Search Based Automatic Test-Data Generation at an Architectural Level

Yuan Zhan and John Clark

Department of Computer Science
University of York
York YO10 5DD, UK
{yuan, jac}@cs.york.ac.uk

Abstract. The need for effective testing techniques for architectural level descriptions is widely recognised. However, due to the variety of domain-specific architectural description languages, there remains a lack of practical techniques in many application domains. We present a simulation-based testing framework that applies optimisation-based search to achieve high-performance testing for a type of architectural model. The search based automatic test-data generation technique forms the core of the framework. *Matlab/Simulink* is popularly used in embedded systems engineering as an architectural-level design notation. Our prototype framework is built on *Matlab* for testing *Simulink* models. The technology involved should apply to the other architectural notations provided that the notation supports execution or simulation.

1 Automatic Testing at the Architecture Level

Software testing is an expensive procedure. It typically consumes more than 50% of the total development budget [1]. 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 sufficient confidence in system rigour. Automated test-data generation is one 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. *Matlab/Simulink* is a widely used notation in dynamic systems development industry that allows models to be created and exercised. *Mat-*

K. Deb et al. (Eds.): GECCO 2004, LNCS 3103, pp. 1413–1424, 2004.
© Springer-Verlag Berlin Heidelberg 2004

lab/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. We are aware that *Matlab* itself provides a '*Simulink Performance Tool Set*', which aids auto testing. However it's functionality is restricted to only measuring test completeness. In this work, we focus on automatically generating effective test-data for testing *Matlab/Simulink* models. Other authors have recognized the practical significance of such modeling and the need to provide assurance information automatically, e.g. the worst case execution times for such models [11].

An *adequacy criterion* is a criterion that defines what constitutes an *adequate* test-set [5]; it provides a measure of how effective a given test-set is. The ability to compare test-sets allows the tester to identify how to add tests to an existing set to improve effectiveness of the overall set. (Additional tests should lead to an improvement in the measure of effectiveness.) [5] introduced different types of test adequacy criteria; these can be generally categorized as: structural-based, fault-based and error-based. 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.

Code level coverage criteria are explained in [5]. However, these can be adapted to specification and architecture level testing too. Our work is concerned with interpretations of widely used structural coverage criteria. We implemented an automatic test-data generation tool to cover particular *paths* of *Simulink* models. Combined with random test generation, this tool enables efficient automation of structural coverage test-data generation. The construction of the structural coverage test generation tool is detailed in the next section.

2 Search Based Automatic Test-Data Generation

Test data generation has been a very successful branch of search based software engineering. Most work however, has been at the code level (e.g. [2,6,8,9, 11,12,13,14]). The reader is referred to the authoritative survey [18] for a thorough overview of the field. Below we explain how we implemented the automatic structural coverage test-data generation tool for testing *Simulink* models. First, we give some background on *Simulink* and then describe our strategies for applying the test-data generation technique.

Simulink Introduction. *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.

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

Each block can be a subsystem comprising other blocks and lines. Fig. 1 is a simple *Simulink* model.

Simulink models have their special way of forming branches compared to programs. Basically *Simulink* uses the 'Switch' block or its derivatives, like the 'Multiport Switch' block, to form branches. A 'Switch' block has three 'in' ports, one 'out' port and there is a threshold value associated with the block. When the value of the second 'in' port is greater than or equal to the threshold parameter, the output will equal to the value carried on the first 'in' port, otherwise the value carried on the third 'in' port will be channelled through to the output. Therefore a 'Switch' block can map to an 'if … then … else' branching structure in code.

Simulink models execute (calculate the outputs of) all branches of the models, whether the branches are selected or not, while for programs, only the selected branches are executed. For example, the following code matches the model in Fig. 1. In the *Simulink* model, both 'x−y' and 'y−x' are calculated although only one of these results is channelled through to the output by the 'Switch' block. However, in the code, only one of them will be executed depending on the evaluation of the predicate (x>=y).

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

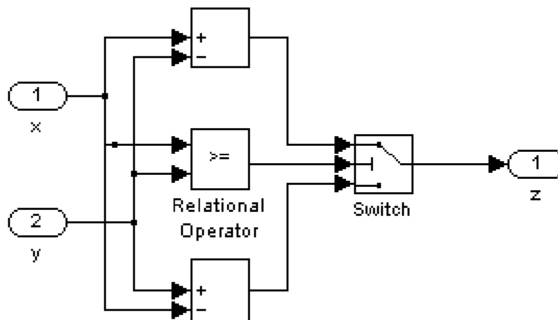

Fig. 1. An example of a *Simulink* model

Simulink is generally used for designing embedded systems – of which a significant feature is that they 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 millisecond. Therefore, for the model in Fig. 1, the input to the system over n time steps should be a sequence < $(x_1, y_1), (x_2, y_2), \dots (x_n, y_n)$ >, and the corresponding output should also be a sequence < $z_1, z_2, \dots z_n$ >.

Interpreting a Test-Data Generation Problem as a Search Problem. In the prototype tool implementation we consider only models whose branching blocks are 'Switch' blocks. A requirement for the generation of a particular structural coverage test input comprises specifying a subset of all 'Switch' blocks involved together with the required condition values (satisfied or unsatisfied). We can consider such a requirement as the equivalent of a 'sub-path' coverage requirement in programs. A single test-data generation requirement for the model in Fig. 2[2] might be: Switch2 = satisfied, Switch3 = unsatisfied (which means the outcome of block 'Product1' is channelled through 'Switch2', and the outcome of block 'Switch2' is channelled through 'Switch3' to the final output). Some combinations of 'Switch' conditions may be over-restrictive, e.g. in Fig. 2, 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 weather block 'Switch2' is to be satisfied or not because the outcome of 'Switch2' would not affect the model outcome anyway. Over-restrictive combinations may also be infeasible.

Fulfilment of structural adequacy criteria will require a test-set to exercise identified combinations of 'Switch' predicates. We may impose a simple 'branch 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*).

The automatic test-data generation for satisfying each path coverage requirement is fairly straightforward. Firstly, we need to locate the 'Switch' blocks listed by the coverage requirement in the model and insert probes into the second input signal/line of those 'Switch' blocks. (The purpose of inserting probes is to view the runtime values 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 'Outport' 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. Detailed cost function construction will be described in the next sub-section. Then we need to apply dynamic search procedure to search for desired test-data. The search will be based on the simulation of models with candidate test-data as inputs. Each simulation provides information about how good the current candidate test-datum is. The usage of the optimisation search techniques will be detailed in sub-section after the next.

[2] The 'Threshold' parameter of all three 'Switch' blocks in the figure has value '0'.

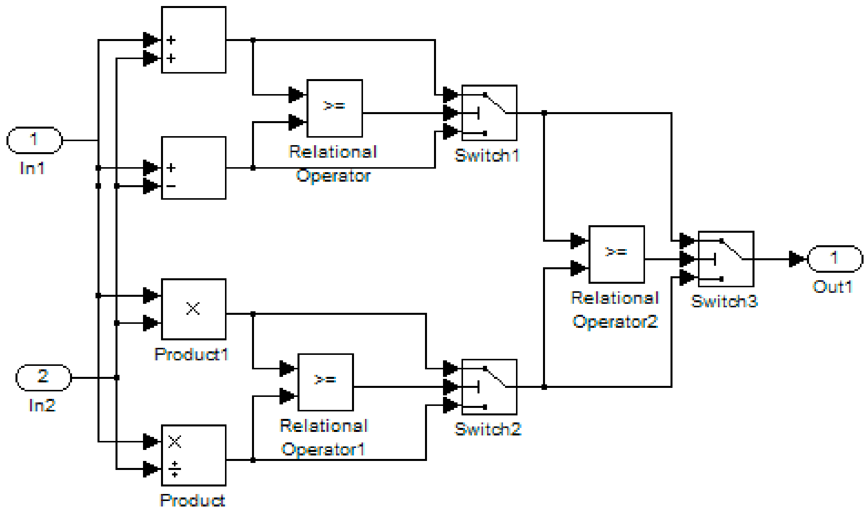

Fig. 2. *Simulink* model branching structure

Cost Function Design. We wish to guide the search towards test-data that causes identified 'Switch' block branches to be taken. With each 'Switch' block we call the 'Threshold' parameter 'para'. If the run-time value 'Vp' of the second input port (whose value our probe monitors) of the 'Switch' block satifies 'Vp ≥ para' then input port 1 is selected for output. If 'Vp < para' then input port 3 is selected. For any such identified condition, we can associate a cost indicating how far the current data is from satsifying 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. The cost function encoding scheme we apply for such relational predicates is illustrated in Table 1. Similar approaches have been used by Korel [9], Tracey et al. [6], Wegener et al. [11], Jones et al. [8] etc. But cost function encoding for logical operations also needs to be defined because when combining all the branching requirements together, we need to calculate the cost of a *conjunction* of various relational predicates. 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. We will need to evaluate this sequence based on the degree of achievement of goals at each step. We need our goal (predicate) to be met at *any* step and so need to evaluate the cost of a *disjunction* of predicates. Bottaci [2] suggested a set of cost function encodings for logical operations that can be more accurate in reflecting the fitness of test-data compared to that of other researchers. We adapt his idea for our application as shown in Table 1.

Table 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 $E_1 - E_2 + \delta$
$E_1 \le E_2$	if $E_1 - E_2 \le 0$ then 0, else $E_1 - E_2$
$E_1 > E_2$	if $E_2 - E_1 < 0$ then 0, else $E_2 - E_1 + \delta$
$E_1 \ge E_2$	if $E_2 - E_1 \le 0$ then 0, else $E_2 - E_1$
$E_1 = E_2$	if $Abs(E_1 - E_2) = 0$ then 0, else $Abs(E_1 - E_2)$
$E_1 \ne E_2$	if $Abs(E_1 - E_2) \ne 0$ then 0, else K
$E_1 \vee E_2$ (E_1 unsatisfied, E_2 unsatisfied)	(cost (E_1) $\times$ 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

Here is an example of using the above encoding scheme (see the model in Figure 3):

Assume that:

The testing requirements are: Switch**1** to be unsatisfied, Switch**2** to be satisfied;

Threshold parameters are: Switch**1**para = 100, Switch**2**para = 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: Switch**1**probe = 15, Switch**2**probe = 15;

Step2: Switch**1**probe = 110, Switch**2**probe = (-90);

Step3: Switch**1**probe = 40, Switch**2**probe = 40.

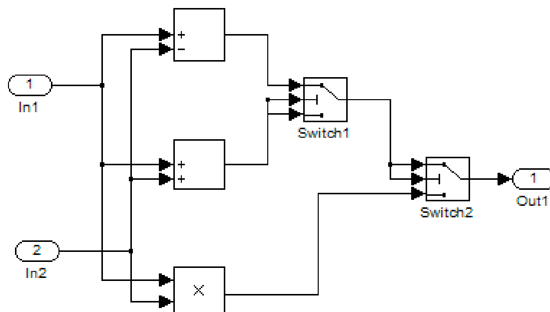

Fig. 3. *Simulink* model for cost function encoding scheme demonstration

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

We have carried out multi-step evaluation for illustration only. (The system obviously has no feedback within it, so a sequence of length 1 would be entirely appropriate in practice). As a result of such cost function encoding scheme, the test-data generation problem can be interpreted as a search for a test-datum that can minimize the underlying cost function. The target is zero. Next we provide a brief outline of the optimization technique we used – simulated annealing, and its application details.

Optimization Techniques. In this framework, we have used the well-established technique of simulated annealing [7] to search for the desired test-data. Simulated annealing is a global optimization heuristic that is based on the *local* descent search strategy. The annealing algorithm we apply is shown below.

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 $testData_0 = testData$;
    Until *innerLpCount* = *maxInnerLpNo* or $f(testData_0)$satisfies the requirement;
    Set $t = \alpha(t)$;
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.

The initial solution is usually randomly generated. Then the search keeps generating, considering and possibly moving to local neighborhood solutions of the current solution. A move is accepted if it improves the evaluation of the cost function. A worsening move may also be accepted probabilistically in a way that depends on the temperature t in the search. The higher the temperature is, the easier a worsening solution can be accepted. Initially the temperature is high and a lot of worsening solutions may be accepted. As the time passes by, the temperature drops and eventually it ‘freezes’ and therefore no worsening solutions can be accepted. A number of moves are considered at each temperature. If no move has been accepted for some time then

the search halts. For a problem that does not require reaching global optima, the search procedure halts at any time when a satisfactory solution is found.
Interested readers are referred to [15], [7] and [10] for more details about the annealing algorithm. In our application a move effectively perturbs the value of one of the inputs in the current test sequence by a value less than or equal to 1 percent of the range of the input. We applied a geometric cooling rate of 0.9. The number of attempted moves at each temperature was 500, with a maximum of 100 iterations (temperature reductions) and a maximum number of 30 consecutive unproductive iterations (i.e. with no move being accepted). These parameters may be thought to be on the 'small' side, but the computational expense of simulation requires us to make pragmatic choices.

3 Automating Structural Coverage Test Generation

In terms of structural coverage, we evaluate test-sets by assessing the percentage of paths being covered by the tests. As has been explained in section 2, in *Simulink*, the branches are typically caused by 'Switch' blocks. (We exclude the usage of the other branching blocks in this prototype framework.) Therefore the *all-paths-coverage* can be defined as having all *combinations* of the 'Switch' block satisfaction conditions being covered. For example, in Fig. , there are altogether four *full* paths[3]. They are:

1) 'Switch1'=satisfied and 'Switch2'=satisfied;
2) 'Switch1'=satisfied and 'Switch2'=unsatisfied;
3) 'Switch1'=unsatisfied and 'Switch2'=satisfied;
4) 'Switch1'=unsatisfied and 'Switch2'=unsatisfied.

A test-set having 10 test cases but covering only path 1) and 3) would be evaluated as having 50% path coverage. By this means, we can evaluate the structural coverage capability of test-sets by running all the test cases within the underlying test-set against the model under test and recording the paths being covered by those test cases. The more paths can be covered, the better the test-set is.

There are usually multiple test cases executing the same path in a random test-set. For the sake of efficiency, we may want to remove the test cases that can not increase the structural coverage of the test-set. For our *all-paths-coverage* testing, each test case covers one and only one full path. So the redundant test case removal is straightforward. For less stringent sub-path coverage requirements (e.g. the equivalent of 'branch coverage') test-set reduction may be more sophisticated.

To efficiently automate the structural coverage test data generation, we propose to combine the random test-data generation and our targeted test-data generation together, which means we generate a moderate sized random test-set and check the coverage capability of it first, then use our automatic targeted test-data generation tool to generate test-data that can cover those paths which were not covered by the

---

[3] A *full path* specifies the branching preference of all 'Switch' blocks, while a *sub-path* specifies the branching preference of only a subset of the 'Switch' blocks in the model. A sub-path coverage test requirement is less stringent than a full path coverage requirement.

initial random set. We use random testing because it generally can achieve a certain amount of coverage at a very low cost (cheaper than applying the optimization based search technique). For those paths that are difficult to be covered by the random test-set, we use our instrumented test-data generation tool, expecting to find the desired test-data quicker than random search.

Since our test-data generation tool is geared towards the 'hard'[4] targeted testing aims, To demonstrate its effectiveness and efficiency, we compare it with random test-data generation in both the coverage capability and the number of test cases tried during the searching of test-data.

In the experiment, we automatically generate a random test-set of the size[5] that doubles the path number first. E.g. for a model that has 3 'Switch' blocks, and therefore has 8 full paths, we generate a random test set of the size of 16. Then we mark the paths that the random set can cover. For those paths that are not covered by the random test-set under evaluation, we use our automatic test-data generation tool to generate test-data to cover them and increase the structural coverage of our test-set.

Therefore we compare both the coverage capability and the number of test cases tried during the searching of test-data. The comparison results are recorded in Table 2. We tried both approaches (random test generation and simulated annealing search based test generation) on 4 models: 'SmplSw', 'Quadratic', 'RandMdl', and 'Combine'. All the models used are hand-crafted and designed for providing hardness in generating test-data for covering some paths. In the table, for model 'Quadratic', we generate a random test set with 16 test cases, and it covered 3 paths. For the remaining 5 paths, we use both simulated annealing approach and random approach to generate test data that can cover them, attempting each path execution aim in turn. Simulated annealing on its own used 1,641 cases and random approach tried 25,377 cases. Both approaches reached a full coverage eventually. Therefore the total test case number used by each of them are 1,657 and 25,393 respectively (16 cases from the initial random set).

For each path execution aim up to 50,000 tests were allowed (both for annealing and for random generation). Therefore there are some paths that cannot be reached within our effort allowance for model 'RandMdl' and 'Combine'.

Our observation is that the first model 'SmplSw' is rather straightforward for locating test-data. All paths can be covered by the initial small random test set of 8 test cases. And there was no need to use the our instrumented test generation tool. However the next three models present greater difficulty. Our instrumented test-data generation approach achieved greater coverage with fewer 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 tried to run our search-based test-data generation once again and found out that both desired

[4] By 'hard' we mean those testing aims that are difficult to be covered by a random test-set.

[5] The optimal size of the initial random test-set varies from model to model. Research needs to be done to investigate how to set this size.

Table 2. Case study result for the automatic test-data generation tool

Model Name	Model Size	'Switch' Block No.	SimAnneal Case No	Random Case No	SimAnneal Coverage	Random Coverage
SmplSw	8 blocks	2	8	8	4/4	4/4
Quadratic	15 blocks	3	1,657	25,393	8/8	8/8
RandMdl	14 blocks	4	38,161	347,605	16/16	10/16
Combine	29 blocks	7	1,062,993	3,907,080	126/128	52/128

test-data could be found by this technique within the lengths we allowed to try. The simulated annealing approach 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.

4 Conclusions and Future Work

The basic aim of this work is to facilitate test automation at the architectural level. We have adopted a two-pronged attack strategy. The major tool is the automated test generation facility. This has been applied to generate the architectural equivalent of structural coverage tests and is entirely automatic. We believe that this can easily be extended to provide architectural analogues of the various code-level test applications (e.g. the safety analysis, exception generation and falsification testing of Tracey et al. [12,13,14]). For example, to carry out safety analysis, safety invariant checkers are inserted into the model under test as probes. Therefore for different test inputs, we may observe from the safety invariant checkers how close the test-data comes to breaking the safety invariant. Such information can direct our optimisation heuristic search for the test-data. This method can be used at the early stage of system safety analysis; it may show cheaply (because it is fully automated) that some safety property does not hold. However to *show rigorously* that some safety property *holds*, we will have to rely on formal methods or other rigorous techniques.

The second avenue of attack is simply to observe that random test-sets are usually cheap in achieving a moderate coverage but they contain significant redundancy. We propose to combine the random test generation with our targeted test generation to achieve high structural coverage with comparatively low cost. Meanwhile we remove the redundant test cases from the random set. Currently our full path coverage test requirement ensures such redundancy removal to be straightforward. As stated in the text, less stringent requirements will make the redundancy removal a computationally hard problem. We plan to use optimisation techniques to provide a subset extraction facility later on. This approach requires only that you know what each test input actually achieves. The test-data could be generated by any method. Thus, this has the benefit of being able to be directly applied in almost any industrial test process. Similar ideas have been applied to regression test-sets [16].

This framework is conceptually extensible. As has been mentioned in the text, we intend extending the test-data generation tool to automatically generate test-data that can detect particular faults. The conceptual framework should extend to other architectural notations provided that the notation selected supports execution or simulation. Should there be any other advanced optimisation-based search technique or constraint solving techniques proved to be superior for some problems, such emerging tools can be easily incorporated. Many interesting questions arise. Can the architectural level test-data be used for code testing? What kind of refinement needs to be done? Can the refinement be done automatically? If so, there will be a large payback. If the developers maintain a fairly straightforward mapping of inputs and outputs when refining to code then the task is greatly facilitated.

Testing and analysis at the architectural level is now considered a crucial part of effective software development. We believe that our emerging automated test-data generation and test-set reduction techniques applied at the architectural level can form a useful complement to other automated techniques such as model checking and proofs of correctness.

References

1. B. Beizer: Software Testing Techniques. Thomason Computer Press, 2nd edition. 1990.
2. Leonardo Bottaci: Predicate Expression Cost Functions to Guide Evolutionary Search for Test Data. GECCO 2003.
3. J. Beiman, D. Dreilinger and L. Lin: Using Fault Injection to Increase Software Test Coverage. In Proc. 7th Int. Symp. on Software Reliability Engineering (ISSRE'96).
4. Jeffrey Voas and Gary McGraw: Software Fault Injection: Innoculating Programs Against Errors. By John Wiley & Sons, 1997
5. 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.
6. Nigel Tracey, John Clark, Keith Mander and John McDermid: An Automated Framework for Structural Test Data Generation. Automated Software Engineering 1998, Honolulu.
7. S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. Science, 220(4598): 671-680, May 1983.
8. B. F Jones, H. Sthamer, and D. E. Eyres. Automatic Structural Testing Using Genetic Algorithms. Software Engineering Journal, 11(5): 299-306, 1996.
9. B. Korel. Automated Software Test Data Generation. IEEE Transactions on Software Engineering, 16(8): 870-879, August 1990.
10. C. R. Reeves (Ed.). Modern Heuristic Techniques for Combinatorial Problems. Blackwell Scientific Publications, Oxford, 1993.
11. J. Wegener, A. Baresel, and H. Sthamer. Evolutionary Test Environment for Automatic Structural Testing. Information and Software Technology, 43: 841-854, 2001.
12. Nigel Tracey, John Clark, John McDermid and Keith Mander. Integrating Safety Analysis with Automatic Test-Data Generation for Software Safety Verification. 17th International System Safety Conference. Pages 128-137. August 1999.
13. Nigel Tracey, John Clark, Keith Mander and John McDermid. Automated test-data generation for exception conditions. Software Practice and Experience, January 2000.

14. Nigel Tracey, John Clark and Keith Mander. Automated Program Flaw Finding using Simulated Annealing. International Symposium on Software Testing and Analysis (ISSTA) 1998.
15. N. Metropolis, A. W. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculation by Fast Computing Machine. Journal of Chem. Phys., 21:1087-1091, 1953.
16. Ghinwa Baradhi and Nashat Mansour. A Comparative Study of Five Regression Testing Algorithms. In the Proceedings of the Australian Software Engineering Conference, 1997.
17. R. Kirner, R. Lang, G. Freiberger and P. Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. 14th Euromicro Conference on Real-Time Systems (ECRTS'02), Austria, 2002.
18. Search Based Software Test Data generation: A Survey. Phil McMinn. Preprint (to appear in STVR). http://www.dcs.shef.ac.uk/~phil/pub/sbst.pdf