Comparing Algorithms for Search-Based Test Data Generation of Matlab® Simulink® Models

Kamran Ghani, John A. Clark and Yuan Zhan

***Abstract*— Search Based Software Engineering (SBSE) is an evolving field where meta-heuristic techniques are applied to solve many software engineering problems. One area of SBSE, where considerable research is underway, is software testing. We see much application of meta-heuristics search techniques for generating input test data. But most of the work in this area is concentrated on test data generation from source code. We see very little application of such techniques to testing from other sources such as requirement and design models.**

Zhan and Clark applied such techniques to generate test data for Simulink models. This paper extends the work of Zhan and Clark by investigating the application of Genetic Algorithms (GAs) to Simulink models and then statistically compares the results to the existing work, which is mainly based on Simulated Annealing (SA).

I. Introduction

A. Dynamic Testing

Dynamic testing — "the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behaviour" [1] — is used to gain confidence in almost all developed software. Various static approaches can be used to gain further confidence but it is generally felt that only dynamic testing can provide confidence in the correct functioning of the software in its intended environment.

We cannot perform exhaustive testing because the domain of program inputs is usually too large and also there are too many possible execution paths. Therefore, the software is tested using a suitably selected set of test cases. A variety of coverage criteria have been proposed to assess how effective test sets are likely to be. Historically, criteria exercising aspects of control flow, such as statement and branch coverage [2], have been the most common. Further criteria, such as data flow [3], or else sophisticated condition-oriented criteria such as MC/DC coverage [4] have been adopted for specific application domains. Many of these criteria are motivated by general principles (e.g. you cannot have much confidence in the correctness of a statement without exercising it); others target specific commonly occuring fault types (e.g. boundary value coverage).

Finding a set of test data to achieve identified coverage criteria is typically a labour-intensive activity consuming a good part of the resources of the software development process. Automation of this process can greatly reduce the cost of testing and hence the overall cost of the system. Many automated test data generation techniques have been proposed by researchers. We can broadly classify these techniques into three categories: random, static and dynamic[5] [6].

Random approaches generate test input vectors with elements randomly chosen from appropriate domains. Input vectors are generated until some identified criterion has been satisfied. Random testing may be an effective means of gaining an adequate test set for simple programs but may simply fail to generate appropriate data in any reasonable time-frame for more complex software (or more sophisticated criteria).

With static techniques an enabling condition is typically generated that is satisfied by test data achieving the identified goal. For example, symbolic execution can be used to extract an appropriate path traversal condition for an identified path. Such enabling conditions are solved by constraint solving techniques. However, current application of static code analysis techniques to generate data is not widespread. Despite much research, these approaches do not scale well, and are problematic for some important code elements, such as loops, arrays and pointers[7].

Recently a new approach to test data generation has emerged: the use of guided search techniques to home in on test data that satisfies identified test criteria. This is discussed in the following section.

II. Background

A. Search Based Test Data Generation

In search based test data generation (SBTDG) achieving a test requirement is modeled as a numerical function optimisation problem and some heuristic is used to solve it. The techniques typically rely on the provision of "guidance" to the search process via feedback from program executions. For example, suppose we seek test data to satisfy the condition $X <= 20$. We can associate with this predicate a cost that measures how close we are to satisfying it, e.g. $cost(X <= 20) = max(X - 20, 0)$. The value $X == 25$ clearly comes closer to satisfying the condition than does $X == 50$, and this is reflected in the lower cost value associated with the former. The problem can be seen as minimising the cost function $cost(X <= 20)$ over the range of possible values of X.

SBTDG for functional testing generally employs a search/optimisation technique with the aim of causing assertions at one or more points in the program to be satisfied. We may require each of a succession of branch predicates to

Kamran Ghani and Prof. John A Clark are with the Department of Computer Science, The University of York, Heslington, York, YO10 5DD, UK (phone: 44-1904-432722; fax: 44-1904-432767; email: kamran, jac@york.ac.uk). Yuan Zhan is with The MathWorks, inc. (email: yuan.zhan@mathworks.com.)

be satisfied (or not) to achieve an identified execution path; we may require the program preconditions to be satisfied but the postconditions to be falsified (i.e. falsification testing — finding test data that breaks the specification) [8]; or else we may simply require a proposed invariant to be falsified (e.g. breaking some safety condition, or causing a function to be exercised outside its precondition)[9].

There has been a growing interest in such techniques and we see more and more applications of these techniques to software testing. Some of the techniques that have been successfully applied to test data generation are Hill Climbing (HC) [10], [11], Simulated Annealing (SA) [12], Genetic Algorithms (GAs) [13], [14], [15], [9], Tabu Search (TS) [6], Ant Colony Optimisation (ACO) [16], Artificial Immune Systems (AIS)[17], Estimation of Distribution Algorithms (EDAs) [18], Scatter Search (SS) [19] and Evolutionary strategies (ESs) [20].

In this work we have compared SA and GA for Simulink models. We used Standard SA, as given in appendix A, for this work and used Genetic and Direct Search Toolbox (GADS) [21] for the GA.

B. Simulink

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

Simulink models have their special way of forming branches compared to programs. Blocks such as 'if-else', and 'Switch' are used to form branches. We have used only 'Switch' blocks in our work for branching structure. However, the work can be easily extended to include other blocks as well. A 'Switch' block has three 'in' ports, one 'out' port and a control parameter 'Threshold'. 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.

Two other important blocks that need to be introduced here are LogicalOperator and RelationalOperator blocks.

A LogicalOperator block has two parameters: operator parameter (which can be 'AND', 'OR', 'NAND', 'NOR', 'XOR', or 'NOT') and input-number parameter (which can be any integer number except that when the operator parameter is 'NOT', in which case, the input-number must be '1').

A RelationalOperator block 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: ==, $\sim=$, $<$, $<=$, $>=$, and $>$. Figure 1. is an example of a Simulink model showing the above mentioned blocks. This model has three input variables; 'IN-A', 'IN-B' and 'IN-C'. The 'Product' block multiplies all its inputs. The model calculates if an equation of the form:

$$ax^2 + bx + c = 0$$

is a quadratic equation and if it has real-valued solution(s). If it is a quadratic equation and it has one or two real-valued roots, '1' is output; otherwise, the output is '-1'

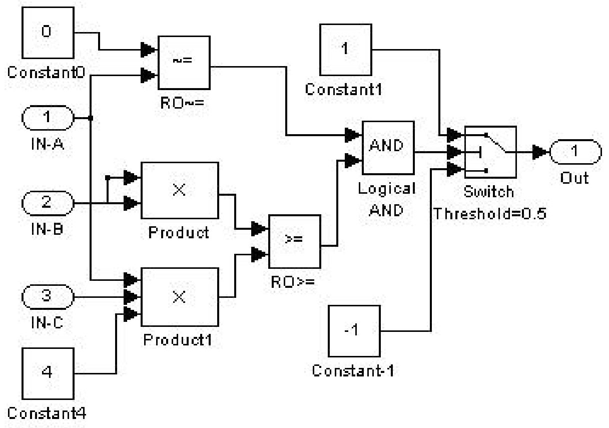

Fig. 1. A Simulink Model for the Quadratic Equation.

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 Figure 2.

```
{

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

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 *'$x \leq y$'*.

C. SBTDG for Simulink Models

Simulink has been popularly used as a higher level system prototyping or design tool in many domains, including aerospace, automobile and electronics systems. This facilitates investigation (e.g. for both verification and validation purposes as well as optimisation) of the system under consideration at an early stage of development. Code can then be

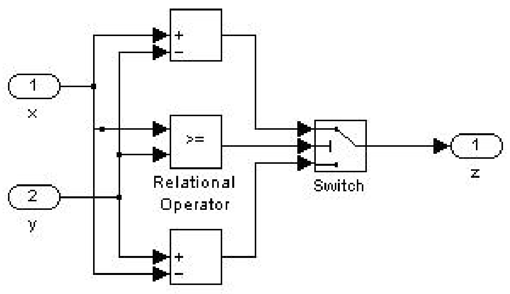

Fig. 2. A Simple Simulink Model

generated either manually or automatically. Simulink plays an increasingly important role in system engineering, and the verification and validation of Simulink models is becoming vital to users. SBTDG techniques have seen little application to Simulink models, which is surprising since the execution model of Simulink would seem to allow analogous SBTDG techniques to be applied as for code. Here we build on earlier work by Zhan and Clark [22].

D. Existing Work

Zhan and Clark [23] proposed search based techniques for generating test data for Simulink models. They successfully generated input test data for structural and mutation testing [24]. They also proposed a technique to address the state problem in search based testing [25].

The search techniques that were applied for test data generation are random testing, which has been the choice for some of the existing tools for Simulink models, and Simulated Annealing. Various models were used for experiments. Results showed that SA was more efficient than random search in finding input test data. The work that we propose in this paper is the extension of the above work. We suggest using GAs for test data generation as our results show that it is more efficient in generating test data for Simulink models than SA.

E. GAs for test data generation of Simulink Models

SA can be a useful technique for generating test data as suggested by existing work [9]. The existing work for Simulink models is in agreement with this. However, there are situations where Simulated Annealing may not be efficient in finding the test data e.g. where the search space is complex and the input domain is very large. In such situations there is a need to apply a more generalized search technique. That is why we see that GAs have been the most common choice in SBTDG techniques.

We believe that GAs will also be more successful in case of Simulink models. However, before making any such claim, we need to have some foundation, either theoretical or empirical. The work presented below is an empirical comparative study of SA and GA. We used the existing prototype tool by Zhan [22] with some modification for SA and incorporated the Genetic Algorithm and Direct Search toolbox (GADS) by The MathWorks, inc. [21] for GA in the same tool.

F. Cost Function

As stated above the main component to guide the search is the fitness function. In the work by Zhan [22], the cost function proposed by Tracey et al. [12] is used with the modification proposed by Bottaci [26]. Table I shows this cost function.

TABLE I
COST FUNCTION

Predicate	Value of Cost Function F
Boolean	if TRUE then 0, else K
$E_1 < E_2$	if $E_1 - E_2 < 0$ then 0, else $E_1 - E_2 + K$
$E_1 \leq E_2$	if $E_1 - E_2 \leq 0$ then 0, else $E_1 - E_2 + K$
$E_1 > E_2$	if $E_2 - E_1 \leq 0$ then 0, else $E_2 - E_1 + K$
$E_1 \geq E_2$	if $E_1 - E_2 \leq 0$ 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) x 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

We have adapted the same cost function for our work.

III. EXPERIMENTATION

We performed two sets of Experiments. In the first set, we considered the 'all-paths-coverage' criterion of models as proposed by Zhan and Clark [23], where a path comprises an identified combination of switch blocks. Fulfillment of the structural adequacy criterion will require a test set to exercise all such combinations of switch predicates. For example, if a model contains two switch blocks S_1 and S_2, then the satisfaction of the above criterion will require finding test data for four 'paths', i.e. $S_1 - true$ $S_2 - true$, $S_1 - true$

$S_2 - false$, $S_1 - false\, S_2 - true$, and $S_1 - false\, S_2 - false$.

The 'all-path-coverage' criterion, however, in most cases, may not be practical, where the number of branching blocks is high. For example a model which contains 15 branching blocks may require to satisfy 32768 such 'paths', which may be computationally very expensive as well as impractical in a reasonable amount of time. Moreover, many such paths may be infeasible. Therefore, in the second set of experiments, we considered a more practical criterion of 'branch coverage' for three different Simulink blocks; *Relational*, *Conditional* and *Switch* [22]. The branch coverage criterion requires all conditional behaviours of the blocks to be executed at least once. For example, a *LogicalOperator* block has two conditional behaviours: being evaluated to 'TRUE' or 'FLASE'. Therefore, in the model in figure 1, where there are four branching blocks (*RO*$\sim$=, *RO*>=, $LogicalAND$ and *Switch*), we have eight such branch coverage requirements.

For the first set of experiments, we used the models from [23]. Whereas for the second set, we used the models from [22]. The properties of the models have been summarised in table II and table III. The models are given in appendix C.

TABLE II

EXPERIMENTAL OBJECTS1

Model	No of Input Var	No of Switch Blocks	No of Paths
SmplSw	2	2	4
Quadratic v1	2	3	8
RandMdl	3	4	16
CombineMdl	5	7	128

TABLE III

EXPERIMENTAL OBJECTS2

Model	No of Input Var	No of Blocks	No of Branches
Tiny	3	4	8
Quadratic v2	3	3	6
ClacStart	3	25	50

A. Experimental Setup

We used the following SA and GA configuration for both set of experiments.

Simulated Annealing Configuration : We used the standard SA algorithm with the following parameters. The parameters values are based on the existing work by [24], where the parameters were optimized after a number of experiments.

- Move strategy: Fixed-strategy with a parameter of 0.02
- Geometric temperature decrease rate: 0.9
- Number of inner loop iterations: 100
- Maximum number of outer loop iterations: 300
- Stopping criterion: Either a solution is found or max number of iterations is reached.

Genetic Algorithm Configuration : We used the GADS toolbox [21], mostly, in its default configuration. Following is a summary of these parameters.

- Initial population size=100 (50 for SmplSW and Quadratic V1).
- Maximum number of generations: 300
- Selection: Stochastic uniform which chooses the parents using roulette wheel and uniform sampling [21].
- Elite count: 2
- Crossover rate: 0.8
- Mutation function: Gaussian which creates the mutated children using Gaussian distribution (scale=.5, shrink=0.75).
- Stopping criterion: Either a solution is found, a stall limit of 100 generations is reached, or the maximum number of generations limit is reached.

B. Analysis

Table II shows the number of paths (combinations) for each model. For analysis we didn't consider trivial combinations. We defined a trivial path or branch as the one for which both the algorithms found required data easily, i.e, SA took less than 100 executions whereas GA took 3 or fewer generations to find the test data. We run each algorithm 30 times for all the models to obtain statistically significant results.

Our hypothesis is that *neither of the algorithms is better than the other*. We tested our hypothesis using success rate and number of executions each algorithm took to find the input test data. Table IV gives the results of experiments.

Both algorithms achieved 100% coverage for all the branches for both experiments. However, when compared for success rate, in experiment 1, GA performed much better than SA. GA achieved a much higher success rate in more complex models. In experiment 2, the performance of both the algorithms was not much different.

In experiment 1, GA required less number of cost function evaluations to find a solution for simple models but SA performed much better than the GA for more complex models. In experiment 2, however, GA performed much better for more complex models.

The last two columns of the table IV gives a comparison of means. i.e. how many times each algorithm was better than the other when their means were compared. The results are very similar for experiment No 1. In case of experiment 2, GA performed much better for the more complex model, Calc-Start-Progress.

The above results, though, give GA an edge over SA, but when compared using the means of the number of cost function executions, we found that the number of paths/branches for which SA performed better than GA was almost the same as the number of paths/branches for which GA performed better. Only in the case of Calc-Start-Progress did the GA outperform SA. Still, however, this doesn't give us much

TABLE IV
RESULTS OF EXPERIMENTS

	Coverage		Mean success rate per 30 runs		Mean No of Fitness Evaluations		Mean Comparison	
	SA	GA	SA	GA	SA	GA	SA	GA
SmplSw	2/2	2/2	30	30	377	133	0	1
Quadratic v1	8/8	8/8	30	28	745	497	4	4
RandMdl	16/16	16/16	15	25	443	675	8	8
CombineMdl	128/128	128/128	17	27	1128	1288	63	63
Quadratic v2	6/6	6/6	30	30	143	169	2	2
Tiny	8/8	8/8	28.5	30	2271	721	2	2
Calc-Start-Progress	50/50	50/50	29.7	30	2236	1799	8	17

information about statistically significant differences between the two algorithms in terms of performance. We further conducted the Mann-Whitney non parametric test for the number of fitness function evaluations. Table V summarizes the results of Mann-Whitney test for both experiments. Column 2 in table V gives the total number of paths or branches considered for analysis. Column 3 gives the number of paths of respective models for which SA performed significantly better than the GA. Column 4 gives the same for the GA. Column 5 gives us the number of branches for which there wasn't a significant difference between the two algorithms.

From the analysis results we can see that GA performed slightly better than SA when compared using the number of executions it took to find test data. However, for a significant amount of time there wasn't any statistically significant difference between the two.

TABLE V
MANN-WHITNEY ANALYSIS OF EXPERIMENTS

Model Analyzed	No of Paths/ Branches Analyzed	SA	GA	No Difference
Smplsw	1	0	1	0
Quadratic v1	8	0	3	5
RandMdl	15	3	3	9
CombineMdl	126	34	42	50
Tiny	4	2	0	2
Quadratic v2	4	2	1	1
Calc-Start-Progress	25	6	6	13

IV. CONCLUSIONS AND FUTURE WORK

In this paper we presented the work for an empirical comparison of simulated annealing and genetic algorithms for Simulink models. To the best of our knowledge, this is the first study of the type for the Simulink models. Surprisingly, we also do not see much work of the kind for code based systems. The only work that came to the best of our knowledge is from Nashat et al [27] and Tracey et al [9]. In [27], results show that SA performed better than GA and, therefore, they suggested using SA for generating test data for path testing. In the work of Tracey et al. SA was more efficient for simpler code segments but performed similar to the GA for more complex programs. However, our results show contrary to this. GA performed slightly better than SA, when compared for the fitness function evaluation. But for the most part, there was not much 'statistically significant' difference between the two. But when it comes to the success rate, the performance of the GA was much better than SA, thus making it a more attractive choice for the test data generation for Simulink models.

Current work does not consider state-based testing. Zhan and Clark [25] also suggested a technique for such testing. They further suggested techniques for search based mutation testing of Simulink models [24]. In our future work we want to extend our work to such models as well.

SBTDG is the most addressed field within search-based software engineering. Indeed, it could be said that SBSE grew out of work in SBTDG. However, we believe that the full potential of SBTDG will only be revealed when its techniques can be routinely applied across system descriptions of varying degrees of abstraction. There has been a great deal of work at the code level, and work generating test data from specifications, but surprisingly little in the middle ground of design. Our work here provides an initial comparison of search approaches to systems expressed in one such design notation and does so with experimental rigour. Longer term, it is clear that a rigorous mapping is needed between system complexity (however measured) and efficacy of the various search techniques. This is a valuable long-term strategic goal and we encourage researchers to address this issue.

APPENDIX

A. Simulated Annealing

Simulated annealing is a global optimization heuristic that is based on the local descent search strategy. The original algorithm was given by Metropolis et al in 1953 [28]. It was 30 years later that Kirkpatrick [29] suggested its application to the optimisation problems. Since then SA has been applied in many forms successfully to solve many problems more efficiently. We applied the standard SA in our work, which is shown below.

```
Select an initial solution $testData_0$;
Select an initial temperature $t_0 > 0$;
Select a temperature reduction function $\alpha$ (=0.9 here);
Repeat
    Repeat
        Generate a move $testData \in N(testData_0)$;
        Where $N(testData_0)$ defines the neighbourhood
        of $testData_0$
        $\delta = f(testData) - f(testData_0)$;
        If $\delta < 0$
            Then $testData_0 = testData$;
        Else
            Generate random $x$ uniformly in the range (0, 1);
                If $x < e^{-\delta/t}$ 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.
```

B. Genetic Algorithm

Genetic Algorithms were developed in 1960s and 1970s by Holland and his associates [30]. These, as evident from the name, are bio-inspired search techniques, which define processes taking inspiration from the principle of natural selection to find near optimal solutions to many computing problems.

In natural selection processes, a species evolves by three main processes, i.e. selection, crossover and mutation over a long period of time. These processes are simulated in GAs. The basic GA algorithm is given as follows:

- Initialise a population of individuals (chromosomes).
- Calculate fitness of each individual in the population (relative to some objective function).
- Select prospective parent (using selection methods).
- Create new individual by mating parents (using crossover).
- Mutate some of the individuals to introduce diversity.
- Evaluate the new members and insert them into the population.
- Repeat stage 2 until some termination condition is reached.
- Return the best individual as the solution.

C. Models used for experiments

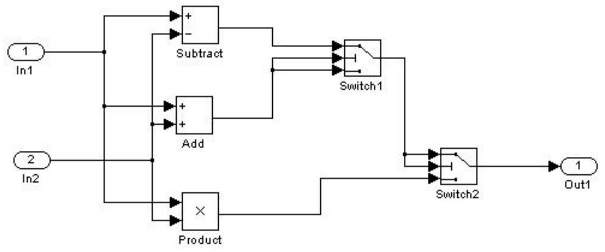

Fig. 3. SmplSW

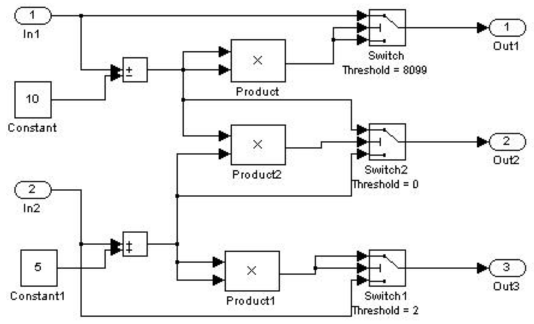

Fig. 4. Quadratic v1

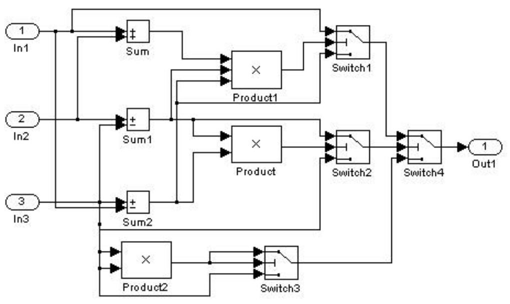

Fig. 5. RandMdl

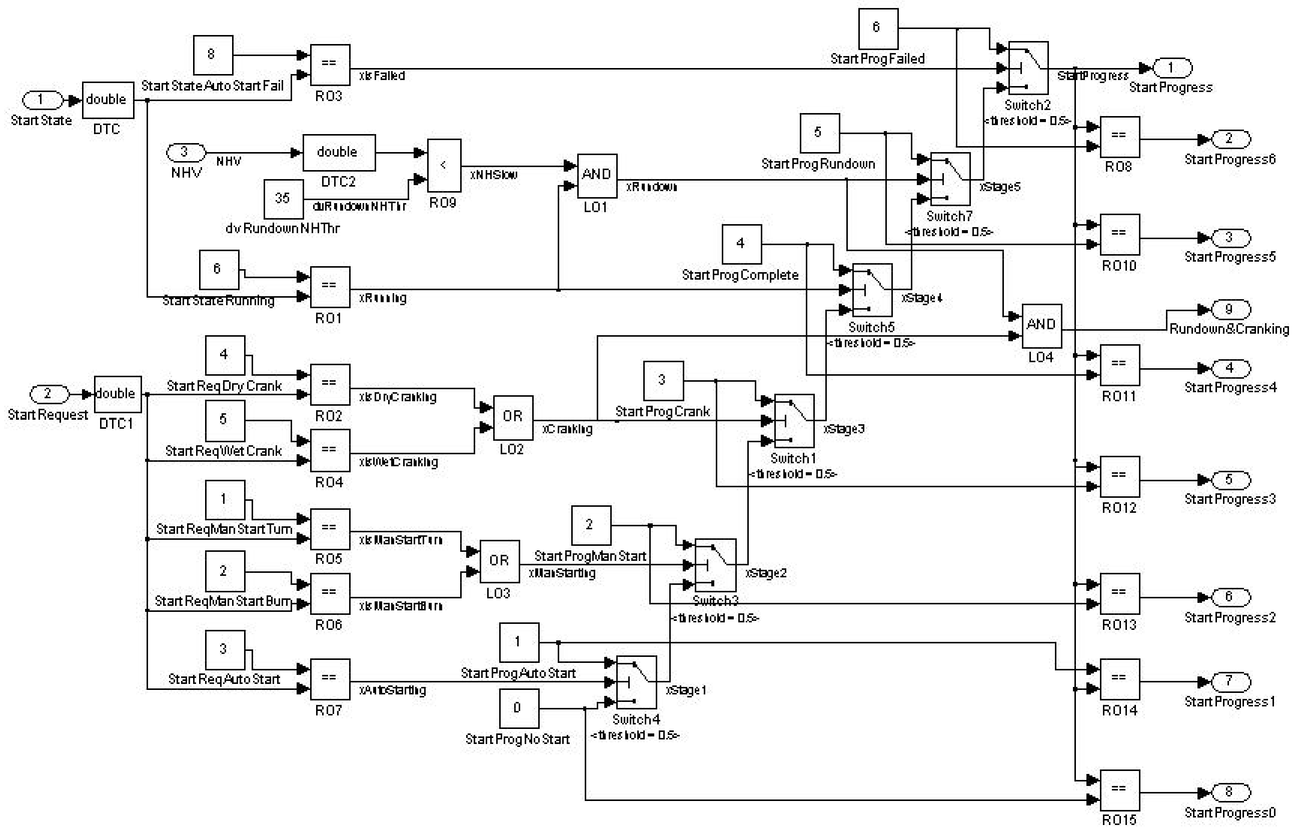
Fig. 9. Calc-Start-Progress

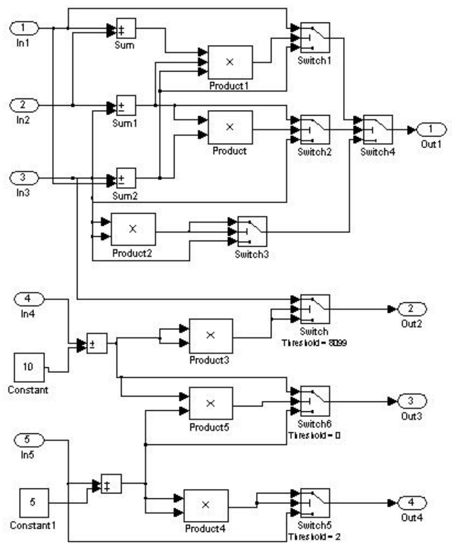
Fig. 6. Combine

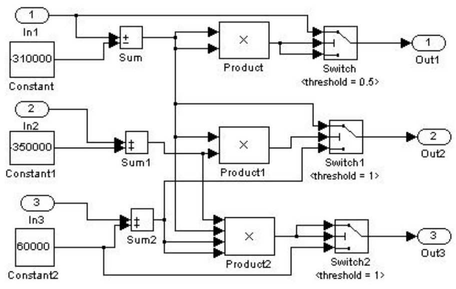
Fig. 7. Quadratic v2

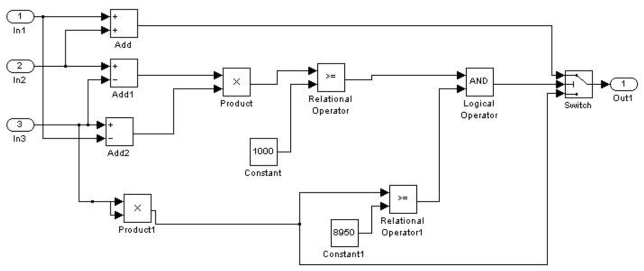
Fig. 8. Tiny

REFERENCES

[1] IEEE, *Guide to software engineering body of knowledge*, ch. 5, p. 1. 2004.

[2] G. Myers, *The Art of Software Testing*. John Wiley & Sons, 1979.

[3] E. Rapps, S. Weyuker, "Test-data generation using genetic algorithms,"

IEEE Transactions of Software Engineering, vol. 11, pp. 367– 375, April 1985.

[4] RTCA, "Software considerations in airborne systems and equipment certification," pp. 2455–2464, 1992.

[5] C. C. Michael, G. McGraw, M. Schatz, and C. C. Walton, "Genetic algorithms for dynamic test data generation," in *Automated Software Engineering*, pp. 307–308, 1997.

[6] E. Diaz, J. Tuya, and R. Blanco, "Automated software testing using a metaheuristic technique based on tabu search," in *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pp. 310 – 313, Oct. 2003.

[7] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[8] K. M. Nigel Tracey, John Clark, "Automated program flaw finding using simulated annealing'," in *proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, (Clearwater Beach, Florida, United States), pp. 73 – 81, 1998.

[9] N. J. Tracey, *A Search-Based Automated Test-data Generation Framework for safety-critical Softwares*. Dphil, University of York, 2000.

[10] W. Miller and D. Spooner, "Automatic generation of floating-point test data," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 223–226, Sept. 1976.

[11] B. Korel, "Automated software test data generation.," *IEEE Trans. Software Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[12] N. Tracey, J. Clark, K. Mander, and J. A. McDermid, "An automated framework for structural test-data generation," in *Automated Software Engineering*, pp. 285–288, 1998.

[13] B. F. Jones, H. H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms," *Software Engineering Journal*, vol. 11, pp. 299–306, Sep 1996.

[14] J. T. Alander, T. Mantere, and P. Turunen, "Genetic algorithm based software testing," in *Artificial Neural Nets and Genetic Algorithms*, (Wien, Austria), pp. 325–328, Springer-Verlag, 1998.

[15] R. Pargas, M. J. Harrold, and R. Peck, "Test-data generation using genetic algorithms," *Journal of Software Testing, Verification and Reliability*, vol. 9, pp. 263–282, Sep 1999.

[16] H. Li and C. P. Lam, "Software test data generation using ant colony optimization.," in *International Conference on Computational Intelligence*, pp. 1–4, 2004.

[17] P. May, K. Mander, and J. Timmis, "Mutation testing: An artificial immune system approach," in *UK-Softest. UK Software Testing Workshop*, (University of York. UK.), September 2003.

[18] J. L. Sagarna, "On the performance of estimation of distribution algorithms applied to software testing.," *Applied Artificial Intelligence.*, vol. 19, no. 5, pp. 457–489., 2005.

[19] J. L. Sagarna, "Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms.," *European Journal of Operational Research*, vol. 169, no. 2, pp. 392–412., 2006.

[20] E. Alba and J. Chicano, "Software testing with evolutionary strategies.," in *The 2nd Workshop on Rapid Integration of Software Engineering Techniques (RISE05)*, vol. 169, (Heraklion, Greece), pp. 50–65., September 2005.

[21] "Genetic algorithm and direct search toolbox, the mathworks inc.." http://www.mathworks.com/products/gads/.

[22] Y. Zhan, *Search Based Test Data Generation for Simulink Models*. PhD thesis, University of York, 2005.

[23] Y. Zhan and J. A. Clark, "Search based automatic test-data generation at an architectural level," in *Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO'04)*, pp. 1413–1424, Springer, 2004.

[24] Y. Zhan and J. A. Clark, "Search-based mutation testing for simulink models," in *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO '05)*, pp. 1061–1068, ACM, 2005.

[25] Y. Zhan and J. A. Clark, "The state problem for test generation in simulink," in *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, pp. 1941–1948, ACM, 2006.

[26] L. Bottaci, "Predicate expression cost functions to guide evolutionary search for test data," in *In Proceeding of the Genetic and Evolutionary Computation Conference (GECCO03)*, pp. 2455–2464, 2003.

[27] M. S. Nashat Mansour, "Date generation for path testing," *sotware Quality Journal*, vol. 12, pp. 121–136, 2004.

[28] M. R. A. T. N. Metropolis, A. Rosenbluth and E. Teller, "Equation of state calculations by fast computing machines," *Journal of Chem. Phys*, vol. 21, pp. 1087–1091, 1953.

[29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.

[30] J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.