Transactions on Information and Communications Technologies vol 11, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

The automatic generation of software test data sets using adaptive search techniques

B.F. Jones,[a] H.-H. Sthamer,[b] X. Yang,[a] D.E. Eyres[a]
[a] *Department of Computer Studies,* [b] *Department of Electronics and Information Technology, University of Glamorgan, Pontypridd, Mid Glamorgan, CF37 1DL, UK*
Email: bfjones@glamorgan.ac.uk

Abstract

Test sets which cover all branches of a library of five procedures which solve the triangle problem, have been produced automatically using genetic algorithms. The tests are derived from both the structure of the software and its formal specification in Z. In a wider context, more complex procedures such as a binary search and a generic quicksort have also been tested automatically from the structure of the software. The value of genetic algorithms lies in their ability to handle input data which may be of a complex data structure, and to execute branches whose predicate may be a complicated and unknown function of the input data. A disadvantage of genetic algorithms may be the computational effort required to reach a solution.

1. Introduction

Many approaches have been used to automate the generation of test sets for software. Random testing has been investigated thoroughly by Duran and Ntafos [8] and Hamlet and Taylor[9]. Deterministic heuristics have also been used notably by Korel [10]. Unfortunately, none of these prototypes has been widely adopted in testing because of the difficulty of automating the testing of complex software. The purpose of this investigation is to assess the usefulness of applying genetic algorithms [4] to the problem of automatic test generation. Genetic algorithms are a search technique which is often successful when deterministic heuristics fail. They may be applied to test generation in the sense that deriving tests may be likened to searching the input domain for data which will exercise a particular branch or path through the software, or a particular aspect of functionality of the specification. We have investigated the derivation of tests from both the structure of the software (*white-box* testing) and from the formal Z specification (*black-box* or *functional* testing).

2. The Z specification of a triangle system

Z is a model-based, formal specification language which was developed at the Oxford Programming Research group in the 1980s in collaboration with IBM UK (Spivey [1], Wordsworth [2], McMorran and Nicholls [3]). It is based on set theory and uses mathematical schemas interspersed with English statements to describe the properties of a software system precisely and unambiguously.

As a simple example of deriving tests from a Z specification, we describe a triangle system which determines whether three integer values form a valid triangle, which may be isosceles, equilateral or right-angled. First, the given sets which are manipulated by the system are defined, along with global variables and constraints on their values:

[INPUT] is the set of three integer values.
[MESSAGE] :: = Invalid | Equilateral | Isosceles | Scalene | Rightangle

The state space of the system is described in a schema which is split into two sections; it may be written horizontally, or (more usually) vertically. We adopt the vertical notation. The upper section is a collection of variables which define the state of the system, and the lower section describes state invariants.

```
__________Triangle0______
|
| x?,y?,z? : Z
|______________
| x? >0  ∧  x? < 200
| y? >0  ∧  y? < 200
| z? >0  ∧  z? < 200
| x? + y? > z?
| y? + z? > x?
| z? + x? > y?
|______________________
```

The state space schema declares three input variables x?, y?, and z? belonging to the set of integers, Z, and invariant relationships such as that each input must be strictly positive and less than a defined maximum value.

Further schemas describe operations belonging to the system; some operations may seek to change the system state whereas others may only query the state leaving it unchanged. We define four which define operations to determine whether the triangles are scalene

```
_____ScalTri________
|
| Ξ Triangle0
| reply! : MESSAGE
|________________
|
|  x? ≠ y?
|  y? ≠ z?
|  z? ≠ x?
|  reply! = Scalene
|__________________
```

or equilateral

```
_____EquiTri_________
|
| Ξ Triangle0
| reply! : MESSAGE
|________________
|
| x? = y?
| y? = z?
| reply! = Equilateral
|
|__________________
```

or isosceles

```
_____IsosTri________
|
| Ξ Triangle0
| reply! : MESSAGE
|__________________
|
| (x? =y?  ∧  y? ≠ z?) ∨
| (y? =z?  ∧  z? ≠ x?) ∨
| (z? =x? ∧  x? ≠ y?)
| reply! = Isosceles
|____________________
```

or right angled

```
______RightTri________
|
| Ξ Triangle0
| reply! : MESSAGE
|__________________
|
| (x? * x? + y? * y? = z? * z?) ∨
| (y? * y? + z? * z? = x? * x?) ∨
| (z? * z? + x? * x? = y? * y?)
| reply! = Rightangle
|____________________
```

Schemas may also define inherent or representation errors, such as

```
____NumError________________
| x?, y?, z? : INPUT
| reply! : MESSAGE
|________________
|
| (x?∉Z  ∨  x?>200  ∨  x? <= 0)  ∨
| (y?∉Z  ∨  y?>200  ∨  y? <= 0)  ∨
| (z?∉Z  ∨  z?>200  ∨  z? <= 0)
| reply! = Invalid
|________________________________
```

and

```
__________TriangleError______
| x?, y?, z? : INPUT
| reply! : MESSAGE
|________________
| (x?+ y? <= z?) ∨
| (y? + z? <= x?) ∨
| (z? + x? <= y?)
| reply! = Invalid
|________________________________
```

3. The application of Genetic Algorithms to test generation

Genetic Algorithms (GAs) were described by Holland [4], and they have been used with much success to solve non-linear optimisation problems by searching the solution space for the best data [6]. We have applied GAs to generating test sets (Sthamer, Jones and Eyres [5]) by searching the input domain for test data which ensure that each branch of the code is exercised.

The input parameters to a problem are represented typically by a binary code. This is straightforward for the input data to a computer program where the machine memory image can be used. For example, in our triangle problem, there are three integer inputs, x?, y?, z?, and each will be represented by 32 bits on a DEC Alpha processor. The three combined inputs are thus represented by a string of 96 bits. Normally the GAs work on a square array of bits, so that 96 random guesses are made for the input data x?, y?, z?. The input domain is thus sampled at random for possible solutions; here, the meaning of the term *solution*

Transactions on Information and Communications Technologies vol 11, © 1995 WIT Press, www.witpress.com, ISSN 1743-3517

must be clarified. In the context of structural testing, the aim may be to exercise a branch
if A = B then....
where A and B are functions of x?, y? and z?. A critical part of using GAs is to specify a *fitness function* which is a measure of how close a guessed test set is to the goal. The goal in this case is to find a combination of x?, y? and z? which ensure that the values of A and B are equal. A convenient way of establishing a fitness for this case is to use the reciprocal of the Hamming distance between the binary patterns corresponding to A and B. The fitness is established for each of the 96 combinations of x?, y?, and z?. One of the strengths of this approach is that A and B may be complex functions of x?, y?, and z?, but this does not increase the complexity of the method since the fitness only depends on the actual values of A and B at this point in the program.

The GAs then go through a process of mixing bit patterns from two guesses which may have been chosen because they have a high fitness. An arbitrary point along the bit string is chosen at random, and the tails of the two guesses are exchanged, thus producing two offspring. This process is known as *crossover*. In addition, there is a small probability that a bit will be *mutated* ie flipped from one binary state to the other. The fitnesses of the offspring are calculated. The next generation of guesses is chosen from the original population and the offspring. One strategy is to allow the fittest to survive (cf Darwinian evolution), though allowing some guesses with a poor fitness to join the new generation prevents stagnation at a sub-optimal solution.

GAs have been applied to generating tests for a wide variety of programs by attempting to exercise every branch in the code (*structural testing* [5]). In this case the selections in the code form the basis for the fitness function. The code is instrumented with calls to procedures which record the passage of control through each branch and calculate the fitness at that point.

Each node in control the flow tree is exercised in turn, and the fitness of the population of guesses increases in a saw tooth like pattern where the peaks correspond to a successful execution of the branch under investigation, and the troughs to the start of the search for a test to satisfy the next node.

Formal specifications have been used to define the use or changes in the state space of the system (*functional* and *state-based testing*). In this case, the predicates in the Z specification form the basis of the fitness function.

4. Results of structural testing

The Z specification described above has been implemented in Ada83; there are five procedures which
1. check that the three sides of the triangle are positive (Triangle0 schema),

2. check for a scalene triangle (ScalTri schema),
3. check for an isosceles triangle (IsosTri schema),
4. check for an equilateral triangle (EquiTri schema),
5. check for a right-angled triangle (RightTri schema).

The control flow tree for the combination of these five procedures contains a total of 26 branches to be executed. When the input range of x?, y? and z? are restricted to be between ±100, complete branch coverage is obtained in all test runs, and demands a total of 18,800 tests. As a bench mark, this may be compared with pure random testing for which complete branch coverage is frequently not achieved, and at least 163,300 tests are needed. Most of the predicates are linear combinations of x?, y? and z? and are satisfied relatively easily. There are three non-linear tests for a right-angled triangle

$$x?^2+y?^2-z?^2=0 \qquad x?^2+z?^2-y?^2=0 \qquad z?^2+y?^2-x?^2=0.$$

There are only 104 combinations of inputs out of a possible 8 million which satisfy this predicate, and consequently have a small probability of occurring at random. Additionally, there are successive predicates which check that x?=y? and y?=z? which are equally difficult to satisfy. It is these circumstances which give GAs the advantage over random testing, and over the effort required to derive inputs manually to satisfy complex predicates, though the latter predicates can be satisfied simply by deterministic heuristics (Holmes, Jones and Eyres [7]).

The real power of GAs is evident when predicates which are complex functions of many input parameters need to be tested and when the input parameters are combination of data structures rather than primitive data types. GAs have been used to cover the structures of a wide range of procedures including a *remainder* calculation which has four while loops, a *binary search* procedure and a *generic sort* which several nested loops and which has been tested with arrays of records whose fields are characters and integers (Sthamer, Jones and Eyres [5]). In all of these cases, full branch coverage is achieved. In procedures containing iterations, the number of iterations is controlled to be zero, one, two or any pre-defined number. This seems to be a good strategy, based intuitively on proof by induction, i.e. if valid outputs are achieved for zero, one and two iterations then there is confidence that the iteration has been coded correctly.

The experiments with GAs indicate that the most successful strategy uses a *uniform crossover* strategy which exchanges each pair of bits in the two parent bit strings with a probability of 0.5, rather than picking a single point in the string and exchanging the tails. The fitness function is based on the Hamming distance of the guessed solution from the goal; although the Hamming distance does not give much improved results over a simple reciprocal function for numerically based predicates, the Hamming distance can be applied to the range of data structures involved in branch decisions.

5. Results of tests based on Z specifications

Our work on deriving tests from Z specifications is less advanced than our work on structural testing. So far we have concentrated on the triangle system which was specified earlier.

The Z specification language uses many symbols which cannot be input directly into a computer, and when a Z specification is entered for analysis by the automated testing system, the mappings shown in Table 1 are used. The Z symbols are taken from the Z User Manual prepared by the IBM Hursley Laboratories [3].

Name	Input String	Z Symbol
integers	&Int	**Z**
quotient	&div	÷
	&leq	<=
comparison	&neq	≠
	&geq	>=
and (conjunction)	&and	∧
or (disjunction)	&or	∨

Table 1: Mapping from Z symbols to input strings for parsing

The triangle problem may be rewritten as
Triangle ::= (Triangle0 ∧ EquiTri) ∨ (Triangle0 ∧ IsosTri) ∨ (Triangle0 ∧ ScalTri) ∨ (Triangle0 ∧ RightTri) ∨ NumError ∨ TriangleError

The test strategy which has been adopted for Z specifications is first to derive tests which exercise the response of the software to invalid code. The state space schema Triangle0 defines three input variables x?, y? and z? as integers lying in the range

0< x? <=200 0< y? <=200 0< z? <=200

For these input parameters, character and real types would be invalid. Also, entry of zero, two and four input parameters would also be invalid.

The functional information to be tested lies in the Z specification. The second part of the schema comprises a sequence of predicates which are either conjoined or disjoined; the absence of either operator is an implied conjunction. The disjunction of predicates defines different routes through the tree of functionality. For example,

```
 ________________
|(Predicate 1 ∨
|Predicate 2 )
|Predicate 3
|________________
```

defines two routes of functionality ie 1 and 3, or alternatively 2 and 3. All the routes through the triangle problem as defined by combinations of the schemas ScalTri, EquiTri, IsosTri and RightTri are exercised by deriving tests for them. Furthermore, the GAs derive tests which come close to the boundary of each functionality.

The whole system results in 13 test cases; the first seven relate to providing invalid inputs:
case 1 Invalid { }
case 2 Invalid { 50 45}
case 3 Invalid { 47 28 34 4}
case 4 Invalid { a e Q}
case 5 Invalid { 0.2 0.19 0.15}
case 6 Invalid { 201 201 201}
case 7 Invalid { 0 0 0}

Cases 8 and 9 simply provide valid input sets at the extremes of the allowed range:
case 8 Valid { 200 200 200}
case 9 Valid { 1 1 1}

The remaining cases generate valid data according to Triangle0, and check the remaining schemas:

case 10 Valid { 33 31 29}	*ScalTri*	*average generations needed 1*
case 11 Valid { 12 12 3}	*IsosTri*	*average generations needed 1*
case 12 Valid { 24 24 24}	*EquiTri*	*average generations needed 7*
case 10 Valid { 21 20 29}	*RightTri*	*average generations needed 8*

In general fewer tests were needed than for the structural testing because only the state space schema was conjoined with the function schema, whereas the structural test always dealt with the complete software subsystem of 5 procedures. However, covering the functionality would not necessarily cover the entire structure of the software, and this issue will be the subject of future investigations.

6. Discussion

Functional testing is valuable because it checks the functionality which may have been omitted from the code. Its omission cannot be recognised by structural testing. It is also able to check for exceptional cases when these have been recognised as a possibility and incorporated into the schemas. When these exceptions rely on the system raising pre-defined exceptions, there are no branch predicates in the code to alert the structural test that it should be checked. Calculations where the maximum permitted integer is exceeded come into this category. Explicit specifications define the structures of the software, and therefore generate similar tests to structural testing. Implicit specifications are quite different, and may present more problems for the functional test to cover the software.

Structural testing is valuable because the actual code is checked in its entirety for correctness. A functional test may fail to exercise code which should not be there, and its presence may cause problems during operation. A combination of functional and structural tests seems to be the safest approach.

GAs are valuable for deriving tests with complex data structures, and solving complex functions of several inputs. They can also guide tests to the sub domain boundary where errors in the code are more likely to be revealed. However, they may need much computational effort, though this does depend on the nature of the software under test. GAs have been used successfully in deriving test from both the structure and formal specification of software, and they merit further investigation particularly in respect of functional testing.

Future work will include the investigation of formal specifications for non-numerical problems, and of the use of simulated annealing to complement the use of GAs.

7 . Conclusions

GAs have been used successfully to derive tests with complex data structures, and to execute branches where the predicates are complex functions of the inputs. GAs can also guide tests to the sub domain boundary where errors are more likely to be revealed. One drawback is the computational effort which may be required to derive tests to cover a large program.

The value of applying GAs to structural testing has been established by covering all branches in a variety of procedures. Similar success has been achieved in deriving tests from a formal Z specification for a triangle classifier. More work is needed to confirm their usefulness to derive tests from formal Z specifications in a wider context.

References

1. Spivey, J M, *The Z Notation: a Reference manual*, 2nd edition, Prentice Hall, 1992.

2 Wordsworth, J B, *Software Development with Z: a practical approach to Formal methods in Software Engineering*, Addison-Wesley, 1992.

3 McMorran, M A, Nicholls, J E, *Z User Manual*, IBM UK Hursley Park Laboratories Technical Report TR12.274, 1989.

4 Holland, J, *Adaptation in natural and artificial systems*, University of Michigan Press, 1975.

5 Sthamer, H-H, Jones, B F, Eyres, D E, *Generating test data for Ada generic procedures using genetic algorithms*, Proc of Adaptive Computing in Engineering Design and Control, ISBN 0 905227 33 6,University of Plymouth, 1994

6 Liepins, G E, Hilliard, U R, *Genetic Algorithms: Foundations and applications*, Annals of Mathematics and Artificial Intelligence, **21**, 31-57, 1989.

7 Holmes, S T, Jones, B F, Eyres, D E, *An improved strategy for the automatic generation of test data*, Proc of Software Quality Management '93, 565-77, 1993.

8 Duran J W, Ntafos S C, *An evaluation of random testing*, IEEE Trans on Software Engineering, **10(4)**,438-44, 1984.

9 Hamlet, D, Taylor, R, *Partition testing does not inspire confidence*, IEEE Trans on Software Engineering, **16(12)**, 1402-11, 1990

10 Korel, B, *Automated software test data generation*, IEEE Trans on Software Engineering, **16(8)**, 870-9, 1990