IGUANA: Input Generation Using Automated Novel Algorithms. A Plug and Play Research Tool

Phil McMinn
University of Sheffield
Department of Computer Science
Regent Court, 211 Portobello Street,
Sheffield, S1 4DP, UK
p.mcminn@dcs.shef.ac.uk

Abstract

IGUANA is a tool for automatically generating software test data using search-based approaches. Search-based approaches explore the input domain of a program for test data and are guided by a fitness function. The fitness function evaluates input data and measures how suitable it is for a given purpose, for example the execution of a particular statement in a program, or the falsification of an assertion statement.

The IGUANA tool is designed so that researchers can easily compare and contrast different search methods (e.g. random search, hill climbing and genetic algorithms), fitness functions (e.g. for obtaining branch coverage of a program) and program analysis techniques for test data generation.

1 Introduction

In contrast to methods like symbolic execution, search-based techniques take a heuristic approach to the test data generation problem. Symbolic execution works to extract a series of constraints from the program that describe the execution of a particular path. These constraints are solved using linear programming techniques. A search-based approach to this problem instead involves designing a *fitness function* which essentially gives a measure of how close input data were to executing a program structure of interest. The approach is dynamic, and the program is instrumented in order to feedback information to the search algorithm for fitness function computation. In this way, some of the problems associated with symbolic execution, e.g. the handling of loops and dynamic memory, can be circumvented.

There has recently been an explosion of work in the area search-based testing, and in particular the generation of test data. Search-based approaches have been shown to be an effective approach for functional [6, 16, 15], non-functional [20, 14, 21], structural [7, 8, 3, 22, 5, 13, 19, 11, 10], and grey-box [9, 18] testing criteria. McMinn's survey in 2004 [10] cites approximately 40 publications in the area, mainly concentrating on structural test data generation. To date, several search algorithms have been employed, including random search, local search (including hill climbing, tabu search and simulated annealing), evolutionary algorithms (including genetic algorithms, evolution strategies and genetic programming), ant colony optimisation and particle swarm optimisation; with several different approaches to fitness function construction for both structural testing, functional testing, and non-functional properties such as worse case execution time, stress-based testing and so on. Techniques such as testability transformations [4] have been proposed to circumvent certain search-based testing problems, for example plateaux in the fitness landscape.

Until now, there has not been a general framework for comparing different search methods and techniques on the test data generation problem. It is has therefore been difficult to judge results presented in the literature, since different implementations of algorithms have been used, with

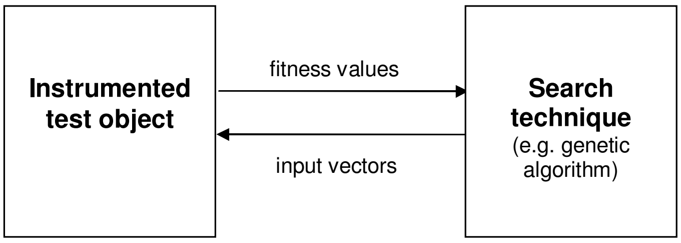


Figure 1: Overview of the process

different settings and test object configurations, leading to results which could be contradictory or misleading. The IGUANA tool attempts to address this problem. It is an object-oriented architecture for test data generation written in Java, incorporating libraries of search algorithms and operators, and standard interfaces for the addition of more. It is possible to 'plug in' different search methods and fitness functions at ease.

2 Search-Based Test Data Generation

At the time of writing, the IGUANA tool has been applied to structural test data generation, although it is easily extensible to other types of testing. Structural testing is by far the most studied area of search-based testing [10]. This section outlines how search-based structural test data generation works, in particular the coverage of individual branches for branch coverage.

In order to cover a particular branch in a unit under test, the goal is to construct an input vector for a function which drives execution of the program down the branch of interest. The search space is formed from the set of possible input vector parameter–value combinations. The test object is instrumented to return fitness information. The search then uses this information to explore promising areas of the program's input domain, which could lead to the discovery of the required test data (Figure 1).

2.1 Fitness Function

For coverage of a branch, the fitness function is calculated by combining a 'branch distance' measure with another metric known as the 'approach level'. The goal of the search is to find the global minimum of the fitness function, i.e. zero.

Approach Level

The approach level calculation comes into effect when there are several conditions that must be satisfied in order to execute the target, for example the true branch from node 8 (Figure 2(a)). It is a measure of how many control dependent nodes were not encountered in the path executed by the input vector. For structured programs, the approach level reflect the levels of nesting surrounding the target (Figure 2(b)). The approach level is referred to as the 'approximation level' by Wegener *et al.*[19].

Branch Distance

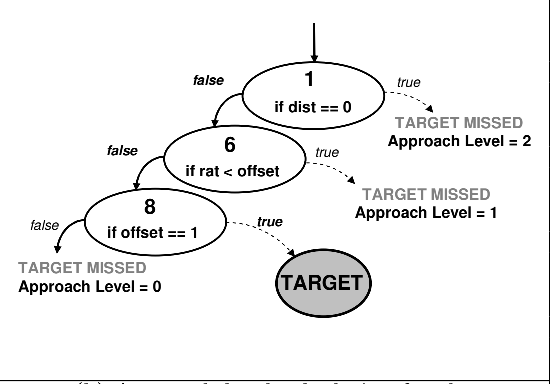
When execution of a test case diverges away from the target branch, the branch distance expresses how close an input came to satisfying the condition of the predicate at which point control flow for the test case went 'wrong'. For example, for the coverage of the true branch from node 1 in Figure 2(a), a predicate distance for '`dist == 0.0`' can be computed using the formula |`dist-0.0`|. The closer `dist` is to zero, the 'closer' the true branch is to being taken. This can be seen in a

```
        double gradient_calc_radial_factor(
                double dist, double offset,
                double x, double y)
        {
            double r, rat;

(1)         if (dist == 0.0) {
(2)             rat = 0.0;
            } else {
(3)             offset = offset / 100.0;
(4)             r   = sqrt (x * x + y * y);
(5)             rat = r / dist;

(6)             if (rat < offset)
(7)                 rat = 0.0;
(8)             else if (offset == 1.0)
(9)                 rat = (rat >= 1.0) ? 1.0 : 0.0;
                // ...
```

(a) Code snippet



(b) Approach level calculation for the true branch from node 8

Figure 2: Code from the gimp open source graphics package and corresponding fitness analysis

plot of the fitness landscape Figure 2(c). Predicate distance calculations for different types of inequalities can be found in Table 1.

The situation becomes more complicated with conjoined or disjoined predicates, especially where the target language employs short-circuiting (where short-circuiting is not employed the rules of Tracey *et al.*[17] can be applied). In this case, evaluation breaks off early at any predicate where the result of the entire condition has been decided. The predicate distance is found for the last predicate evaluated, with the number of unevaluated conditions also figuring in the overall fitness calculation.

Overall Fitness Calculation

Many search techniques just require the comparison of fitness values. Therefore one numerical value, including the branch distance and approach level components is not required - just the knowledge that one candidate solution is 'better' than another. For this comparison, a lower approach level corresponds to a better solution. If there is a tie, then it is a lower number of unencountered predicates in the condition is preferred, and if there is a tie in the number of unencountered predicates, then a lower predicate distance corresponds to the better solution.

However where one numerical value is required, the predicate distance is normalized according to the following equation of Baresel [2], where d is the predicate distance:

$$n = 1 - 1.001^{-d} \tag{1}$$

The branch distance, b is then calculated as:

$$b = \frac{u + n}{t} \tag{2}$$

where u is the number of unencountered predicates, n is the normalized predicate distance, and t is the total number of predicates in the branch condition.
This is then added to the approach level a to form an overall fitness value f:

$$f = a + b \tag{3}$$

2.2 Search Algorithms

The following sections review some common search algorithms used for search-based test data generation.

Table 1: Distance calculations for relational predicates (from Tracey *et al.*[17]). The value K, $K > 0$, refers to a constant which is always added if the term is not true

Relational Predicate	Objective Function *obj*
Boolean	if $TRUE$ then 0 else K
$a = b$	if $abs(a - b) = 0$ then 0 else $abs(a - b) + K$
$a \neq b$	if $abs(a - b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + K$
$\neg a$	Negation is moved inwards and propagated over a

```
          void all_zeros(int list[], int size) {
              int total = 0, i;

(1)           for (i=0; i < size; i++) {
(2)               if (list[i] == 0) {
(3)                   total ++;
                  }
              }

(4)           if (total == size) {
(5)               printf("All zeros \n");
              }
          }
```

Figure 3: The `all_zeros` function

2.2.1 Genetic Algorithms

Genetic algorithms (GAs) belong to the family of evolutionary algorithms, which work to evolve superior candidate solutions (known as *individuals*) using mechanisms inspired by genetics, natural selection and survival of the fittest. The search evolves several individuals at once in a 'population'.

GAs generally make use of binary representations, however real-valued encodings are depicted throughout this paper, as they are more commonly used for test data generation. The 'chromosome' making up each individual is more or less a direct representation of the input vector to the program concerned. The `all_zeros` program of Figure 3, for example, would have the chromosome `<list[0], ... list[4], size>`, where the array size of `list` is fixed at 5. As can be seen, the 'genes' of the chromosome represent the input values that the program will be executed with.

A distinguishing feature of GAs over other forms of evolutionary algorithm is the emphasis they place on crossover. Crossover is a mechanism of exchanging genetic material between individuals, with the aim of breeding new, potentially 'fitter' offspring. A series of 'crossover points' are used to decide where two parent chromosomes are to be spliced in order to form the composite chromosomes of two children. The example below shows two input vectors to the `all_zeros` program being crossed over at position 3 to produce two offspring.

Parents

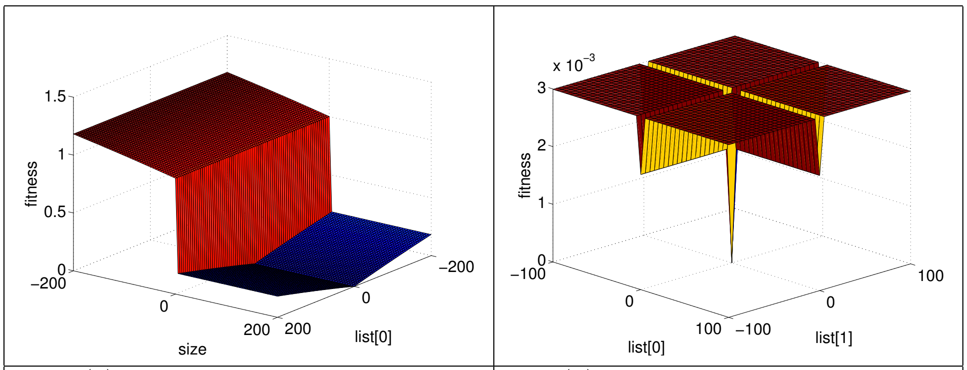
`list[0]`	`list[1]`	`list[2]`	`list[3]`	`list[4]`	`size`
0	*0*	*0*	*20*	*20*	*4*
20	20	20	0	0	5

→

Offspring

`list[0]`	`list[1]`	`list[2]`	`list[3]`	`list[4]`	`size`
0	*0*	*0*	0	0	5
20	20	20	*20*	*20*	*4*

A GA is essentially a loop of operations on the population of individuals. Each iteration of the population is known as a 'generation'. The aim is to evolve generations containing generating fitter input vectors. The steps of each iteration, comprise *selection* of individuals for *crossover*, *mutation*, and *reinsertion* of individuals into the population for the next generation. At selected generations sub-populations exchange individuals (*migration*), and compete for resources (*competition*), as introduced above. The search continues until test data has been found, or resources have been exhausted.



(a) True branch from node 2 (b) True branch from node 4

Figure 4: Fitness landscapes the all_zeros function, showing two inputs

Selection. Selection is the process of selecting parents for crossover. A selection strategy is generally biased towards the best individuals, but weaker individuals are selected as well in order to keep the population diverse, and to prevent premature convergence on sub-optimal areas of the search space. One method is *stochastic universal sampling* [1], whereby the probability of an individual being selected for reproduction is proportionate to its fitness. In order to avoid a situation where a few very fit individuals are selected often, and dominate the search, 'ranked' fitness values are used. Ranked values depend on the individual's position in the overall population sorted into fitness order.

Crossover. Once the selection pool has been decided, parents are taken two at a time for crossover. There are several operators available for crossover, where the individual is spliced at one location (one-point crossover), multiple locations (multi-point crossover) or at all possible locations (uniform crossover). Another method is *discrete recombination* [12] to generate offspring. Discrete recombination is similar to *uniform crossover*, in that every position in the chromosome is a potential crossover point. However, a gene can be copied into one or both children with an even probability.

Mutation. The offspring are then mutated. For binary encodings this involves flipping bits at low probability. For real-valued encodings, potential operators include replacement of a gene with a newly randomly generated value, or gaussian mutation, where a new value is selected using a gaussian distribution around the current value.

Reinsertion. The next generation is then constructed using the existing generation and the new offspring. An elitist strategy to reinsertion, for example, replaces the worst of the current generation with the best offspring.

2.2.2 Hill Climbing

Hill climbing is a very simple local search algorithm which works to improve a single candidate solution, starting from a random point in the search space. The method simply explores the neighbouring search space around the current 'point'. If a fitter candidate solution is found, the search moves to that point. If no better solution is found in the neighbourhood, the algorithm terminates. The method is called 'hill climbing', because the process is likened to the climbing of hills on the surface of the fitness function. However, a 'climb', or an improvement in fitness, is represented by acceptance of a decreased numerical value, since the fitness function is to be minimized.

The coverage of the true branch from node 2, therefore, is represented by the valley touching zero on the z axis of the fitness function surface (Figure 4a).

A form of hill climbing, referred to as the *alternating variable method*, was used by Korel in early papers in the search-based test data generation literature [7, 8]. The idea of the approach is to take each input variable in turn and adjust its value in isolation from the rest of the vector. If altering the variable does not result in better fitness, the next input variable is targeted, and so on, until no modification of input values results in an improved fitness. If altering the variable does result in an improvement in fitness, accelerated moves are made in the direction of improvement by increasing the numerical step size of the move.

A well-known problem with local search methods like hill climbing is that they can easily land at the base of sub-optimal 'hills', and fail to find a good solution. Once a hill has been climbed, or a plateau encountered, the rest of the search space remains unexplored. A GA tends to be more robust in such landscapes, sampling many points in the search space at once.

By way of example, recall the `all_zeros` function from Figure 3 and assume `size` is fixed and not alterable by the test data search. The relationship between inputs and values for node 4's predicate is not as simple and direct as that for node 2. For the most part, exploratory moves for values of `list` have no effect on `total`, resulting in areas of undistinguished fitness. On encountering one of these plateaux or valleys, as seen in Figure 4b, hill climbing terminates without finding the required test data.

In general, it could be that the hill climb is successful if it is restarted in another (randomly selected) area of the search space, and therefore this strategy is often employed until the search has run out of resources.

3 IGUANA: An Overview

IGUANA is essentially a set of libraries for search-based test data generation. It is written in Java and currently supports structural test data generation for C code. At the time of writing, IGUANA consists of approximately 450 classes. IGUANA contains a library of search algorithms, including genetic algorithms and their various operators. It also contains a C parser for parsing and instrumenting C code. The C code to be tested is compiled into a DLL which is executed by IGUANA code via the Java Native Interface. The instrumentation inserted into the code takes the form of modifying the conditions in branch predicates to call fitness computation code in IGUANA.

3.1 Overview of the Java packages in IGUANA

IGUANA is made up of the following Java packages:

- cparser

  This library parses C code, automatically instrumenting it with callbacks so that the fitness function can be computed. The C parser was generated by the javacc tool using an adapted C grammar. The control flow and dependence graphs are extracted from each function. The control dependence graph is used in computation of approach levels. The control flow graph is used in order to automatically determine the test targets of a coverage type, for example all branches or all statements.

- expt

  This library contains bootstraps for starting experiments. Sub-libraries are automatically generated with classes for each test object. It is within these classes that the user specifies input type information for each function to be tested.

- inputgeneration

  This library contains the 'generator' classes, which are wrappers around search algorithm classes but with all the parameters specified (for example which mutation operator to use, if it is a genetic algorithm). It also contains objective function classes for different types of fitness function for test data generation. The Trace class is used to monitor the path taken through a C function by a particular input vector, and keeps track of the various predicate distances found along the way.

- log

  The log library is made up of classes for logging the progress of test data generation runs.

- programstructure

  This library provides classes which model a program. These classes are instantiated upon parsing by the cparser library. Classes include CFG, which models a control flow graph and CFGNode, for control flow graph nodes. ControlDependency models control dependencies, from which the control dependency graph can be recovered. The sub-library condition models the conditions of a branching statement, with classes such as AndCondition, AtomicCondition etc.

- search

  The search library contains sub-libraries and classes for the various search methods supported. Individuals, or candidate solutions, are modelled in the solution sub-library, which contains classes for the different types of encoding - for example as a vector of real values. The evolve library is the genetic algorithm library. The library supports multiple populations, with competition and migration amongst them. Mutation operators currently implemented include Gaussian mutation, Breeder genetic algorithm mutation [12], and uniform mutation. One point, uniform and discrete recombination crossover operators are implemented in the library. Several selection methods are also supported, including elitest selection, random selection, stochastic universal sampling and tournament selection. The random library supports random search, whilst the hillclimb library supports basic hill climbing and the alternating variable method, as described in the last section.

- testobject

  Finally, the testobject library contains base classes for representation of a test object, which are inherited by the concrete classes auto-generated for the expt library.

3.2 Directory Structure

There are three main top level directories in IGUANA; classes and src, where the compiled Java class and source files respectively live, and ctestobj where the source and DLL files for the test objects (written in C) live.

Within the ctestobj directory, the sub-directory include, contains C code for wrapping up the test object into a form which IGUANA requires; including Java native interface headers and functions. These are called by the instrumented C code which in turn call methods in IGUANA Java code for fitness computation. The subdirectory lib is where the library of compiled DLLs are stored. The src directory contains source C code for each test object. Each test object has a sub-directory within src. Within this directory, the instrumented C code is stored, along with generated reference image files for the control flow and dependence graphs, and a directory called structure which contains control flow and branching condition information in serialised Java object form. The *.struc files in this directory correspond to each function in the test object and are essentially serialised iguana.programstructure.CFG objects.

4 Preparing a Test Object for Test Data Generation

This section outlines the steps involved in preparing test objects and initiating test data generation with IGUANA.

4.1 Setup Environmental Variables

Initially, the IGUANA_HOME variable needs to be set to the directory in which IGUANA resides, and the PATH variable needs to be set to point to the ctestobj/lib IGUANA directory.

4.2 Run Instrumenter

Firstly the test object C source (called TESTOBJECTNAME.c) needs to be placed in the directory ctestobj/src/TESTOBJECTNAME. The command java iguana.testobject.MakeCTestObject -p TESTOBJECTNAME will then prepare the test object, instrumenting it and wrapping it in Java Native Interface headers. Java classes will also be automatically created to provide IGUANA with a handle to the test object and as a basis for definition of the search vector.

4.3 Define Search Vector

The search vector is the raw sequence of double variables to be optimised by the search algorithm. It is essentially a numerical model of the input domain of the function under test. The search vector for a function to be tested is defined by an 'input specification'.

A package will have been created in the IGUANA source directory called iguana.expt.TESTOBJECTNAME, with a FUNCTIONTOTEST.java class for testing of each C function in the test object. A 'loader' class, called TESTOBJECTNAME_Loader.java will also be created. Loader classes instantiate each FUNCTIONTOTEST object for a test object, and return the objects in a list. The code of each FUNCTIONTOTEST.java file needs editing to initialise an instance variable called inputSpec, of class InputSpecification. Using the methods of InputSpecification, the input specification needs to be assembled, stating the minimum and maximum of each variable, and its accuracy (i.e. the number of decimal places it invovles).

The check_ISSN test object of bibclean takes an array of 30 characters. Thus the search vector is defined with the simple line 'inputSpec.addInt(30, -128, 127);' - i.e. 30 variables long, in the range for the C char type (that is, -128 to 127 of integer form - i.e. no decimal places).

4.4 Define Mapping from Search Vector to Input Parameters

The C file iguana_expt_TESTOBJECTNAME_FUNCTIONTOTEST.c, in the instrumented directory then needs to be edited. The perform_call function, contained within, needs to be edited to map the search vector from IGUANA (an array of double variables) into parameters with which to execute the function under test. The following is an example for the check_ISSN function of bibclean. Essentially the array of doubles is recast to an array of characters.

```
void perform_call(double* args, int num_args)
{
 // declarations
 const int NUM_ARGS = 30;
 int i;

 // check correct number of arguments
 if (num_args != NUM_ARGS) {
   native_error("Wrong number of generated inputs for check_ISSN");
 }

 for (i=0; i < NUM_ARGS; i++) {
   current_value[i] = (char) args[i];
 }

 // test object calling code
 check_ISSN();
}
```

4.5 Create DLL

The command java iguana.testobject.MakeCTestObject -c TESTOBJECTNAME can now be run to compile all the functions to test object DLLs.

5 Performing Test Data Generation

In order to perform test data generation, one of the 'generator' classes in the inputgeneration.generator package must be invoked. The generators are essentially bootstrap classes which initialise a search method with a set of parameters, load up the required test object and appropriate fitness function and begin the search process. The output is currently handled by a series of classes in the log package (currently, information is dumped to the screen or redirected to a text file).

6 Defining New Search Techniques and Fitness Functions

A new search technique simply needs to extend the Search abstract class, found in the search package. Parameters to the specifics of the search can be passed in via a newly defined constructor or factory method. Candidate solutions used by the search should extend the search.solution.Solution class.

Fitness functions should implement the search.objective.ObjectiveFunction interface, with fitness values extending the search.objective.ObjectiveValue class. This class has the method isIdeal which returns a Boolean value denoting whether the fitness value represents the global optima. Fitness 'values' need not be represented internally in a numerical format (the class extends java.lang.Comparable for comparison purposes), but as most search methods do require this, the method getNumericalValue should be overridden to return a numerical representation.

Fitness functions for automatic test data generation are found in the inputgeneration.objectivefunction package. This is where code for calculating predicate distances resides. The inputgeneration.trace package contains classes for tracing the execution path taken through test objects and collating branch distances.

7 Test Object Library

Test objects that have been used with IGUANA to date can be seen in Table 2.

`bibclean-2.08` is an open source program used to syntax check and pretty-print BibTeX bibliography files. The two functions tested are validity checks for ISBN and ISSN codes used to identify publications. `eurocheck-0.1.0` is also an open source program. It contains a single function used to validate serial numbers on European bank notes. `gimp-2.2.4` is the open source GNU image manipulation program. Several library functions were tested, including routines for conversion of different colour representations (for example RGB to HSV) and the manipulation of drawable objects. `space` is program from the European Space Agency used for scanning star field patterns, of which nine functions were tested. `spice` is an open source general purpose analogue circuit simulator. Two functions were tested, which were clipping routines for the graphical front-end. `tiff-3.8.2` is a library for manipulating images in the Tag Image File Format (TIFF). Functions tested include image placing routines and functions for building 'overview' compressed sample images.

8 Summary

This paper has presented an overview of search-based structural test data generation with the IGUANA system. The IGUANA system is a research tool and is thus under constant adaption to incorporate new ideas and algorithms.

Table 2: Test object details

Test Object / Function	Lines of of Code	Number of Branches
bibclean-2.08		
check_ISSN		42
check_ISBN		42
Total	178	
eurocheck-0.1.0		
main		22
Total	70	
gimp-2.2.4		
gimp_rgb_to_hsl_int		14
gimp_rgb_to_hsv		10
gimp_hsv_to_rgb		16
gimp_hsv_to_rgb_int		16
gimp_rgb_to_hsv_int		14
gimp_rgb_to_hsl		14
gimp_rgb_to_hsv4		18
gimp_hwb_to_rgb		18
gimp_hsv_to_rgb4		16
gradient_calc_radial_factor		6
gradient_calc_square_factor		6
gradient_calc_conical_sym_factor		8
gradient_calc_conical_asym_factor		6
gradient_calc_bilinear_factor		6
gradient_calc_spiral_factor		8
gradient_calc_linear_factor		8
Total	867	
space		
addscan		32
fixgramp		8
fixport		6
fixselem		8
fixsgrel		68
fixsgrid		22
gnodfind		4
seqrotrg		32
sgrpha2n		16
Total	2210	
spice		
cliparc		64
clip_to_circle		42
Total	269	
tiff-3.8.2		
TIFF_SetSample		14
TIFF_GetSourceSamples		18
PlaceImage		16
Total	182	

References

[1] Baker, J. E. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the 2nd International Conference on Genetic Algorithms and their Application* (Hillsdale, New Jersey, USA, 1987), Lawrence Erlbaum Associates.

[2] Baresel, A. Automatisierung von strukturtests mit evolutionren algorithmen. Diploma Thesis, Humboldt University, Berlin, Germany, July 2000.

[3] Ferguson, R., and Korel, B. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology 5*, 1 (1996), 63–86.

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

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

[6] Jones, B., Sthamer, H., Yang, X., and Eyres, D. The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of the 3rd International Conference on Software Quality Management* (Seville, Spain, 1995), pp. 435–444.

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

[8] Korel, B. Dynamic method for software test data generation. *Software Testing, Verification and Reliability 2*, 4 (1992), 203–213.

[9] Korel, B., and Al-Yami, A. M. Assertion-oriented automated test data generation. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)* (1996), pp. 71–80.

[10] McMinn, P. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability 14*, 2 (2004), 105–156.

[11] McMinn, P., and Holcombe, M. Hybridizing evolutionary testing with the chaining approach. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004), Lecture Notes in Computer Science vol. 3103* (Seattle, USA, 2004), Springer-Verlag, pp. 1363–1374.

[12] Mühlenbein, H., and Schlierkamp-Voosen, D. Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. *Evolutionary Computation 1*, 1 (1993), 25–49.

[13] Pargas, R., Harrold, M., and Peck, R. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability 9*, 4 (1999), 263–282.

[14] Puschner, P., and Nossal, R. Testing the results of static worst-case execution-time analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium* (Madrid, Spain, 1998), IEEE Computer Society Press, pp. 134–143.

[15] Tracey, N. *A Search-Based Automated Test-Data Generation Framework for Safety Critical Software.* PhD thesis, University of York, 2000.

[16] Tracey, N., Clark, J., and Mander, K. Automated program flaw finding using simulated annealing. In *Software Engineering Notes, Issue 23, No. 2, Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1998)* (1998), pp. 73–81.

[17] Tracey, N., Clark, J., Mander, K., and McDermid, J. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering* (Hawaii, USA, 1998), IEEE Computer Society Press, pp. 285–288.

[18] Tracey, N., Clark, J., Mander, K., and McDermid, J. Automated test data generation for exception conditions. *Software - Practice and Experience 30*, 1 (2000), 61–79.

[19] Wegener, J., Baresel, A., and Sthamer, H. Evolutionary test environment for automatic structural testing. *Information and Software Technology 43*, 14 (2001), 841–854.

[20] Wegener, J., Grimm, K., Grochtmann, M., Sthamer, H., and Jones, B. Systematic testing of real-time systems. In *Proceedings of the 4th European Conference on Software Testing, Analysis and Review (EuroSTAR 1996)* (Amsterdam, Netherlands, 1996).

[21] Wegener, J., and Grochtmann, M. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems 15*, 3 (1998), 275–298.

[22] Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Katsikas, S., and Karapoulios, K. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In *5th International Conference on Software Engineering and its Applications* (Toulouse, France, 1992), pp. 625–636.