is always true. A value of '0' or '1' of $P_x(y)$ is called the **target result** of such source blocks.

If the logical operator is 'AND' and if the *target result* is '1', in order to satisfy this predicate, all inputs of the block need to be TRUE. If the *target result* is '0', in order to sat-

isfy this predicate, at least one of the inputs of the block needs to be FALSE.

If the logical operator is 'OR' and if the *target result* is '1', in order to satisfy this predicate, at least one of the inputs of the block needs to be TRUE. If the *target result* is '0', in order to satisfy this predicate all inputs of the block need to be FALSE.

If the logical operator is 'NOT', in order to satisfy the predicate, the value of the input signal of the block should be exactly the opposite of the *target result*.

Detailed implementation will be omitted here.

6.1.3.3. 'RelationalOperator' block.
If the source block is RelationalOperator, as explained in the last section (Section 6.1.3.2), the original *Predicate-Node* '$P_x(y)$ rel C(z)' can be interpreted into one of the following two forms: '$P_x(y) == C(1)$' or '$P_x(y) == C(0)$'. To satisfy such a predicate, the inputs of the block should satisfy the relation defined by the operator in the first case, or fail to do so in the second case.

6.1.3.4. 'UnitDelay' block.
If the source block is UnitDelay, as illustrated in Fig. 17, according to the functionality of the UnitDelay block (the output of it equals the value of its input signal in the previous step), the input signal of the UnitDelay block in the previous step should satisfy exactly the requirement for the output signal of the UnitDelay block defined by the original predicate. The implementation can be defined accordingly:

If the step number y equals to '1', the original *Predicate-Node* '$P_x(y)$ rel C(z)' will be deduced to 'TRUE' or 'FALSE' accordingly. This is because the output value of a UnitDelay block is always '0' for the first step. Otherwise,
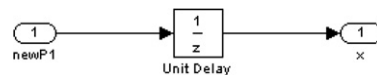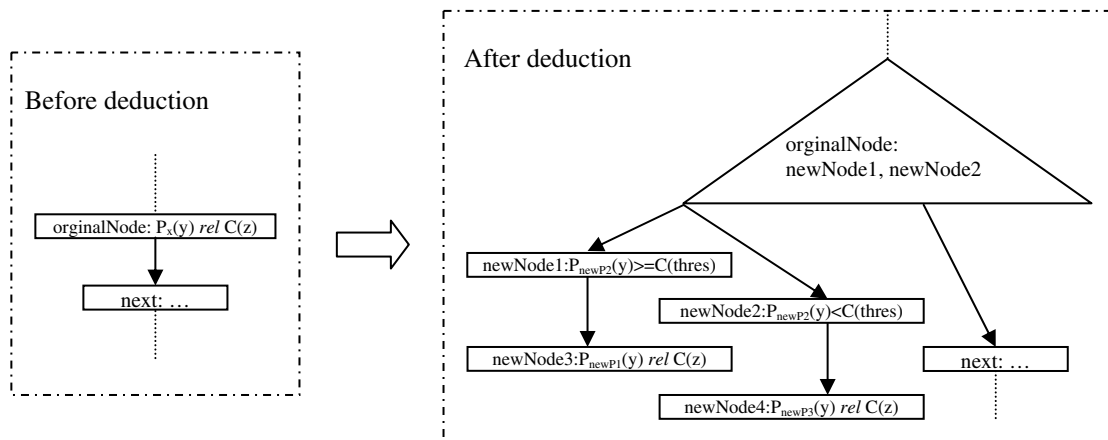

Fig. 17. Deductive process for UnitDelay block.

Fig. 16. Node structure before and after deducing a Switch block.

the original *Predicate-Node* will be changed into 'P_{newP1}`(y-1)` *rel* `C(z)`'. Its 'next' domain remains the same.

6.1.3.5. 'Sum' or 'Product' block. If the source block is `Sum` or `Product` and it has only one non-constant input, the predicate can be deduced into a constraint about the non-constant signal. For example, for the situation illustrated in Fig. 18, if the constant value is A, the original *Predicate-Node* 'P_x`(y)` *rel* `C(z)`' will be changed into 'P_{newP1}`(y)` *rel* `C(z-A)`'. Its 'next' domain remains the same.

6.1.4. Tracing stopping rules

The tracing process is a recursive process. In every pass, it checks each available node in turn; if deducible, the deducing function is called upon. Simplification (introduced in the next section, Section 6.1.5) to the *objective-tree* occurs at the end of each pass.

A node will not be further traced and deduced if both of its relational operands are probes.

Since the output of a `UnitDelay` block is always '0' for the first step, a node cannot be further traced when its step number is '1' and the probe's source block is `UnitDelay`.

To avoid the typical problem caused by symbolic execution (i.e., *objective-tree* explosion), a maximum tree-size is imposed in the tracing process. The process is stopped when the number of nodes reaches this limit. This does not prevent us from using heuristic search to find the ultimate test-data, but the guidance provided to the search may be less informative.

6.1.5. Simplifications

During the *tracing and deducing* process the *objective-tree* gets bigger. The tree is frequently simplified. There are a few simplification treatments defined in the prototyping tool. For example, if a node is determined to be constantly `TRUE`, the node will be deleted; if a node is determined to be constantly `FALSE`, the branch the node belongs to will be removed. Simplification methods are also defined for cases such as: an *Or-Node* has only one child branch left or no child branch at all (such a situation results from previous node-TRUE or node-FALSE simplification), two nodes on one path are conflicting, or two nodes on one path are consistent (i.e., the satisfaction of one can assure the satisfaction of the other). Details of the implementation of the simplification rules are omitted here. Interested readers can refer to Zhan's thesis (Zhan, 2005).
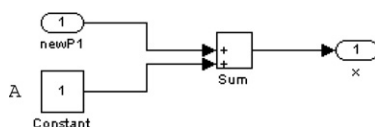

Fig. 18. Deductive process for Sum block.

If a tree is simplified to be empty, it indicates that the test-data generation is infeasible and the number of steps to execute must be augmented.

The current simplification tool is only a proof-of-concept implementation. Complicated conflicting constraints cannot be identified nor thereafter be simplified. For example, constraint $(A <= 0.3 \lor B >= 3) \land (A > 0.3 \land B == 2)$ cannot be identified as `FALSE`. The usefulness of the T&D approach may be enhanced by the incorporation of more powerful constraint solving tools.

6.1.6. Conclusion

The deducing process is indeed a reverse symbolic execution process. The whole technique can be considered as a partial symbolic execution. It is partial because it does not fully execute the system. It executes only to the extent that the symbolic conditions can be handled without any difficulty. This is controlled by the tracing stopping rules. The technique allows some of the benefits of symbolic execution to be obtained within the context of a heuristic search approach, but without suffering its state explosion problems.

6.2. Experimental study 3

An experiment was run with a preliminary implementation of the *T&D* technique. *T&D* was implemented as an additional component that can be inserted into our previously implemented standard search-based test-data generation tool for *Simulink*.

In this experiment, five models are tested. They are selected from the ones used in experimental study 1 that have persistent state. Four test-data generation approaches are compared. They are: random testing, standard SA search-based testing, *tracing and deducing* technique facilitated SA search-based testing and *Reactis Tester*. They are named RAND, SA-STD, SA-T&D and Reactis respectively. The aim of this experiment is to compare the performances of various test generation approaches. The same experimental setting as experimental study 1 is used.

Again, the experiment is made up of two parts. The first part examines the total branch coverage achievements of each approach. Table 6 shows the results. The second part of the experiment compares the costs of different approaches in generating an input sequence to cover an individual branch. For the same reason as in experimental study 1, *Reactis Tester* is not included. Table 7 shows the results. Taking both tables into account, the SA-T&D approach demonstrates superiority in covering models by and large.

In Table 7, the ratio of time cost for objective-tree construction (incurred by the T&D algorithm) is also given. As can be seen, for the fourth problem, the majority of the cost was spent in the tree construction rather than SA search. Such high cost is due to the limitation of the prototype implementation of the simplification tool.

The T&D approach failed to achieve full coverage for two models in the experiments. For Smoke-Detector, the

Table 6
Comparison of structural coverage achievements

Model name	No. of branches	–	RAND (%)	Reactis (%)	SA-STD (%)	SA-T&D (%)
Inputs-Check	8	Succ-Rate	75	100	87.5	100
		Coverage	75	100	100	100
Post-Code-V	76	Succ-Rate	84.2	84.2	91.7	100
		Coverage	84.2	84.2	96.1	100
Sort-Code-V	56	Succ-Rate	76.4	75	86.8	100
		Coverage	78.6	75	91.1	100
Smoke-Detector	34	Succ-Rate	88.2	88.2	90	94.1
		Coverage	88.2	88.2	94.1	94.1
Sys-Fuel-Dip-Ing-Req	20	Succ-Rate	65	65	93	82
		Coverage	65	65	95	95

Table 7
Comparison of costs in generating individual branch-coverage test data

Problem no.	Model name	Random generation probability	Test method	Success rate (%)	Search cost		
					Av Td evals	Av time (s)	Tree construction time (%)
1	Inputs-Check	≈1.98e−13	RAND	0	>100,000	>2457	–
			SA-STD	50	>24,616	>789	–
			SA-T&D	100	2404	54	3.1
2	Post-Code-V	≈1.887e−7	RAND	0	>100,000	>12,101	–
			SA-STD	100	10,605	1386	–
			SA-T&D	100	1446	133	58.6
3	Sort-Code-V	≈3.99e−16	RAND	0	>100,000	>6347	–
			SA-STD	100	6650	712	–
			SA-T&D	100	2270	141	30.5
4	Smoke-Detector	1.07e−7	RAND	0	>100,000	>2017	–
			SA-STD	30	>25,709	>603	–
			SA-T&D	100	3273	839	88.2
5	Sys-Fuel-Dip-Ing-Req	–	RAND	0	>100,000	>2714	–
			SA-STD	80	>13,213	>287	–
			SA-T&D	60	>14,656	>323	<1.2

failure was also due to the immature constraint simplification technique used in the prototype tool. Advanced constraint solving tool should solve the problem. For Sys-Fuel-Dip-Ign-Req, the inability to achieve full coverage was partly due to infeasibility of one branch. In comparison to the SA-STD approach, the SA-T&D approach has a lower success rate in covering Sys-Fuel-Dip-Ign-Req. Detailed information shows that the difference lies in the different solving success rates of 5 of the branches. For this particular problem, SA-STD was allowed 3 steps for each test-datum evaluation whilst the minimum requirement is 2 steps (which is the sequence length used by SA-T&D). Therefore, in each test-datum evaluation, SA-STD actually had two opportunities (goal achieved in the 2nd or 3rd step) to win the assessment while SA-T&D had only one (goal achieved in the 2nd step). This is also the reason SA-T&D costs more than SA-STD in the fifth problem of Table 7.

6.3. Summary

The presence of persistent state causes ineffective cost functions to be generated. As a result, the search may fail due to the insufficient guidance. In order to address this problem, a technique called '*tracing and deducing*' is proposed, which helps provide a more navigable cost function. The experimental results have shown that the *tracing and deducing* technique can substantially enhance the capability of search-based test-data generation for *Simulink*. It improves both the time to produce test data and the coverage.

7. Test-set reduction

The task of generating predicted test results is expensive because it is generally carried out manually. Hence it is desirable to use as few test cases as possible in the test-execution process. In this work, since random testing is used as part of the test-set generation approach we proposed, the incorporation of an effective test-set reduction process is therefore necessary.

Test-set reduction techniques have usually been considered as part of the regression testing techniques, where the main purpose is to reduce retesting in the regression testing stage. The approach used was to identify the elements of the test object that may affect or be affected by the modified

part of the code (i.e., to identify elements to be covered) and then select the minimum number of tests from the original/used test-set that maximises the coverage of these elements. However, it is believed that this kind of test-set selection technique can be used at any stage of the software development and testing cycle, rather than just regression testing.

In this section, we will provide a generic way to formulate the test-set reduction problem and a search-based technique to solve the problem. The purpose of this work is to integrate it into our *Simulink* test-set generation framework. Therefore, experiments will be carried out in the context of *Simulink* model testing in order to show how the test-set reduction process fits in the effective and efficient test-set generation framework. However, the test-set reduction techniques provided here should be universally applicable for any programming/modelling environment and any testing purpose. All the reduction program needs to know is which test aims are met by which test data. The particular means of generating test data is irrelevant.

7.1. Search-based test-set reduction

Given a set of test data, the approach to test-set reduction is to: identify a desired set of test aims,[13] detect which test data can satisfy which test aims (through test execution or simulation), then apply search techniques to extract an optimised sub-set. Such technology can be applied at any testing stage in order to reduce the size of test-sets. In particular, it can be used to improve the efficiency of random test-sets, which, although cheap to create, require excessive time and labour to generate predicted test results for.

The problem of extracting a sub-set from a large test-set with the aim of minimising the sub-set size whilst maintaining its test coverage is an optimisation problem, which is combinatorially hard. Usually an optimisation-based heuristic search technique, such as simulated annealing or genetic algorithm, is used to solve such a problem (Baradhi and Mansour, 1997; Mansour and Fakih, 1997). In this work, we choose to use the simple simulated-annealing algorithm. So the implementation of a test-set selection process typically involves three activities. They are: (1) Coverage 'map'[14] construction; (2) Neighbourhood definition and cost-function design; (3) Heuristic search. These activities will be described individually below.

7.1.1. Coverage 'Map' construction

To provide evaluations for selected sub-sets, a test coverage 'map' needs to be generated. The 'map' can be obtained by executing or simulating the system under test

and monitoring which test aims are met by which test data. To obtain the coverage map, the system under test sometimes needs to be instrumented. For example, in the case of structural coverage observation, probes need to be inserted to detect whether certain structural elements are executed or not. On the other hand, in the case of strong mutation coverage observation, only the outcomes are required to be monitored and therefore the system can be executed as it is. However a mutant system needs to be generated.

Given a set of test aims (`N`) and a set of test data (`M`), a 'map' (matrix) of test-set coverage can be constructed in the following format as illustrated in Table 8. In the table, '`1`' indicates that the test-datum of the row *satisfies* the test aim of the corresponding column and '`0`' indicates that the test-datum of the row *does not satisfy* the corresponding test aim of the column.

7.1.2. Neighbourhood definition and cost-function design

The most basic test-set reduction problem would be: to minimise the test-set whilst maintaining its coverage. A derivative of the problem can be: given the number of tests that one can afford to execute, find a sub-set of tests that can maximise the test coverage. In this section we discuss how to address these two forms of problem. The derivative problem is simpler and therefore will be attempted first.

7.1.2.1. Fixed test-set size, maximise the coverage. Let a_{ij} ($1 \leqslant i \leqslant$ `M`, $1 \leqslant j \leqslant$ `N`) be an element of the test coverage 'map'. Let X_i ($1 \leqslant i \leqslant$ `M`) denote the inclusion (`=1`) or exclusion (`=0`) of the *i*th test-datum. The test-set reduction problem can be interpreted as maximising the following objective function:

$$f = \sum_{j=1}^{N} positive \left(\sum_{i=1}^{M} (a_{ij} \times X_i) \right)$$

with the constraint of

$$\sum_{i=1}^{M} X_i = K$$

where K ($0 < K <$ `M`) is the number of tests that are allowed to remain in the resulting set.

In the objective function formula, the function *positive* returns `1` when the input is a positive value, `0` otherwise (i.e., here, it returns `1` if the selected test-datum *i* meets the test aim *j*).

---

[13] A desired set of test aims can be generalised as 'test coverage requirements'. Each test coverage requirement can be refined as a set of individual test aims. For example, 'branch-coverage requirement' implies a number of individual branch-coverage aims.

[14] A coverage 'map' is a matrix recording the test coverage capability of each test-datum against each individual test aim.

Table 8
Coverage 'map'

	TestAim$_1$	TestAim$_2$	$\cdots$	TestAim$_n$
TestDatum$_1$	1	0	$\cdots$	1
TestDatum$_2$	0	0	$\cdots$	1
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
TestDatum$_m$	0	1	$\cdots$	1

The solution to the problem can be naturally encoded as a vector X ($X_i \in \{0, 1\}$, $1 \leqslant i \leqslant M$). To satisfy the constraint of only selecting K test data, in the initial solution K elements of the vector are randomly set to '1' and all the others are set to '0'. E.g. the solution to a problem of reducing a test-set of size 10 to a test-set of size 3 can be represented as a vector $[0, 0, 1, 0, 0, 0, 0, 1, 1, 0]$, which means the third, eighth and ninth test-datum of the original set are selected for the extracted sub-set. A neighbourhood solution can be generated by flipping two elements of the solution vector that hold opposite values, i.e., flipping a '1' into '0' and flipping a '0' into '1'. This maintains the 'selected test-set size equals to K' constraint.

The constraint can be removed by reformatting the solution into a set with K elements, each of which is a *distinct* integer with a range of $[1 .. M]$, representing the members of the original test-set. Such a solution can be represented as an array Y of length K, where Y_i is the ith element of array Y, $Y_i \in [1 .. M]$. For example, for a test-set of size 10, a candidate extracted set of size 3 can be represented as an array Y' of length 3, (i.e., $[Y'_1, Y'_2, Y'_3]$) of which each element is an integer with a range of $[1 .. 10]$. Array $[3,8,9]$ represents the sub-set containing the third, eighth and ninth test-datum of the original set.

The objective function is therefore converted into:

$$f = \sum_{j=1}^{N} positive\left(\sum_{i=1}^{K} a_{Y_i j} \right)$$

It can also be converted into an objective function for minimisation:

$$f = \sum_{j=1}^{N} positive\left(\sum_{i=1}^{M} a_{ij} \right) - \sum_{j=1}^{N} positive\left(\sum_{i=1}^{K} a_{Y_i j} \right) \qquad (8)$$

In the above function, the first part represents the total number of test aims that can be met by the original test-set; and the second part represents the total number of test aims that can be met by the sub-set selected. This objective function can be directly used as the cost function for the search.

A neighbourhood solution can be defined as: swapping one of the test data in the solution set with one of the test data in the non-solution set.[15] In the previous example, a neighbourhood solution of $[3,8,9]$ can be $[3,6,9]$.

If Eq. (8) is used, the search terminates when the cost function is minimised to zero or the limit of the search procedure is reached. The solution will be the best (i.e., which yields the smallest cost) configuration tried during the run.

7.1.2.2. Minimise test-set size, maintain coverage. There are two ways to solve this problem.

7.1.2.2.1. Solution 1. Applying the same notation as used in Section 7.1.2.1, the test-set minimisation problem can be interpreted as minimising the following objective function (it is also used as cost function for the search):

$$f = \sum_{i=1}^{M} X_i$$

with the constraint of

$$\sum_{j=1}^{N} positive\left(\sum_{i=1}^{M} (a_{ij} \times X_i) \right) = \sum_{j=1}^{N} positive\left(\sum_{i=1}^{M} a_{ij} \right)$$

In the constraint, the left part represents the total number of test aims that can be covered by the sub-set selected; and the right part represents the total number of test aims that can be covered by the original test-set.

To encode the constraint into the objective function (also used as cost function), the objective function will then be made up of two parts:

$$f = \sum_{i=1}^{M} X_i + W$$

$$\times \left(\sum_{j=1}^{N} positive\left(\sum_{i=1}^{M} a_{ij} \right) - \sum_{j=1}^{N} positive\left(\sum_{i=1}^{M} a_{ij} \times X_i \right) \right) \qquad (9)$$

The first part reflects the objective of minimising the test-set. The second part reflects the constraint of maintaining the coverage. The coefficient W ($W > 1$) is a 'fiddle factor' introduced to express the bias towards the constraint (i.e., the constraint must be satisfied). The goal is to minimise the second part to zero, meanwhile also minimising the first part. When $W <= 1$, the increment of the second part may be offset by the reduction of the first part (e.g., the penalty of covering one fewer testing aim may be offset by the cost reduction of losing one test-datum). This is undesired. Therefore W should be assigned a value bigger than '1'.

Naturally the solution of the test-set extraction problem takes the form of vector X ($X_i \in \{0, 1\}$, $1 \leqslant i \leqslant M$). A neighbourhood solution can be defined as flipping the value(s) of one or two of the elements within the vector. The reason for allowing the flip of two elements is that at some point of the search only this kind of neighbourhood movement (one flips from '1' to '0' and the other flips from '0' to '1') can enable a non-worsening move.[16] Such potential is required by the nature of the simulated-annealing search (which is used in this work).

According to the cost function defined in Eq. (9), the cost can never be evaluated to zero, so the search will only

---

[15] The non-solution set results from subtracting the solution set from the original set.

[16] Some test-datum can satisfy more aims than others. By flipping the more powerful candidate datum which was not selected into '1' and flipping the less powerful candidate datum which was selected into '0', the cost evaluation may be improved.

Table 9
Various test-sets coverage and reduction potential

Model name	Coverage type	No. of aims	RAND10		RAND100		RAND1000		RAND10000		RAND + SA	
			Coverage (%)	Min set size	Coverage (%)	Min set size	Coverage (%)	Min set size	Coverage (%)	Min set size	Coverage (%)	Min set size
Tiny	Structural	8	62.5	2	75	2	75	2	75	2	100	2
	Mutation	118	23.7	1	35.6	2	39.8	3	39.8	3	100	6
	Both	126	26.2	2	38.1	2	42.1	3	42.1	3	100	6
Quadratic	Structural	6	83.3	2	83.3	2	83.3	2	83.3	2	100	2
	Mutation	161	85.1	3	85.1	3	85.1	3	86.3	3	100	5
	Both	167	85.0	3	85.0	3	85.0	3	86.2	3	100	5
Calc-Start-Progress	Structural	50	50	1	50	1	52	2	86	5	100	4
	Mutation	327	15.6	1	15.9	1	16.5	2	64.3	8	100	9
	Both	377	20.2	1	20.4	1	21.2	2	74.0	8	100	9
Detect-Duplex-Faults	Structural	20	100	3	100	2	100	2	100	2	100	2
	Mutation	207	73.9	6	76.3	6	76.3	6	99	12	100	12
	Both	227	76.2	6	78.4	6	80.6	6	99	12	100	12
Cockpit-Speed-Trimming	Structural	14	100	3	100	2	100	2	100	2	100	2
	Mutation	229	67.7	4	89.5	5	89.5	5	89.5	5	100	5
	Both	243	69.5	5	90.1	5	90.1	5	90.1	5	100	5

terminate when the limit of the simulated-annealing algorithm is met.

7.1.2.2.2. Solution 2. Another solution is to convert the problem into a series of problems in the form of the one discussed in Section 7.1.2.1 – fixed test-set size, maximise the coverage capability. Since the size of the targeted smallest test-set is unknown, it needs to be acquired through trials. A binary search[17] can be used to locate this number, which has a range of [1 .. M]. The same method as given in Section 7.1.2.1 is called upon each time, the stopping criteria for each search are that a set (target set) that has the same coverage capability as the original one is found or the search procedure limit is reached. If the target set is found, the binary search goes to the smaller side of the search domain (i.e., try a smaller extracted-set size). Otherwise, the larger side of the search domain is explored (i.e., try a bigger extracted-set size). The search will be repeated $\lfloor \log_2 M \rfloor$ times.

7.1.2.2.3. Conclusion. In appearance, the second solution is nearly a logarithmic factor worse than the first solution. However, this is not true in the author's experiments. This is mainly because that the search spaces (which determines the search cost) for the two problem formulation methods are different. A theoretical reasoning can be found in Zhan's thesis (Zhan, 2005).

7.1.3. Solve the optimisation problem

In this section, the simple annealing algorithm is used with the following parameter setting. A geometric cooling rate of 0.9 is used. The number of attempted moves at each temperature is calculated according to Eq. (10), with a maximum of 100 iterations and maximum number of

30 consecutive unproductive iterations. These choices can clearly be varied for other conditions as required.

$$L = 0.2 \times Size_{neighbourhood} \qquad (10)$$

7.2. Experimental study 4

In this section, an experimental study is carried out to demonstrate the effectiveness of the test-set reduction techniques proposed in this section as well as the success of the whole test-set generation framework.

The five models used in this experimental study are those that have been used in experimental study 2. Solution 2 (in Section 7.1.2.2.2) is used in the experiment.

The aim of the experiment is to show the significant saving achieved by using the test-set reduction technique on random test-sets. In this experiment, for each model under test, 5 test-sets are generated. The first four are random sets of size 10, 100, 1000 and 10,000 respectively. The fifth is a test-set that combines the fourth random set and supplementary test data[18] generated for achieving full coverage of the test requirement. The coverage requirement includes both branch coverage (as used in experimental study 1) and mutation coverage (as used in experimental study 2). The experiment will show the coverage achievement of each test-set and the reduced test-set size after applying the test-set minimisation process on each set. Table 9 shows the results of the experiment.

As can be seen, random testing can achieve relatively high coverage for most of the cases used in the experiment. But the coverage is not full. The SA-based targeted test-data generation has to be resorted to complement the random approach. On the other hand random testing involves high redundancy. The higher coverage one intends to

---

[17] Binary search searches a sorted array by repeatedly dividing the search interval in half. Ref.: http://www.nist.gov/dads/HTML/binarySearch.html. In our case, the sorted array is made up of successive integers between 1 and M.

[18] The supplementary test-data were obtained through the SA search-based test-data generations as described in the previous sections.

achieve, the more random test cases are required and the more redundancy is incurred.

For model Calc-Start-Progress, to achieve structural coverage, the minimum set size of the RAND+SA set is smaller than that of the RAND10000 set whilst achieving higher coverage (100% vs. 86%). This is because some case(s) in the RAND+SA are very powerful and can satisfy more test requirements than that (those) in the RAND10000 set.

7.3. Summary

Random testing has high redundancy. The higher coverage one intends to achieve from random testing, the more random test cases are required and the more redundancy is incurred. To exploit the advantages of random test-sets (easy to obtain), an effective test-set reduction technique is needed. The test-set reduction techniques proposed in this section successfully addressed this problem.

8. Conclusion and future work

There has been considerable success applying heuristic search techniques for code-level test-data generation. In this paper, we have applied search techniques to test-data generation for *Simulink* models. A framework has been created that applies search-based techniques to structural coverage problems and mutation coverage problems. In addition, the state problem is addressed. Positive experimental results have been obtained to demonstrate the effectiveness of the proposed techniques.

A full test-set generation framework is proposed, which successfully combines random testing and the search-based targeted test-data generation techniques to enable effective test-sets to be generated efficiently. A generic test-set reduction problem encoding and solving technique is proposed and used in the framework to extract efficient test-sets. Experiments have shown that such a testing framework can successfully serve its role in the *Simulink*-model-based testing.

In this work, applications of optimisation techniques to structural testing and mutation testing for *Simulink* were addressed. The principles should generalise easily to address other forms of testing, e.g., modified condition/decision coverage, boundary testing, etc. However, in-depth investigation is desired.

So far as the author is aware, little work has been carried out to generate mutation-adequate test-data. The only attempt to produce mutation-adequate test-data was made by DeMillo and Offutt in DeMillo and Offutt (1991), where they used the constraint solving approach. The limited practical use of mutation testing is likely due to the lack of supporting tools for the difficult parts. The automated mutation-adequate test-data generation technique proposed here should positively improve the practicality of mutation testing. Given the power of mutation testing, we wish that more interest and work will be inspired by the progress we have made in this area.

Acknowledgements

References

Baradhi, Ghinwa, Mansour, Nashat, 1997. A comparative study of five regression testing algorithms. In: Proceedings of Australian Software Engineering Conference (ASWEC 1997), Sydney.

Beizer, B., 1990. Software Testing Techniques, second ed. International Thomson Computer Press.

Bottaci, L., 2002. Instrumenting programs with flag variables for test data search by genetic algorithm. In: Proceeding of the Genetic and Evolutionary Computation Conference (GECCO 2002), pp. 1337–1342.

Bottaci, L., 2003. Predicate expression cost functions to guide evolutionary search for test data. In: Proceeding of the Genetic and Evolutionary Computation Conference (GECCO 2003), pp. 2455–2464.

Buehler, O., Wegener, J., 2003. Evolutionary functional testing of an automated parking system. In: International Conference on Computer, Communication and Control Technologies (CCCT 2003) and the 9th International Conference on Information Systems Analysis and Synthesis (ISAS 2003).

Clarke, L., 1976. A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering 2 (3), 215–222.

Colin, Séverine, Legeard, Bruno, 2004. Fabien Peureux Preamble computation in automated test case generation using constraint logic programming. Software Testing, Verification and Reliability 14 (3), 213–235.

DeMillo, R.A., Offutt, A.J., 1991. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17 (9), 900–909.

Díaz, Eugenia, Tuya, Javier, Blanco, Raquel, 2003. Automated software testing using a metaheuristic technique based on tabu search. In: 18th IEEE International Conference on Automated Software Engineering. Montreal, Canada, October.

Duran, J.W., Ntafos, S.C., 1984. An evaluation of random testing. IEEE Transactions on Software Engineering 10 (4), 438–443.

Hamlet, D., 1994. Random testing. In: Marciniak, J.J. (Ed.), Encyclopedia of Software Engineering. John Wiley & Sons, pp. 970–978.

Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M., 2004. Testability transformation. IEEE Transactions on Software Engineering 30 (1), 3–16.

Harrold, M.J., Jones, J., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S., Gujarathi, A., 2001. Regression test selection for Java software. In: Proceedings of the ACM conference on object-oriented programming, systems, languages, and applications (OOPSLA 2001), Tampa, FL, USA, October, pp. 312–326.

Jones, B., Sthamer, H., Yang, X., Eyre, D., 1995. The automatic generation of software test data sets using adaptive search techniques. In: The 3rd International Conference on Software Quality Management, pp. 435–444.

Jones, B., Sthamer, H., Eyres, D., 1996. Automatic structural testing using genetic algorithms. Software Engineering Journal 11 (5), 299–306.

Korel, B., 1990. Automated software test data generation. IEEE Transactions on Software Engineering 16 (8), 870–879.

Mansour, N., Fakih, K., 1997. Natural optimisation algorithms for optimal regression testing. In: Proceedings of the 21st International

Computer Software and Application Conference (COMPSAC 1997), Washington, DC, August.

McMinn, Phil, 2004. Search-based software test data generation: a survey. Software Testing, Verification and Reliability 14 (2), 105–156.

McMinn, Phil., 2005. Evolutionary search for test data in the presence of state behaviour. Ph.D. thesis, University of Sheffield, January.

McMinn, P., Holcombe, M., 2003. The state problem for evolutionary testing. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO03), pp. 2488–2500.

Puschner, P., Nossal, R., 1998. Testing the results of static worst-case execution-time analysis. In: Proceedings of the 19th IEEE Real-Time Systems Symposium, pp. 134–143.

Reactive Systems Inc. <http://www.reactive-systems.com/>.

Reeves, C.R. (Ed.), 1993. Modern Heuristic Techniques for Combinatorial Problems. Blackwell, Oxford.

Regehr, John, 2005. Random testing of interrupt-driven software. In: Proceedings of the 5th ACM International Conference on Embedded Software.

The MathWorks. <http://www.mathworks.com/products/simulink>.

Tracey, N., 2000. A search-based automated test-data generation framework for safety critical software. Ph.D. thesis, University of York.

Tracey, N., Clark, J., Mander, K., McDermid J., 1998. An automated framework for structural test-data generation. In: International Conference on Automated Software Engineering, pp. 285–288.

Tracey, N., Clark, J., Mander, K., 1998. Automated program flaw finding using simulated annealing. In: Symposium on Software Testing and Analysis (ISSTA), pp. 73–81.

Voas, Jeffrey, McGraw, Gary, 1997. Software Fault Injection: Inoculating Programs against Errors. John Wiley & Sons.

Wegener, J., Grimm, K., Grochtmann, M., Sthamer, H., Jones, B., 1996. Systematic testing of real-time systems. In: Proceedings of the 4th European Conference on Software Testing, Analysis & Review (EuroSTAR 1996), December.

Wegener, J., Buhr, K., Pohlheim, H., 2002. 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), pp. 1233–1240.

Wong, W.E., Horgan, J.R., London, S., Mathur, A.P., 1995. Effect of test set minimization on fault detection effectiveness. In: Proceedings of the 17th IEEE International Conference on Software Engineering, pp. 41–50.

Zhan, Y., 2005. A search-based framework for automatic test-set generation for MATLAB/Simulink models. Ph.D. thesis, University of York. December.

Zhan, Y., Clark, J., 2004. Search-based automatic test-data generation at an architectural level. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004). LNCS, vol. 3103, pp. 1413–1426.

Zhan, Y., Clark, J., 2005. Search-based mutation testing for Simulink models. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005). ACM Press, pp. 1061–1068.

Zhan, Y., Clark, J., 2006. The state problem for test generation in Simulink. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006). ACM Press, pp. 1941–1948.

Zhu, Hong, Hall, Patrick A.V., May, John H.R., 1997. Software unit test coverage and adequacy. ACM Computing Surveys 29 (4), 366–427.