Automatic Test-Data Generation for Testing *Simulink* Models

Yuan Zhan, John Clark

Department of Computer Science

University of York

Technical Report YCS-2004-382

27 August, 2004

Abstract

Software testing is costly, labour-intensive, and time-consuming. For most practical systems it will not be possible to perform 'exhaustive testing'. Test sets must be effective (i.e. they reveal faults) but also easily generated (i.e. the process of generation must be efficient). We generally aim to develop small sets with high fault detection ability. Systematically generating effective test-data is one of the most interesting and practically relevant topics in the testing domain.

Modern testing requires faults to be discovered at the earliest possible stage, i.e. specification or architecture design stage rather than the coding stage, because the cost of fixing an error increases with the time between its introduction and detection. We need to generate test cases to exercise our high-level models. Again, as with all testing, we wish to develop effective test sets, and to do so efficiently.

One means of capturing high-level behaviour of systems is provided by the *Matlab/Simulink* toolset. *Matlab/Simulink* is popularly used in embedded systems engineering as an architectural-level design notation. Engineers like using it, finding it intuitively appealing. In this report we show how certain techniques, taken largely from automated code-level testing, can be adapted for *Matlab/Simulink* models and applied to generate test sequences that satisfy identified testing aims. In our work, these aims are mainly concerned with analogues of code-level structural and fault-based coverage criteria. We describe the functionality of a toolset developed to automatically generate effective and efficient test sets for the architectural models of interest. We describe also the techniques applied in developing the toolset. Preliminary experimental results show that the toolset can facilitate automatic test-data generation for architectural level models to a certain extent.

Table of Contents

1 Introduction

Software testing is an expensive process. It typically consumes more than 50% of the total development budget [Bei90]. Failure to detect errors can result in significant financial loss or even disaster in the case of safety critical systems. Complete testing is impossible due to the huge input spaces involved. It is desirable, therefore, to seek techniques that will achieve testing rigour (i.e. be effective) at an acceptable cost (i.e. be efficient).

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 especially serious when developers want to achieve high confidence in the correctness of their developed systems. Automated test-data generation is a way forward to solve this problem and to increase testing efficiency. Automation lies at the heart of our proposed research.

The modern aim of 'testing' is to discover faults at the earliest possible stage because the cost of fixing an error increases with the time between its introduction and detection. Thus high-level models have become the focus of much modern-day verification effort and research. The *Matlab/Simulink* notation is a widely used for the design of industrial embedded systems. It allows models to be created and executed. *Matlab/Simulink* models can be architectural level designs of software systems. The simulation facilities allow such models to be executed and observed. This property of *Simulink* turns out to be an advantage for effective dynamic testing. In this work, we focus on automatically generating effective test-data for testing *Matlab/Simulink* models.

Code level coverage criteria are explained in [ZHM97]. However, these can be adapted to specification and architecture level testing too. We construct a framework that is concerned with interpretations of widely used structural coverage criteria and also a form of mutation testing.

Before going into the details of the work, it is necessary to help the readers build up a bit of background on *Simulink*. *Simulink*[1] is a software package for modelling, simulating, and analysing system-level designs of dynamic systems. *Simulink* models/systems are made up of blocks connected by lines. Each block implements some function on its inputs and outputs the results. Outputs of blocks form inputs to other blocks (represented by lines joining the relevant input/output ports). Models can be hierarchical. Each block can be a subsystem comprising a number of other blocks and lines. Figure 1-1 shows a simple *Simulink* model.

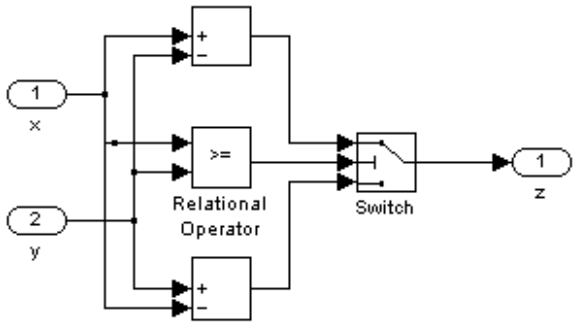

Figure 1-1 An example of a *Simulink* model.

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

2 Testing criteria

A testing *adequacy criterion* is a criterion that defines what constitutes an *adequate* test-set [ZHM97]. An adequacy criterion can be used in two ways. Firstly, it can be used as a stopping rule indicating whether more testing is needed. Secondly, it can be used to measure test quality by associating a degree of adequacy with each test set. An adequacy criterion is an essential part of any testing method. It not only directs the selection of test-data, but also decides the sufficiency of a given test set.
According to the underlying approach, testing can be categorised as structural-based testing, fault-based testing or error-based testing. Some types of adequacy criteria are more suitable than others on particular problems. Generally, different adequacy criteria are complementary and are often combined in practice. Our current work focuses on carrying out structural-based and fault-based testing, which will be detailed below.

2.1 Structural-based testing

Structural-based testing specifies testing requirements in terms of the coverage of a particular set of elements in the structure of the program or the specification [ZHM97]. It can be either control-flow oriented or data-flow oriented.
Control-flow oriented testing is based on the knowledge of the control structure of the program. A variety of coverage criteria such as *all-statements-coverage* [2], *all-branches-coverage* [3], *all-paths-coverage* [4] etc. can be defined for code level testing. For state-charts specifications, the counterpart testing criteria are *all-state-coverage* [5], and *all-transition-coverage*[6] and *all-transition-path-coverage*[7].
Data-flow oriented testing criteria are constructed so that critical associations between the definitions of a variable and its uses are examined during program testing. The basic idea behind it is that if the result of some computation has never been used, one has no reason to believe that the computation was correct.
The control-flow oriented testing criteria are fundamental and popular criteria in testing. Currently we gear our prototype test-data generation tool to carry out control-flow oriented testing.
The control-flow coverage criteria listed above, such as *all-statements-coverage*, are named and defined in the context of code. But they can be easily mapped to the environment of *Simulink*. For example, the *all-statements-coverage* can be mapped to *all-blocks-coverage* criterion, which requires the execution of every block to be reflected in the final model output (rather than being masked off by the 'Switch' branching process) at least once.
In *Simulink*, there are certain blocks that form branches. They are: 'For', 'If', 'Multiport Switch', 'Switch', 'SwitchCase' and 'While' block. 'For', 'If', 'SwitchCase' and 'While' blocks are provided by *Simulink* for the convenience of model construction from programs. But they are not generally used in constructing control system models. In particular, they are

[2] All-statements-coverage requires every statement in the program to be executed at least once.

[3] All-branch-coverage requires every decision in the program to take each possible outcome at least once.

[4] All-path-coverage requires every combination of possible outcomes of decisions in the program to be executed at least once.

[5] All-state-coverage requires every state in the state-chart to be activated at least once by a test sequence in the test suite.

[6] All-transition-coverage requires each transition in the state-chart to be caused by a test sequence in the test suite at least once.

[7] All-transition-path coverage requires every combination of transition sequences to be executed at least once.

ruled out in Rolls-Royce Controls[8]. Therefore we do not address problems concerned with these blocks in the current work. The two popularly used branching blocks are: the 'Switch' block and its derivative, the 'Multiport Switch' block. A 'Switch' block can map to an 'if … then … else' branching structure in code. For example, the model in Figure 1-1 can be mapped to the following code:

```
program calculation;
input x,y;
output z;
begin
  if x>=y
    z = x-y;
  else
    z = y-x;
end;
```

Therefore, for *Simulink* testing, we can keep the terms of *all-branches-coverage* and *all-paths-coverage* that have been used at the code level, but interpret them as meaning 'the Switch/MultiportSwitch block predicate takes each possible value at least once' and 'each possible combination of Switch/MultiportSwitch block predicate outcomes are executed at least once'.

2.2 Fault-based testing

Fault-based testing mainly focuses on measuring the quality of a test set according to its ability to detect specific faults. Mutation testing is a fault-based testing technique proposed by DeMillo et al. [DLS78].

Mutation testing works in the following way: a large number of simple faults, such as alterations to operators, constant values and variables are introduced into the program under test one at a time. The resulting programs are called *mutants*. Next, the goal is to generate test cases that can distinguish each mutant from the original program by 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. Figure 2-1 gives an example of mutant construction. For this instance, if the input variables are y and z, and the output is variable x, then input case (y=0, z=0) cannot kill either of the mutants as the output x will be the same for all three programs (one original and two mutants). However, the test input data (y=1, z=2) can distinguish both mutants from the original.

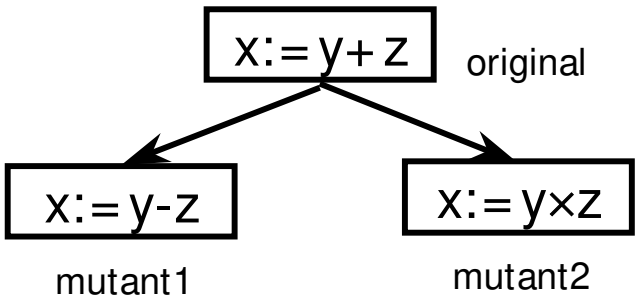

Figure 2-1 Illustration of mutants.

[8] Yuan Zhan holds a Rolls-Royce University Technology Center studentship.

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

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

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

To test *Simulink* models, we introduce errors to the system by perturbing the values of signals carried on wires/lines rather than the operation performed within blocks. For example, in the system illustrated in Figure 2-2, a mutant model can be created by inserting a mutation block 'AddMut' into one of the wires (which is the line connecting block 'Sum' and block 'Product1' in the model), as illustrated in Figure 2-3. Such perturbation can be used to model initialization faults, assignment faults, condition check faults and even function/subsystem faults.

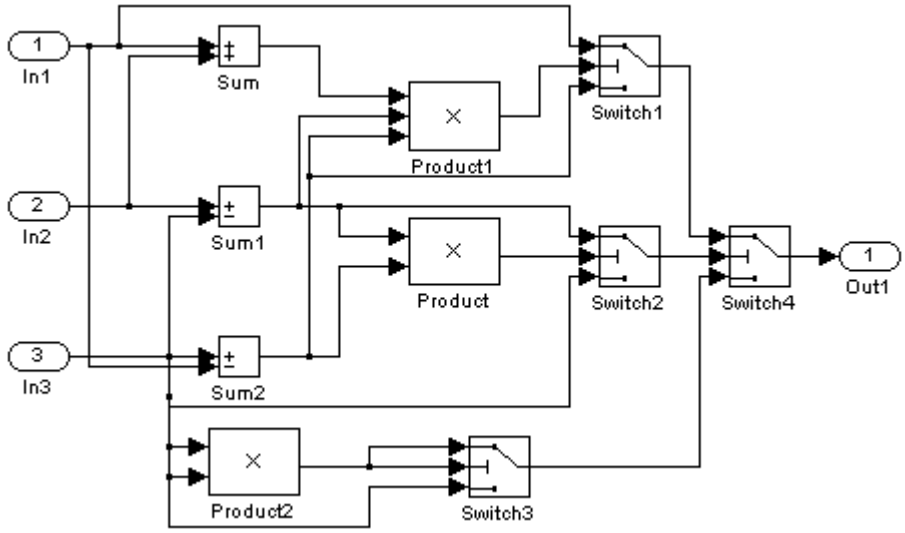

Figure 2-2 *Simulink* Original Model.

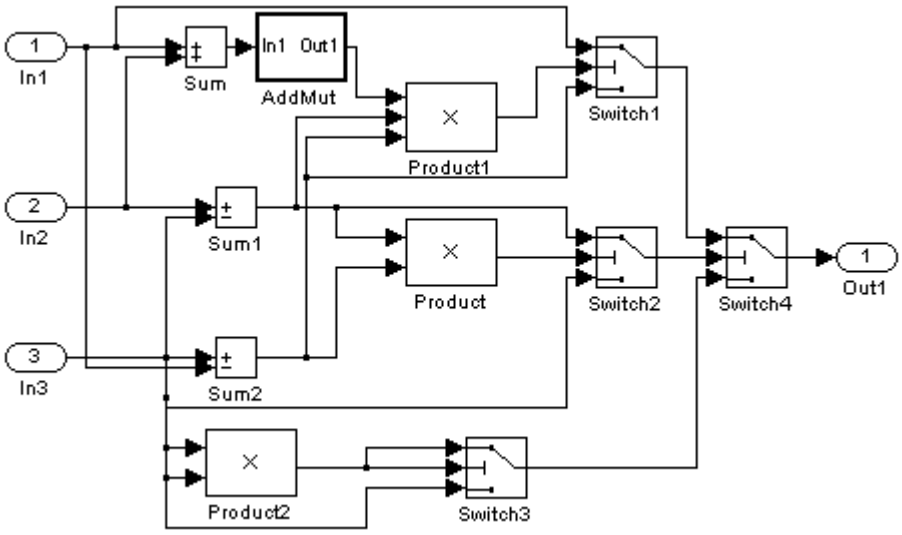

Figure 2-3 *Simulink* Mutated Model.

3 Simulation based test-data generation

Simulation based test-data generation is a kind of dynamic test-data generation, in which the dynamic simulation results are effectively used as information to direct the targeted test-data search. Depending on what property of the system we are testing, probes are inserted into the system under test for us to monitor the runtime values of some particular internal variables so that appropriate information can be obtained (e.g. at the code level, programs are instrumented to record and show some internal runtime information; at the architectural level, in particular for *Simulink* models, an 'Out' block is connected to each wire whose run-time values we wish to monitor). The internal runtime information is then used to guide the search for the desired test-data. We choose to use a global optimisation-based approach in the search for test-data. Such dynamic test-data generation technology has been applied at the code level by many researchers, Korel [Kor90], Tracey et al. [TCMM98], Wegener et al. [WBS01] etc.
Application of the approach requires you to:

1. Interpret the problem into an optimisation search problem: decide on how to evaluate the suitability of test-data in satisfying the underlying test goal and what kind of information is needed to make the evaluation, then instrument the model by inserting probes at the points we need to obtain the appropriate runtime information.
2. Define the cost function, which should assign a small cost to test-data that are 'close to' satisfying the test goal and assign a high cost to those that are 'far from' satisfying the test goal. A good cost function definition should be able to reflect the small differences in quality of test-data; a cost function that gave the same value to all data not achieving the goal could provide no guidance to the search.
3. Apply optimisation-based search to find the desired test-data. This will be accomplished dynamically. Each simulation/run of the system with a particular test-datum provides the suitability information of it and directs the move towards the goal test-datum for the next simulation and evaluation cycle.

Details of the application of these components for testing architectural level models in our work will be described in the next section.

4 Tool suite construction

4.1 Problem conversion

4.1.1 Structural-based

In the prototype tool implementation we consider only models whose branching blocks are 'Switch' blocks. Thus a requirement for generating a test input that covers a particular path comprises a subset of the 'Switch' blocks involved in the model together with their respective required condition values (satisfied or unsatisfied[9]). We can take such a requirement as the equivalent of a 'subpath' requirement in programs. A single test-data generation requirement for the model in Figure 4-1 might be: Switch2 – satisfied, Switch3 – unsatisfied. Some combinations of 'Switch' conditions will be over-restrictive, e.g. in Figure 4-1, if we require that 'Switch3' predicate is to be satisfied, which means the first (top) input of it is put through to the output, it is over-restrictive to specify whether block 'Switch2' is to be satisfied or not because the outcome of block 'Switch2' will not be reflected in the model outcome anyway. Over-restrictive combinations may also be infeasible.

[9] A 'Switch' branching condition is satisfied when the first input of the 'Switch' block is channelled through to the output and is unsatisfied when the third input of the 'Switch' block is channelled through.

Fulfilment of structural adequacy criteria will require a test-set to exercise various identified combinations of 'Switch' predicates. We may impose a simple '*all-branches-coverage*' criterion (each branch of a 'Switch' must be exercised by at least one test input vector) through to 'exhaustive coverage' of each possible combination of 'Switch' predicates (we shall term this *all-paths-coverage*).

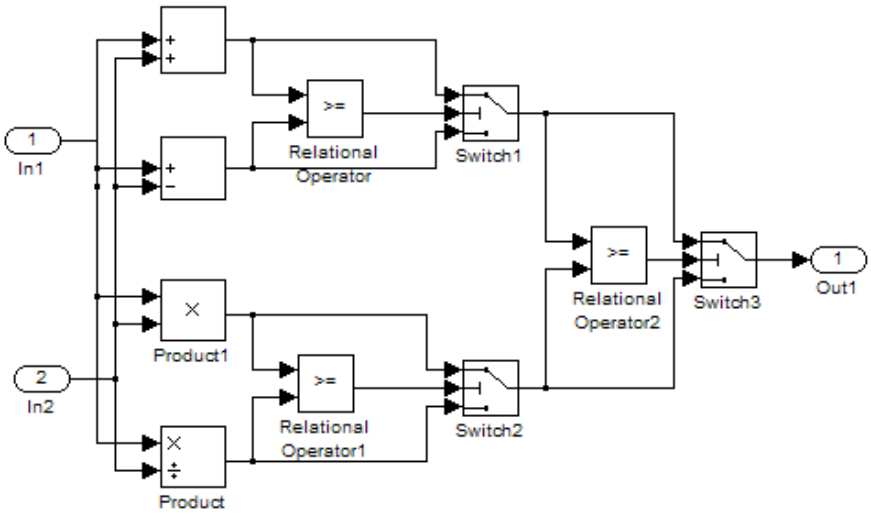

Figure 4-1 *Simulink* model branching structure.

To convert a single path coverage test generation problem into an optimisation problem is fairly straightforward. Firstly, we need to locate those 'Switch' blocks listed by the coverage requirement in the model and insert probes into the second input signal/line of the 'Switch' blocks. (The purpose of inserting probes is to view the runtime value of those points and use the information collected to direct moves of the test-data search. Therefore the probes are inserted by connecting the signal to an 'Out' block for observation.) The second step is to design the cost-function to evaluate the quality of an input test-datum according to three types of information: path requirement (satisfiability of 'Switch' blocks), threshold parameter value for each 'Switch' block, and values observed by probes. The basic concept behind the cost-function construction is to make the cost evaluation reflect the quality of the evaluated test-datum. Detailed cost function construction will be described in the section 4.2.
This part of work has been published in [ZC04].

4.1.2 Fault-based

The goal of fault-based testing is to make sure that the test-set applied can expose faults. That is to say: we expect errors to be detected by at least one of the test cases in the set. We assume that errors can be detected if and only if the model produces different outcomes from those expected. Therefore, according to the theory of fault-based testing as introduced in section 2.2, we conjecture a number of errors that might happen in the model design and try to generate test-data that can detect these errors. We systematically conjecture a number of mutant models from the model under test (although the original may be faulty and the mutant correct) and try to generate test-data that can distinguish the mutant models from the model under test. If such test-data can be found, we gain more confidence in that our test set may be able to discover the errors we conjecture if they exist in the system under test. At the model level, with a test-datum that is capable of distinguishing an original model from a mutant model, we can judge whether the original or the mutant is correct. We can also exercise an

implementation with such test-datum to determine whether the original model or mutant model has actually been implemented.

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

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

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

However, it is much more complicated to cause the input to achieve the second requirement, which is to make the difference at the fault-injection point affect the outputs. To fulfil this requirement, we need to trace down the structure of the model and make sure each point on the path from the fault-injection point through to the output differs between the two models. There may be a number of paths from the fault-injection point to any of the output ports of the system. In that case, we require at least one of them to propagate the difference (show the error).

Due to the special function of 'Switch' blocks, errors are often masked for certain inputs. Therefore in the process of test-datum search to detect a particular error, we need to identify such positions where 'Switch' blocks might disguise the error, and direct the search to test-data that can cause the effects of the error to propagate through the 'Switch' blocks.

With special consideration of 'Switch' blocks, the evaluation of test-datum for detecting a particular error can be defined in the following way (the probe insertion strategy is also included in the description below):

1. If there is a point in the mutant system where the value carried on the wire may be different from the corresponding point in the original system, and this point is connected to only one other block (non-Switch), as shown in Figure 4-2[10], then the cost (cost has been explained briefly in section 3) will be $C = C_D + C_{OP} + C_R$, where C_D is the cost of causing this point to make a difference between the two models; C_{OP} is the cost of causing the difference to show at point P (in other words, to show after going through the operation of block 'OP'); and C_R is the cost of causing the difference to ripple after the 'OP' block.

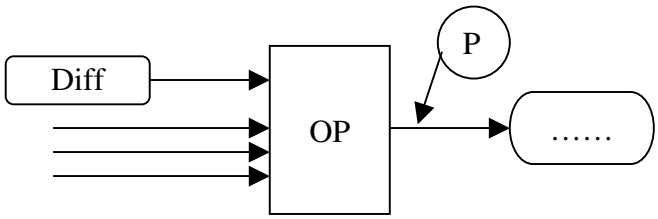

Figure 4-2

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

2. If there is a point in the mutant system where the value carried on the wire may be different from the corresponding point in the original system, and this point is connected to more than one other block, as shown in Figure 4-3, then the cost will be $C = C_D + (C_{P1} \vee C_{P2})$[11], where C_D is the cost of causing this point to make a difference between the two models, $C_{P1} = C_{OP1} + C_{RP1}$ and $C_{P2} = C_{OP2} + C_{RP2}$. C_{OP1} represents the cost of causing the difference to show at point P1 (in other words, to show after going through the operation of block 'OP1'), and C_{RP1} represents the cost of causing the difference to ripple after the 'OP1' block. C_{OP2} and C_{RP2} are defined likewise.

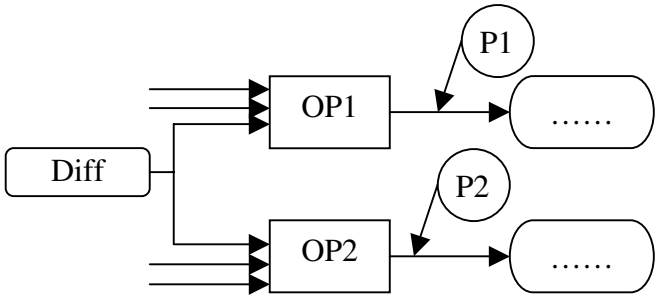

Figure 4-3

3. If there is a point in the mutant system where the value carried on the wire may be different from the corresponding point in the original system, and this point is connected to the first or third in-port of a 'Switch' block, as shown in Figure 4-4 and Figure 4-5, then the cost will be $C = C_D + C_{P1} + C_R$, where C_D is the cost of causing this point to make a difference between the two models, C_{P1} is the cost of causing the value at point P1 to satisfy (for the scenario in Figure 4-4) or dissatisfy (for the scenario in Figure 4-5) the branching requirement of the 'Switch' block and C_R is the cost of causing the difference to ripple after the 'Switch' block.

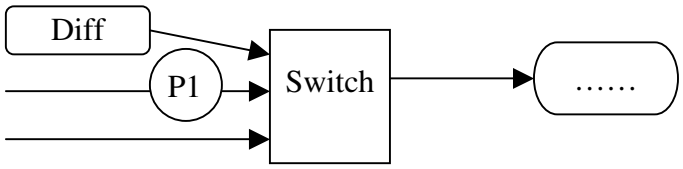

Figure 4-4

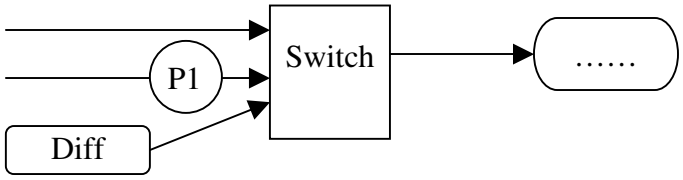

Figure 4-5

[11] Here the cost of $(C_{P1} \vee C_{P2})$ will be evaluated as the cost of either satisfying the predicate formula constructed at P1 or satisfying the predicate formula constructed at P2. Cost function evaluation of logical predicates is defined in section 4.2 Table 4-1.

4. If there is a point in the mutant system where the value carried on the wire may be different from the corresponding point in the original system, and this point is connected to the second in-port of a 'Switch' block, as shown in Figure 4-6, then the cost will be $C = C_D + (C_{P1P2} + C_{P3}) \vee (C'_{P1P2} + C'_{P3}) + C_R$, where: C_D is the cost of causing values carried at this point to make a difference between the two models; C_{P1P2} is the cost of causing the value at point P1 in the mutant model to be different from the value at point P2 in the original model; C_{P3} is the cost of causing the value at P3 to satisfy the branching requirement of the 'Switch' block in the mutant model but to dissatisfy the branching requirement in the original model; C'_{P1P2} is the cost of causing the value at point P2 in the mutant model to be different from the value at point P1 in the original model; C'_{P3} is the cost of causing the value at P3 to dissatisfy the branching requirement of the 'Switch' block in the mutant model but to satisfy the branching requirement in the original model; and C_R is the cost of causing the difference to ripple after the 'Switch' block.

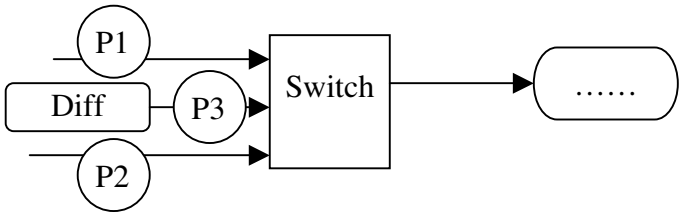

Figure 4-6

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

In a real application, the evaluation of C_{OP}, C_{OP1} and C_{OP2}, should be assessed according to the functionality carried out by the block 'OP', 'OP1' or 'OP2'. If this block is a subsystem, then detailed analysis into the block needs to be done in order to give a meaningful evaluation (the analysis can be performed by recursively applying the cost evaluation rules given above). For a basic block[12], e.g. a mathematical block, if the inputs are different, the outputs will often be different too. In this case, the cost will be '0'. If the outputs detected are the same (although there is little chance that this would happen), we assign a big value, say '10,000', as the cost. If this block is a logical block[13], to give a good cost evaluation that can reflect the test-datum quality, we need to chain-back to obtain the information of what contributes to the evaluation of the Boolean inputs of the logical block and form the cost function with that information. To simplify the problem, currently we do not address this kind of situation and we deal with this kind of block in the same way as dealing with other basic blocks.

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

[13] A logical block is a block that carries out logical calculations.

4.2 Cost function encoding

Table 4-1 illustrates the cost function encoding method that has been popularly used by other researchers who carried out search based test-data generation [Kor90], [TCMM98], [WBS01]. It also incorporates the cost function encoding ideas for logical operations put forward by Bottaci [Bot03], which enhance the cost functions of other researchers.

Table 4-1 Cost function encoding method.

Predicate	Value of Cost Function F
Boolean	if TRUE then 0, else *maxcost*
$E_1 < E_2$	if $E_1 - E_2 < 0$ then 0, else $E_1 - E_2 + \delta$
$E_1 \leq E_2$	if $E_1 - E_2 \leq 0$ then 0, else $E_1 - E_2$
$E_1 > E_2$	if $E_2 - E_1 < 0$ then 0, else $E_2 - E_1 + \delta$
$E_1 \geq E_2$	if $E_2 - E_1 \leq 0$ then 0, else $E_2 - E_1$
$E_1 = E_2$	if $Abs(E_1 - E_2) = 0$ then 0, else $Abs(E_1 - E_2)$
$E_1 \neq E_2$	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) × 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

Together with cost evaluation rules defined in section 4.1, such cost function encoding enables us to assess the 'distance' from the current test-datum to a desired test-datum. Below we will use two examples to illustrate how these cost function encoding and evaluation rules are used in real problems.

4.2.1 Structural-based

In structural-based testing, we wish to guide the search towards test-data that causes identified 'Switch' block branches to be taken. With each 'Switch' block there is a control parameter 'threshold'. If the run-time value 'Vp' of the second input port (whose value our probe monitors) of the 'Switch' block satisfies 'Vp ≥ threshold' then input port 1 is selected for output. If 'Vp < threshold' then input port 3 is selected. For any such identified condition, we can associate a cost indicating how far the current data is from satisfying the condition. Thus, if we require 'Vp≥20', say, then a 'Vp' value of 0 should have greater cost than a 'Vp' value of 19, since the latter 'nearly' causes the required predicate to be true, and the former clearly does not. According to the cost function encoding strategy listed in Table 4-1, the cost will be evaluated to be 20 and 1 respectively for the above situations.

In general (with models that maintain state) a test-datum will comprise a *sequence* of consecutive test inputs $<TI_1,\ldots, TI_k>$ over k time steps. Therefore we should evaluate this sequence based on the degree of achievement of goals at each step. Our goal (predicate) can be met at *any* step and so we need to evaluate the cost of a *disjunction* of predicates.

Now we use a full model to demonstrate how this works (see Figure 4-7):

Assume that:

The testing requirements are: Switch1 – unsatisfied, Switch2 – satisfied;

Threshold parameters are: Switch1 – 100, Switch2 – 50;

Input: step1: In1 – 5, In2 – 10; step2: In1 – 10, In2 – 100; step3: In1 – (-10), In2 – 50.

Therefore, the probes observed for the three steps should be:

Step1: Switch1probe – 15, Switch2probe – 15;
Step2: Switch1probe – 110, Switch2probe – (-90);
Step3: Switch1probe – 40, Switch2probe – 40.

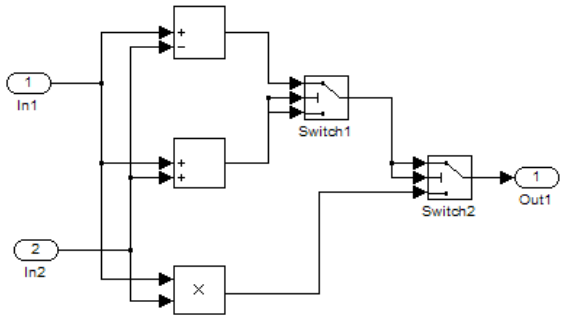

Figure 4-7 Example model.

The total cost of this test case will be:
$cost(((15<100)\wedge(15\geq 50)) \vee ((110<100)\wedge(-90\geq 50)) \vee ((40<100)\wedge(40\geq 50)))$
$C_1=cost((15<100)\wedge(15\geq 50))=35;$
$C_2=cost((110<100)\wedge(-90\geq 50))=(10+140)=150;$
$C_3=cost\ ((40<100)\wedge(40\geq 50))=10;$
$Cost=(\ C_1\ C_2\ C_3)/(\ C_1\ C_2+\ C_1\ C_3+\ C_2\ C_3)=7.394.$

4.2.2 Fault-based

Here we use the example illustrated in Figure 2-2 and Figure 2-3.
Problem description:

The fault is 'increment the value carried on the wire by 1' and is injected between block 'Sum' and block 'Product1'.

Parameters for the model: the threshold parameters for the blocks Switch1 to Switch4 are 8100, 1000, 8100, and 0 respectively.

Current test-input: (in the example, to simplify the problem, we check only one step of input) In1 = 5, In2 = 10, In3 = -20.

Probe insertion:

Probes will be inserted in the model as demonstrated in Figure 4-8 and Figure 4-9. The rationale of inserting probes in these places will be explained in the cost evaluation part.

Value notations:

Runtime value for point where probe 1 is inserted in the original model: Vop1;
Runtime value for point where probe 1 is inserted in the mutant model: Vmp1;
Likewise runtime values for other probe insertion points are: Vop2, Vop3, Vop4, Vop5, Vmp2, Vmp3, Vmp4 and Vmp5;
Thresholds for the 'Switch' blocks are Thres1, Thres2, Thres3, and Thres4 respectively.

Cost evaluation:

The first 'Diff' point is in the original model Figure 4-8, the point between 'Sum' and 'Product1', and in the mutant model Figure 4-9, the point after 'AddMut' block

(where 'Probe1' is inserted). Since block 'Product1' is the only block this point is connected to, the situation matches rule 1 in section 4.1.2. The cost is evaluated as

$$\mathbf{C = C_D + C_{OP} + C_R}$$

Equation 4-1

Here C_D = cost(Vop1≠Vmp1) = cost(15≠16) = 0;
C_{OP} = cost(Vop2≠Vmp2) = cost(-11250≠-12000) = 0;
To compute C_R, we need to move the 'Diff' point to where 'Probe2' is inserted, which is after block 'Product1', and analyse. Since 'Switch1' is the only block this 'Diff' point is connected to, The situation matches to rule 4 in section 4.1.2. The cost is evaluated as

$$\mathbf{C = C_D + (C_{P1P2} + C_{P3}) \vee (C'_{P1P2} + C'_{P3}) + C_R}$$

Equation 4-2

where C equals to C_R in Equation 4-1.
In Equation 4-2, C_D is equal to the cost that was before we reach this point, which has been calculated in Equation 4-1.
C_{P1P2} = cost(Vmp3≠Vop4) = cost(5≠-25) =0;
C_{P3} = cost((Vmp2≥Thres1)∧(Vop2<Thres1))
=cost((-12000≥8100) ∧(-11250<8100))
=19350;
C'_{P1P2} = cost(Vmp4≠Vop3) = cost(-25≠5) =0;
C'_{P3} = cost((Vmp2<Thres1)∧(Vop2≥Thres1))
= cost((-12000<8100) ∧(-11250≥8100))
=19350;
$(C_{P1P2} + C_{P3}) \vee (C'_{P1P2} + C'_{P3})$ = 19350×19350/(19350+19350) = 9675;
Again, in Equation 4-2 C_R relies on the cost analysis at the point after block 'Switch1'. Since 'Switch4' is the only block this point is connected to, the situation matches to rule 3 in section 4.1.2. The cost is evaluated as

$$\mathbf{C = C_D + C_{P1} + C_R}$$

Equation 4-3

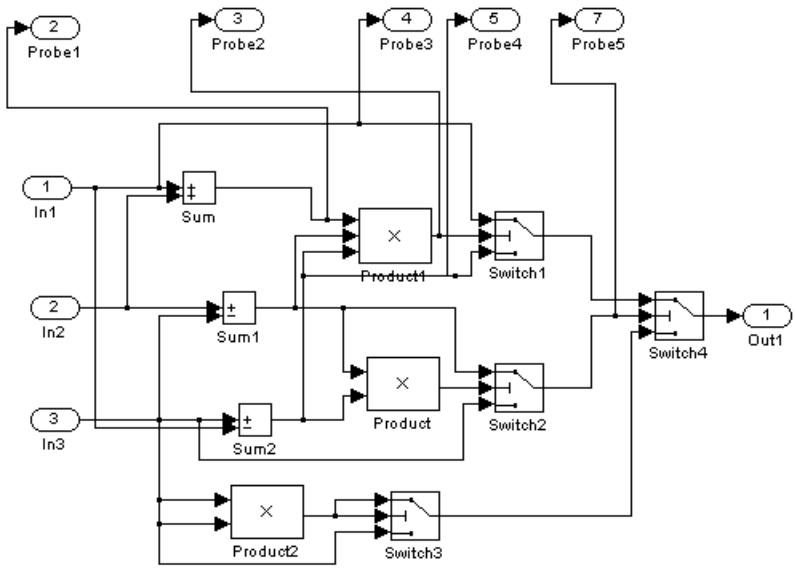

Figure 4-8 Original model after probe insertion.

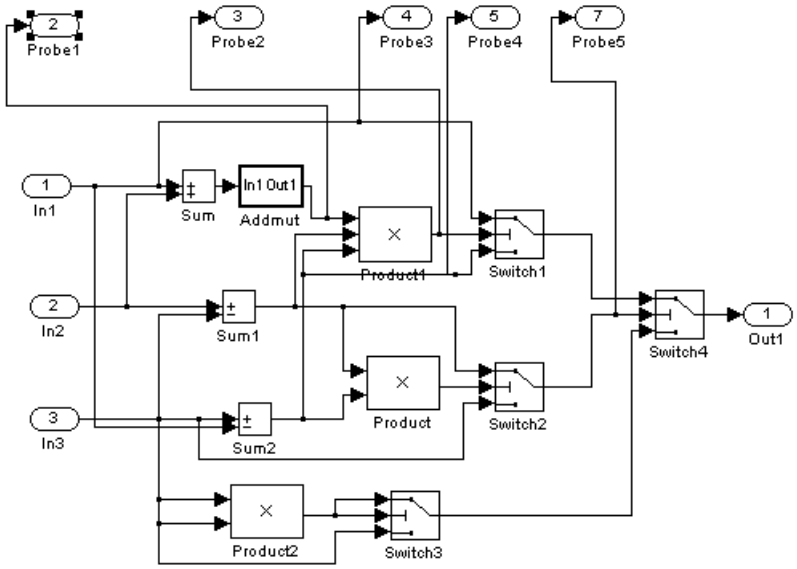

Figure 4-9 Mutant model after probe insertion.

Similarly, in Equation 4-3, C_D is equal to the cost that was before we reach this point, which has been calculated in Equation 4-2 and Equation 4-1. In Equation 4-3,
C_{P1} = cost((Vmp5≥Thres4)∧(Vop5≥Thres4)) = cost((-20≥0)∧(-20≥0)) = 40;
Since the 'Switch4' block is connected to an 'Out' block, there is no cost to make this point ripple the difference and therefore the cost of C_R is 0.
Therefore, the total cost (for Equation 4-1) is: 0 + 9675 + 40 = 9715.

4.3 Optimisation search

In this work, we have used the well-established technique of simulated annealing [Ree93], a *local* search technique that allows escape from local optima. The annealing algorithm we apply is provided below. Interested readers are referred to [KGV83] and [MRRTT53] for more details about the annealing algorithm. In our application a move effectively perturbs the value of one of the inputs in the current test sequence by a value less than or equal to 1% of the range of the input. We applied a geometric cooling rate of 0.9. The number of attempted moves at each temperature was 500, with a maximum of 100 iterations (temperature reductions) and a maximum number of 30 consecutive unproductive iterations (i.e. with no move being accepted). These parameters may be thought to be on the 'small' side, but the computational expense of simulation requires us to make pragmatic choices.

Select an initial solution $testData_0$;
Select an initial temperature $t_0>0$;
Select a temperature reduction function α(=0.9 here);
Repeat
    Repeat
        Generate a move $testData \in N(testData_0)$;
        $\delta = f(testData) - f(testData_0)$;
        If $\delta < 0$
            Then $testData_0 = testData$;
        Else
            Generate random x uniformly in the range (0, 1);
            If $x < \exp(-\delta/t)$ then $testData_0 = testData$;
        End;

Until *innerLpCount* = *maxInnerLpNo* or $f(testData_0)$satisfies the requirement;
Set t = α (*t*);
Until *outerLpCount* = *maxOuterLpNo* or *nonAcceptCount* = *maxNonAcceptNo* or $f(testData_0)$ satisfies the requirement.
$testData_0$ is the desired test-data if $f(testData_0)$ satisfies the requirement.

We also realised that simulated annealing algorithm is sensitive to the parameter settings and the move strategy definition. Our next phase of work will focus on investigating these factors in order to deliver a technology that is not problem-specific.

5 Case study

Random testing can easily fulfil part of our testing requirements. But there are usually a certain requirements that cannot or are difficult to be covered by random testing. For example, the model in Figure 2-2, to find a test input that can cause the execution of an over-restricted path (Switch1-unsatisfied, Switch2-satisfied, Switch3-satisfied, Switch4-satisfied)[14], random test-data generation tried 5,000 cases, but failed. Similarly random test-data generation failed in generating any test input that can distinguish the faulty model[15] illustrated in Figure 2-3 from its original one even after 50,000 attempts. We expect that our targeted test-data generation tools can generate such 'hard' test-data at a moderate cost.
Preliminary case study results are provided in this section to demonstrate that the two targeted test-data generation tools can, to a certain extent, generate test-data that are hard to produce by random test generation.

5.1 Automatic test-data generation for structural testing

We used our tools to generate *all-paths-coverage* test-data, attempting each path execution aim in turn. For each test aim (e.g. for each path), up to 50,000 tests were allowed to be tried (both for annealing and for random generation). When a test-datum was found that achieved the aim, each procedure was terminated. The total numbers of test cases executed over all test aims (e.g. model 'Quadratic' has 8 test aims, 'Combine has 128 test aims) are recorded in the table below. The coverages achieved are shown too. We can see that our instrumented test-data generation approach achieved greater coverage with fewer executions for most of the cases. All the models used are hand-crafted and designed for providing difficulties in generating test-data for covering some paths.

Model Name	Model Size	'Switch' Block No.	SimAnneal Case No	Random Case No	SimAnneal Coverage	Random Coverage
SmplSw	8 blocks	2	2,275	1,896	4/4	4/4
Quadratic	15 blocks	3	2,096	25,393	8/8	8/8
RandMdl	14 blocks	4	45,756	347,620	16/16	10/16
Combine	29 blocks	7	1,144,828	3,908,000	126/128	52/128

Our observation is that the first model 'SmplSw' is rather straightforward for locating test-data. Our instrumented test-data search procedure did not show any advantages compared to random test-data generation. However the next three models present greater difficulty. Our instrumented test-data generation approach achieved greater coverage with far fewer

[14] The threshold parameters for the blocks Switch1 to Switch4 are 8100, 1000, 8100, and 0 respectively.

[15] The fault is injected on the wire/line connecting block Product1 (port 1) and Switch1 (port 2) with a mutation parameter of 'Add by 1'.

executions. We noticed that for the 'Combine' model, there are a couple of paths that failed to be covered by our search-based test-data generation in the batch run. For these two paths, we ran our search-based test-data generation once again and found that both desired test-data could be found by this technique within the number of allowed test-data evaluations. Since the simulated annealing approach is a variant of the heuristic technique of local search, it may sometimes result in convergence of a local optimum. In our test-data search, it may result in not being able to provide a satisfactory solution. To overcome this problem, the approach of repeating the algorithm using several different starting solutions is suggested.

5.2 Automatic test-data generation for mutation testing

The following table shows the comparison of using Simulated Annealing search and Random search for the fault-based test-data generation task. Each task is specified with the name of the model under test and the fault description (including fault-injection source and destination block, mutation operator and mutation parameter). The result is based on the average of 10 individual runs of the program. All the test generation tasks demonstrated here are selected because a random test-set of 500 test cases failed in generating satisfactory tests.

Here we give an illustration of what the data in the table represent. The first row of data means: in the 'Quadratic' model, we inject a fault on the wire connecting block 'Product2' port '1' and block 'Switch2' port '2'. The fault adds '1' to the value carried on the wire. The simulated annealing search based test-data generation tool consumed 274.7 tries on average (the result is based on 10 individual runs) to find the desired test-data. The random test-data generation cost 2,689.7 tries on average (the result is also based on 10 individual runs) to find the appropriate test-data. This case is selected because the test-data generation is not as easy as the other cases; we tried 500 random cases but failed to satisfy the test requirement while most of the other cases can be satisfied within 500 random test generation tries.

Model Name	SrcBlock & SrcPort	DstBlock & DstPort	MutOp	MutPara	SimAnneal Case No	Random Case No
Quadratic	Product2(1)	Switch2(2)	Add	1	274.7	2,689.7
Quadratic	Product2(1)	Switch2(2)	Add	-1	444.3	2,018.4
RandMdl	Product(1)	Switch2(2)	Add	1	8,008.1	>26,219.5 (5)
RandMdl	Sum1(1)	Product1(2)	Add	-1	>16,575.4 (2)	649.2
RandMdl	In1(1)	Sum(1)	Add	-1	>12,643.3 (2)	507.9

The test-data generation (both simulated annealing approach and random approach) procedure were allowed up to 50,000 test executions. For some models, each run produced a successful outcome with this limit. In others, only some of the runs did so. The "SimAnneal Case No" is the average number of executions per run. The number in parenthesis is the number of successful runs. It is likewise with "Random Case No".

As we can see, the simulated annealing test-data search outperformed the random test-data search on some cases but performed worse than random test generation on some other cases. This might be due to the different distribution of the data. It is observed that simulated annealing performs better on uniform data than on data that is clustered in some way [Ree93]. It is suggested that allowing some form of reheating may improve the performance for the clustered data distribution. Further study needs to be done to enable the search allow more frequent escapes from deep troughs.

6 Conclusion

The basic aim of this work is to facilitate test automation at the architectural level. The case study results demonstrated some encouraging facts for applying the optimization-based search technique to generate test-data. However, the technique did not work for all the cases. Further study about tuning the parameters of simulated annealing search is needed to optimize the performance of the technique.
The conceptual framework should extend to other architectural notations provided that the notation selected supports execution or simulation. Should other advanced optimisation-based search technique or constraint solving techniques prove superior for some problems, such emerging tools can be easily incorporated.

References

[Bei90] B. Beizer. **Software Testing Techniques**. *Thomason Computer Press, 2nd edition. 1990.*

[Bot03] Leonardo Bottaci. **Predicate Expression Cost Functions to Guide Evolutionary Search for Test Data**. *GECCO 2003.*

[Kor90] B. Korel. **Automated Software Test Data Generation**. *IEEE Transactions on Software Engineering, 16(8): 870-879, August 1990.*

[KGV83] S. Kirkpatrick, C. Gelatt, and M. Vecchi. **Optimization by Simulated Annealing**. *Science, 220(4598): 671-680, May 1983.*

[MRRTT53] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. **Equations of State Calculations by Fast Computing Machines**. *Journal of Chemical Physics 21, 1087-1091. 1953.*

[Ree93] C. R. Reeves (Ed.). **Modern Heuristic Techniques for Combinatorial Problems**. *Blackwell Scientific Publications, Oxford, 1993.*

[TCMM98] Nigel Tracey, John Clark, Keith Mander and John McDermid. **An Automated Framework for Structural Test Data Generation**. *Automated Software Engineering 1998, Honolulu.*

[VM97] Jeffrey Voas and Gary McGraw. **Software Fault Injection: Innoculating Programs Against Errors**. *By John Wiley & Sons, 1997.*

[WBS01] J. Wegener, A. Baresel, and H. Sthamer. **Evolutionary Test Environment for Automatic Structural Testing**. *Information and Software Technology, 43: 841-854, 2001.*

[ZC04] Y. Zhan and J. Clark. **Search Based Automatic Test-Data Generation at an Architectural Level**. *GECCO 2004.*

[ZHM97] Hong Zhu, Patrick A. V. Hall and John H. R. May. **Software Unit Test Coverage and Adequacy**. *ACM Computing Surveys, Vol. 29, No. 4 December 1997. annoy*