SEARCH-BASED SOFTWARE ENGINEERING

Joachim Wegener, chair

Fitness Function Design to improve Evolutionary Structural Testing

André Baresel, Harmen Sthamer and Michael Schmidt
DaimlerChrysler AG, Research and Technology
Alt-Moabit 96a, 10559 Berlin, Germany, +49 30 39982 222
{andre.baresel;harmen.sthamer;michael.a.schmidt}@daimlerchrysler.com

Abstract

Evolutionary Structural Testing uses Evolutionary Algorithms (EA) to search for specific test data that provide high structural coverage of the software under test.
A necessary characteristic of evolutionary structural testing is that the fitness function is constructed on the basis of the software under test. The fitness function itself is not of interest for the problem; however, a well-constructed fitness function may substantially increase the chance of finding a solution and reaching higher coverage. Better guidance of the search can result in optimizations with less iterations, therefore leading to savings in resource expenditure.
This paper presents research results on suggested modifications to the fitness function leading to the improvement of evolutionary testability by achieving higher coverage with less resources. A set of problems and their respective solutions are discussed.

1 INTRODUCTION

Evolutionary testing designates the use of metaheuristic search methods for test case generation. The input domain of the test object forms the search space in which one searches for test data that fulfil the respective test goal. Due to the non-linearity of software (if-statements, loops, etc.), the conversion of test problems into optimization tasks mostly results in complex, discontinuous, and non-linear search spaces. The use of neighborhood search methods such as hill climbing are therefore not recommended. Instead, metaheuristic search methods are employed, e.g. evolutionary algorithms, simulated annealing or tabu search. In this work, evolutionary algorithms are utilized to generate test data, since their robustness and suitability for the solution of different test tasks has been proven in previous work. e.g. [19] and [17].

The only prerequisites for the application of Evolutionary Tests (ET) are an executable test object and its interface specification. In addition, for the automation of structural testing, the source code of the test object must be available to enable its instrumentation.

1.1 A BRIEF INTRODUCTION TO EVOLUTIONARY ALGORITHMS

Evolutionary algorithms represent a class of adaptive search techniques and procedures based on the processes of natural genetics and Darwin's theory of biological evolution. They are characterized by an iterative procedure and work in parallel on a number of potential solutions for a population of individuals. Permissible solution values for the variables of the optimization problem are encoded in each individual. An overview of evolutionary algorithms is presented in Figure 1 and a detailed description can be found in [20] and [12].

The fitness value is a numerical value that expresses the performance of an individual with regard to the current optimum so that different individuals can be compared. Usually a spread of solutions exists ranging in fitness from very poor to good. The notion of fitness is fundamental to the application of evolutionary algorithms; the degree of success in their application may depend critically on the definition of a fitness that changes neither too rapidly nor too slowly with the design parameters of the optimization problem. The fitness function must guarantee that individuals can be differentiated according to their suitability for solving the optimization problem.

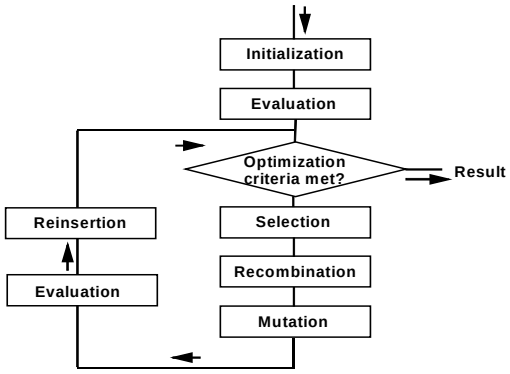


Figure 1: Overview of a typical procedure for evolutionary algorithms

Our experiments used a population of 300 individuals split into 6 subpopulations of 50 individuals. In order to combine multiple strategies, migration was introduced to permit an exchange of the best individuals between subpopulations at regular intervals. The subpopulations

also compete with each other. Strong ones receive more individuals, the others diminish in size. Details of the evolutionary settings are described in [20].

1.2 APPLICATION TO SOFTWARE TESTING

In order to automate software tests with the aid of evolutionary algorithms, the test goal itself must be transformed into an optimization task. This necessitates a numeric representation of the test goal, from which a suitable fitness function for evaluation of the test data generated may be derived. Different fitness functions emerge for test data evaluation according to which test goal is pursued. For structural testing, fitness functions may be based on a computation for each individual that indicates its distance from the desired program predicate execution [19] and [17]. For example, if a branching condition *"x==y"* needs to be evaluated as *True*, then the fitness function may be defined as $|x-y|$ (the fitness values are minimized).

Each individual within the population represents a test datum with which the test object is executed. For each test datum the execution is monitored, and the fitness value for the corresponding individual determined. It is important to ensure the test data generated are in the input domain of the test object.

2 STRUCTURE OF FITNESS FUNCTION

Structural testing is widespread in industrial practice and stipulated in many software development standards, e.g. [22], [23], [24], and [25]. The execution of all statements (*statement coverage*), all branches (*branch coverage*), or all conditions with the logical values *True* and *False* (*condition coverage*) are common test aims. The aim of applying Evolutionary Testing to structural testing is the generation of a quantity of test data, leading to the best possible coverage of the respective structural test criterion.

Whereas all previous work from other authors has concentrated on single selected structural test criteria (*statement-*, *branch-*, *condition* and *path-test*), DaimlerChrysler Research has generated a test environment to support all common control-flow and data-flow oriented test methods [20]. For this purpose, the structural test criteria are divided into four categories, depending on control-flow graph and required test purpose:

- node-oriented methods,
- path-oriented methods,
- node-path-oriented methods, and
- node-node-oriented methods.

The separation of the test into partial aims and the definition of fitness functions for partial aims are performed in the same manner for each category. Each partial aim represents a program structure that requires execution in order to achieve full coverage of the selected structural test criterion, i.e. each single statement represents a partial aim when using statement coverage criterion. For the Evolutionary Test, the test therefore has to be divided into partial aims. These depend on the specified structural test criteria. Identification of partial aims is based on the control-flow graph of the program under test.

As mentioned previously, the definition of a fitness function that represents the test aim accurately, and supports the guidance of the search, is conditional to the successful application of Evolutionary Tests. In order to define the fitness function, this research builds upon previous work dealing with branching conditions (among others [17], [8], and [18]). These are extended in [20] by introducing the idea of an approximation level. A more detailed definition of approximation level for node-oriented and path-oriented methods is provided in sections 2.1 and 2.2. These are the basis for the remaining node-path-oriented and node-node-oriented methods and, for this reason, the last two methods are not further discussed here.

2.1 NODE-ORIENTED

Node-oriented methods require the attainment of specific nodes in the control-flow graph. The *statement test* as well as the different variants of the *condition test* may be classified in this category. As regards condition testing ([10] and [2]), a special case applies for the fulfillment of the respective test criterion. In addition to the branch nodes, the necessary logical value allocations for the atomic predicates in the conditions must also be attained.

For node-oriented methods, partial aims result from the nodes of the control-flow graph. The objective of the Evolutionary Test is to find a test data set that executes every desired node of the control-flow graph. For the *statement test*, all nodes need to be considered; for the different variations of the *condition test*, only the branching nodes are relevant. Condition testing also requires the predicates of the branching conditions to be evaluated. In the case of the *simple condition test*, for example, the evaluation of each of the atomic predicates must be inventoried to represent *True* and *False* partial aims. In the case of the *multiple condition test*, all combinations of logical values for the atomic predicates form independent partial aims.

In node-oriented methods, the fulfillment of a partial aim is independent of the path executed in the control-flow graph. This has been taken into account by our fitness function. The fitness functions of the partial aims consist of two components. In addition to the calculation of the distance in the branching nodes; which specifies how far away an individual is from fulfilling the respective branching condition (compare [17], [8], and [18]); an approximation level is introduced as an additional element for the fitness evaluation of individuals:

```
Fitness = AL  +  DIST
AL:   approximation level (natural numbers)
DIST: normalized  local  condition  distance
(value range 0..1)
```

The approximation level enables the comparison of individuals that miss the partial aim in different branching nodes (details in [20] and [1]). It indicates how close the

executed path is, as compared to the required partial aim. This extension enables different paths through the program, to the desired partial aim, to be treated equally with respect to the fitness evaluation. Unlike previous work, it is unnecessary to select a specific path to a distinct node through the control-flow graph. In this approach, only the execution of the specific node is of relevance. The higher the attained level of approximation the better the fitness of the individual. This extends the idea stated in [17], where a small fixed value was added to the calculated distance for every executed node belonging to a path that leads to the target node. Therefore, individuals closer to the target node receive a higher fitness value as compared to those that branch away earlier.

2.2 PATH-ORIENTED

Path-oriented methods require the execution of certain paths in the control-flow graph. All variations of *path tests,* from the *reduced path test* [6] to *complete path coverage* [7], belong to this category. Therefore all paths through the control-flow graph, necessary to fulfil the chosen structural test criterion, are determined and identified as partial aims.

Establishing fitness functions for path-oriented test methods is much simpler than for node-oriented methods because the execution of a certain path through the control-flow graph forms the partial aim for the Evolutionary Test. Corresponding to the node-oriented methods, the fitness function for path-oriented methods consists of two components: approximation level and distance calculation.

The covered program path is compared to the program path specified as a partial aim in such a way that the length of the identical initial path section reflects the approximation level (compare [9]), and the last non-fitting condition is used for distance evaluation.

3 IMPROVING THE FITNESS FUNCTION

In Evolutionary Structural Testing, the fitness function is constructed on the basis of the software tested. The function itself is not of interest for the problem, the only goal is to find a test datum that fits a test criterion. A well-constructed function can:

- considerably increase the chance of finding the solution and reach a better coverage of the software under test and
- result in a better guidance of the search and thus in optimizations with less iterations.

Other work on designing fitness functions and the results of the optimization process can be found in [8]; this investigates the use of various distance functions. Hamming distance, reciprocal function and their influence on optimization performance are discussed. In [8], a decision was made in favor of the Hamming distance because the authors used genetic algorithms with a bit representation of all parameters in their approach. Evolutionary algorithms are used with integer and floating point representations and their corresponding mutation and recombination operators. For this reason Hamming distance is not investigated in this paper.

Modifying the distance function of branch conditions is only one possible mechanism for modifying the fitness function. In the following sections it is argued that more general alterations to the fitness function may lead to better results in Evolutionary Testing. This results in a higher chance of finding the solution or a better performance of the optimization process in general.

Evolutionary testing has been successfully used for complex functions (e.g. shown in [20]). The optimization problem often relies on some details, for this reason the authors present very simple examples for the discussion.

3.1 COMPOSED CONDITIONS AND NESTED BRANCHES

In the general solution proposed, the test object is instrumented in such a way that the semantics of the software under test is not changed by the added code. Special care is taken regarding side effects that might occur during the execution of branch conditions if short circuit evaluation is implemented by the compiler. To keep side effects unchanged, the fitness function should only take into account the results of the executed parts of the condition. This leads to a problem with optimization performance and, at worst, to a low chance of successful optimization. Example 1 presents an extreme situation in which the solution of an input, which evaluates the condition as *True*, can only be found by optimizing the test data for the atomic conditions one after the other.

```
if(a==0 && b==0 && c==0 && e==0 && f==0)
  {  /* ex 1 - to be executed */ }
```

Example 1: Composed condition and its problem with short circuit evaluation

There is a very low probability of coming across a solution by chance which fits all of these atomic conditions. When executing this example code, only the first parts of the condition are evaluated until one atomic condition is evaluated as *False*. Whenever an individual is found that fits one more atomic condition, the probability of finding a solution which also fits the next one decreases considerably. This is due to the fact that a better solution must be found by not changing those input parameter values that correspond to these first condition parts. The number of input parameters that should not be changed increases with each atomic condition that is executed in the desired way. The same problem occurs with nested *if-then-else* structures. Consequently, a better fitness function should be introduced which could compensate for this behavior. Resolution of this issue for nested *if-then-else* structures will now be discussed. As this example shows, a better guidance of the search takes into account all parts of the composed condition. This facilitates the optimization of individuals for all atomic conditions at the same time. The function is constructed by the summation of all atomic

condition distances. This increases the chance of more effective mutations and achieves a well-performing recombination.

For this, the test object has to be changed to enable evaluation of all atomic conditions for every test execution so that no short-circuit evaluation is performed. This is not a problem for side-effect-free conditions when an instrumentation code is added in front of the *if-statement*. If the test object does not allow this due to side effects, the removal of these side effects by program transformation as shown in [5] may improve evolutionary testability.

No side effects are present in Example 1. A fitness function built upon the distances of all atomic conditions fundamentally increases the chance of finding the right solution as shown in Figure 2.

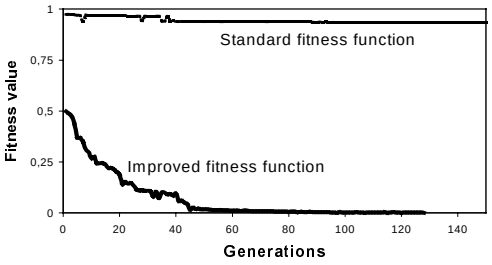


Figure 2: Optimization progress of the standard and improved fitness function

The fitness values of two test runs using the standard and improved fitness functions are displayed over the number of generations. The improved fitness function for complex conditions performs very well. It found a solution already after approximately 130 generations. However, using the standard fitness function, building only upon the evaluated atomic conditions, no individual was found within 1000 generations that achieved the partial aim.

Nested *if-then-else* structures may lead to the same behavior as the composed conditions. For nested *if-then-else* structures, shown in Example 2, the optimization is guided by the nested *if-conditions*. Static program analysis may identify the presence (or absence) of side effects. The knowledge of data dependencies makes it possible to pre-calculate where the value of a condition is fixed. In this case, the evaluation of conditions of inner *if-statements* may be performed earlier in the program under test in order to achieve the same improvements as described in Example 1.

```
If (a==0) {
   If (b==0) {
      If (c==0) {
      .../* ex1 */
```

Example 2: Code example for nested *if-statements*

A calculation of all conditions may be performed prior to the first *if-statement*. With this change of the fitness function, a search for a test datum that executes the statement *ex1* performs much better than a search with the fitness function solely on the basis of the executed program parts, since all conditions to be fulfilled are taken into account.

3.2 DEPENDENCIES WITHIN LOOP ITERATIONS

The fitness function that is used to optimize a test datum to execute a certain target node, as described in [11], takes control flow dependencies into account. These are all the branches of a program that lead to a part of the program from which the target node can no longer be reached, as shown in Figure 3. Loops have no special handling in this approach; this means that the evolutionary search of an input to traverse a target node within a loop has no guidance. Jones et al. [8] avoid this problem by unrolling the loop in the control-flow-graph for the fitness evaluation only.

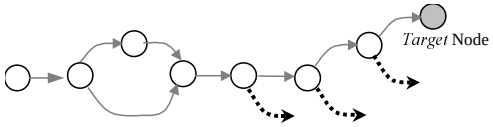


Figure 3: Target node with control dependencies

If the target node is inside a loop, every iteration produces another chance for traversing the target node as long as the loop is not exited. Missing the target node in one iteration has no effect on the fitness using control dependencies. Experiments showed that this hampers Evolutionary Testing efforts; since guidance to test data, resulting in a loop iteration where the execution is closer to the target node than others, is lacking. In many cases, this leads to a random search with a very low chance of finding a solution if the search space is large. Example 3 illustrates the problem.

```
for (idx=1;idx<=10;idx++)
{/* .... inner-pre-code ... */
 if (a==0) {
    if (b==0) {
    { /* Target Node – execute this*/ }
 /* .... inner-post-code .... */
}
```

Example 3: Dependencies in loops

In this simple example there is neither control dependence in the *pre-loop-code,* the *inner-pre-code*, nor in the *inner-post-code*. Only one branch has a control dependence for the selected target node; that is constructed by the loop header. This branch is executed when the counter *idx* is greater than 10. A fitness function building upon this information has no guidance to the target node; it gives the same poor fitness for all solutions that do not execute the target node. The search is therefore arbitrary since the search is not directed towards the execution of the target node.

A human tester would simply recognize additional information built by the nested *if-then* structure. This structure leads however, to no control dependence. This is due to the fact that if the target node in one iteration is missed, another chance presents itself in the next iteration. The solution proposed in this paper is to add dependencies

of one loop iteration to the fitness function. Whilst monitoring the execution of the test object, we can observe this information on all iterations and calculate a fitness from it. This may essentially improve the chance of finding a solution as shown in Figure 4.

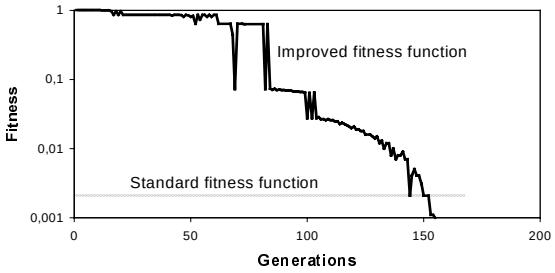


Figure 4: Optimization progress with regard to the control dependencies

Iteration control dependencies can be calculated by analyzing the control flow of one loop iteration. This leads to a set of branches that may miss the target node in a loop iteration. The approximation levels are also calculated for the additional branches. *Iteration control dependencies* are identified with the algorithm for control dependencies after removing the backward branches of a loop. The algorithm to distinguish control dependencies can be found in [16] (backward dominance).

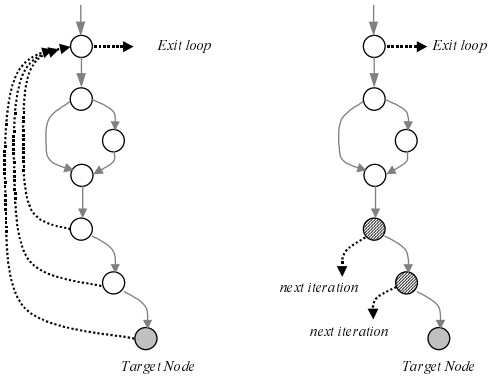


Figure 5: Control flow graph with *iteration control dependencies*

The figure shows the results for one example. The loop exit and backward branches have been highlighted in the left graph of Figure 5. In this example the target node is iteration-control-dependent from the two nodes where the backward branches begin as shown in the right graph.

When comparing the two fitness curves in Figure 4 one notices that, although the standard fitness function calculates a relatively good value from the beginning, whereas the other is relatively poor. This is because the standard fitness function is based on the local distance in *Exit loop*. This is not sufficiently meaningful for the target node and leads to stagnation. In contrast the improved fitness function also takes the conditions of both backward branches into account. Both fitness functions use the same evaluation principle, however the evaluations are carried out in different nodes.

The method used to estimate the backward branches for unstructured loops is not defined but first experiments using this approach are promising.

3.3 MEASURING PATH COVERAGE

The approach of applying evolutionary algorithms to path coverage cited in various publications ([9] and [18]) is to calculate the fitness of an individual by estimating the length of the first matching part of the target path and the actual executed path. This leads to a search where the solution is optimized for the branching conditions of the path in a stepwise manner. For example, when a solution is found for the first condition in the path, the next condition is considered. The poor behavior of an optimization, such as this, has been described previously in section 3.1.

A fitness function is introduced in which the length of all identical path sections is used as approximation level. This evaluation method is advantageous in that an individual diverging from the target path at the beginning, but covering the desired path towards the end, obtains a similarly high fitness value as compared to an individual covering the specified target path at the beginning, but diverging from it towards the end. The combination of two such individuals (recombination) may lead to a considerably better individual. In Figure 6 for instance, the execution of the first individual (covering the six nodes 1, 3, 4, 5, 7, and 8 on the target path), will obtain a high approximation level if all identical path sections are considered for the fitness evaluation. If only the first matching path section is measured, the second individual (covering five nodes 1, 2, 3, 4, and 7) will obtain a higher approximation level than the first one.

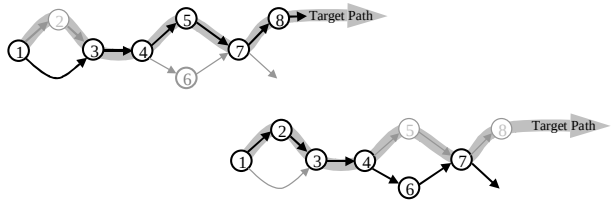


Figure 6: Execution of two individuals for a path-oriented test goal

The optimization goal is the gray path. Two possible execution paths have been highlighted. Example 4 illustrates the target path by *ex1*, *ex2* and *ex3*.

```
if (a==0) /* ex 1 */ ;
if (b==0) /* ex 2 */;
    else /* .... */;
if (c==0) /* ex 3*/ ;
    else return;
```

Example 4: Source code for path-oriented tests

Using the improved fitness function the optimization performs better in finding an input for the requested path that matches sub-paths because it always takes all conditions of the path into account. In contrast to this, other approaches, as described in many publications, optimize the solution condition by condition. The results are displayed in Figure 7.

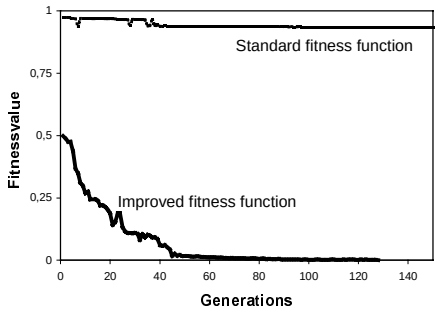


Figure 7: Optimization progress of path optimization

3.4 FITNESS OF DIFFERENT PATHS IN NODE ORIENTED TEST GOALS

For node-oriented test methods fitness evaluation is based on control dependencies of the target node. Measuring fitness is based on the point at which a control dependent node is evaluated incorrectly. An approximation level at this point is used to decide the closeness of the executed path to the target node.

A fitness evaluation was implemented that utilizes an approach level and a local condition distance. Approximation levels are calculated for all the nodes of a program that have a control dependence for the currently selected test goal. The control flow graph is examined for all possible execution orders of these special nodes. Based on this information, approximation levels are assigned. A more detailed description of approximation levels can be found in [20] and [1].

We were able to observe that in some instances the procedure of assigning the approximation levels, which lead to a well performing fitness function, was more complicated than just checking the execution order. This is further highlighted in Example 5.

```
   /* pre-code */
O  switch (a)
   {
O    case 1:
        if ( cond_1 ) return;
O       if ( cond_2 ) break;
        /* ... some code .... */
O    case 2:
        /* ... some code ... */
O       if ( cond_3 ) break;
        return;
   }
●  /* TARGET NODE */
```

Example 5: Source code template to assign the approximation levels

Example 5 has three paths that do not execute the target node:

- path through "1 → 2 → return",
- "1→ 2 → 3 → 4 → 5 → return" and
- "1 → 4 → 5 → return"

The open question for designing the fitness function is the closeness of the branching node 2 to the target node, which can be compared in Figure 8.

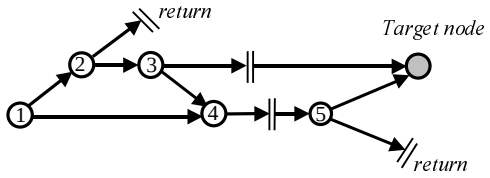


Figure 8: Control flow graph of Example 5

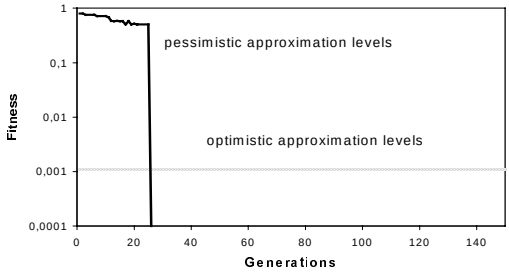
Node 5 gets the best approximation level since it leads directly to the target node if the branching condition is evaluated in the desired way. However, can node 2 achieve the same approximation level since one direct path to the target node exists? Or should this node obtain a different approximation level since another path leading to " 5 → return" exists? These two possibilities in assigning approximation levels to nodes will considerably alter the respective fitness function in the neighborhood of a solution that executes the target node. This may influence the performance of the search.

The above algorithms have been implemented on these two possible approximation level allocations in our structural test system. In this paper, they are called "optimistic" and "pessimistic" approximation levels since the level of node 2 is assigned on the basis of the direct path to the target node (optimistic), or to the path that misses the target node (pessimistic).

In order to check the behavior and performance of "optimistic" and "pessimistic" approximation levels, Example 5 has been tested in three versions with different conditions which can be seen in Example 6.

Version 1
```
cond_1: b>0 && b<4
cond_2: c>0 && c<10
cond_3: a == 1 && b==0 && c==0 && d==0
```
Version 2
```
cond_1: b==0
cond_2: c==2 && d==2
cond_3: b==0 && c==0 && d==0
```
Version 3
```
cond_1: b>0
cond_2: c==0 && d==0
cond_3: a==1 && b==0 && c==0
```

Example 6: Versions for "optimistic" and "pessimistic" approximation levels



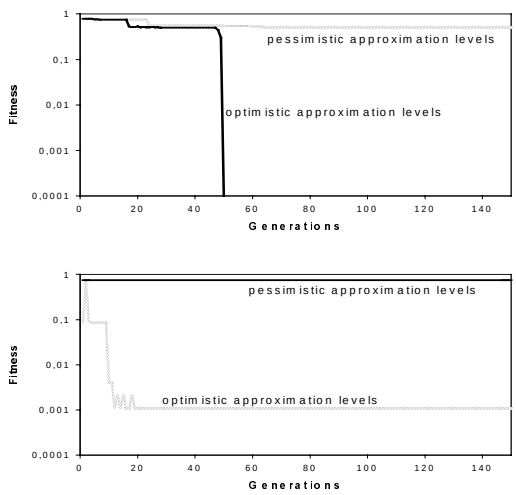


Figure 9: Optimization progress for the three different versions

The chosen expressions change the feasibility and chance of executing the different paths to the target node. As the figure shows, this leads to a different behavior of the fitness function based on *optimistic* versus *pessimistic* approximation levels. For version 3 no solution was found in the test. This is caused by no guidance of the search to execute the path which evaluates condition 2 to *True*. The authors suggest also using node 3 for fitness evaluation. Further research has to be done on this area, since from this point it cannot be decided which function should be used in general, or which analysis methods may identify the best performing fitness function. Another possibility is to use multi objective optimization, but this is not of the scope of this paper.

3.5 CODING OF INPUT SEQUENCES

For structural tests using EA, a test preparation module generates test driver code that maps the variable vector of the EA to parameters of the program under test. This was first introduced by [9]. Mapping can be designed for data structures, arrays and even for dynamic data structures e.g. lists and trees,

Support for dynamic arrays and lists has been implemented in our structural test system. Tests have shown that research on coding these structures is needed. This is because for a program under test, it is not only necessary to find special values in the correct sequences, but also to order them correctly. Finding the correct order with standard EA operators may be very difficult as the following example shows. Example 7 converts an input string of ten characters into a weekday.

```
// computation of day, month, year
// in a loop over the characters

If (year==1752) {
 if (month=9 && day>13)
    // handling of special dates..
}
```

Example 7: Source code illustrating sequence problem

The optimization tends to the solution string "9.9.1752". This achieves a good fitness since the correct year and month are found and the day is close to the 13th. To obtain better results the EA should be able to insert characters. A new coding of array parameters is introduced, enabling the EA to move the elements of a sequence easily.

This paper argues that other EA operators may not be used because variable vectors often consist of sequences of more than one variable (e.g. arrays of structures), as well as other parameters that are not part of the sequence. It was therefore decided not to use scheduling operators because of the possible mixture of variable types.

With the introduction of an additional array which is used to encode the order of the elements, the scheduling problem was mapped back into a parameter optimization. Appending an ordering array to the interface of a test object may be performed automatically within test driver generation whenever a sequence type is detected.

The additional coding array transforms the search space. Despite the increase in the input dimensions, the search problem is more easily solved by EA. This is due to the additional dimensions that introduce more solutions to the problem. Following the introduction of coding, the performance of the test system improved, whereas previously a solution would have been generated simply by chance when the optimization initially generated some special test data. Figure 10 displays the results.

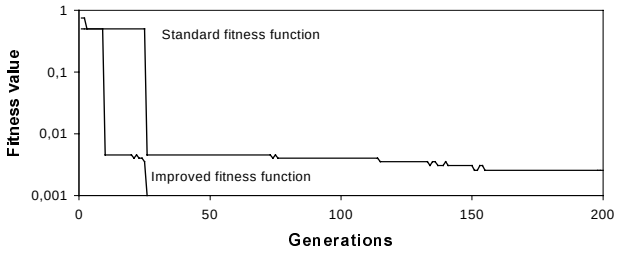


Figure 10: optimization progress

A real example of optimization of a sequence is shown in Table 1. Rows represent different individuals found during the optimization process. The last row displays one solution which is a string with a date of year 1752 in September after the 13th. The gray cells highlight the changes to the previous individual.

Character code	Ordering code	Resulting string
. 1 1 5 . 7 2 6 7 1	3 1 5 6 1 5 6 9 2 6	1 . 7 . 1 7 5 2
. 1 1 5 . 7 2 3 8 2	3 1 6 5 1 5 6 8 2 6	1 . 8 . 1 7 5 2
. 1 6 5 . 7 2 6 9 1	3 1 6 5 1 5 6 8 2 6	6 . 9 . 1 7 5 2
. 1 6 5 . 7 2 6 9 1	3 1 6 5 1 5 6 8 2 0	1 6 . 9 . 1 7 5 2

Table 1: The optimization process in detail; an overview of optimization steps (the best individuals are shown)

Results showed that additional coding may be of assistance for sequences in which the order is important. Consecutively, the values of an array must be optimized. This solution was tested and proved successful in

operation. Special EA operators for scheduling would also perform well if they were able to handle:

- order and parameter optimization at the same time and
- order optimization of sequence elements consisting of structures.

Further work needs to be carried out on complex structure coding, e.g. trees, where the order and the tree structure have to optimized. An example of this is when a partial aim is only executed when a special position in the tree holds a particular value.

4 CONCLUSION

The aim of this work is to enhance the construction of fitness functions in order to improve evolutionary testability by obtaining higher coverage with less resources.

The reason for the failure of an optimization is often difficult to analyze since the search space is usually very large and contains many dimensions. Therefore, visualization of optimization progress is problematic. Different problems and their various solutions have been discussed in this paper.

Poor optimization performance due to composed conditions, nested *if-statements*, and plausible improvement by the avoidance of short circuit evaluation have been discussed. This paper introduces fitness functions with an improved behavior for optimizing test data for target nodes in loops. An alternative method of calculating the fitness value for path coverage including its results on the optimization process is presented. Additionally, problems with approximation levels used to evaluate the closeness of an executed path to a target node are discussed. A search space transformation is introduced which uses a new coding of sequential input parameters.

Future work aims at further improvements to the evolutionary structural test. The idea of solving optimizations problems for difficult conditions as described in [3] is promising. The *chaining approach* introduced in the aforementioned paper may be used as problem transformation.

Further research has to be carried out in the areas presented when using ET for testing modules on a higher system level with sub modules and internal states.

Unstructured loops and loops with flags need more general transformation if the described approach cannot solve the optimization task. In this case a complete change of the program under test may be useful if it assists the discovery of test data for higher coverage of the original program.

References

[1] *Baresel, A.*: Automating structural tests using evolutionary algorithms,(German) Diploma Theses, Humboldt University of Berlin, Germany, *2000*.

[2] *Beizer, B.:* Software Testing Techniques, Van Nostrand Reinhold Company, *1983*.

[3] *Ferguson, Roger; Korel, Bogdan*: The Chaining Approach for Software Test Data Generation. ACM Transactions on Software Engineering and Methodology, Vol. 5 No.1, January 1996, pp.63-86.

[4] *Grochtmann, M., and Wegener, J.*: Test Case Design Using Classification Trees and the Classification-Tree Editor CTE. Proceedings of Quality Week '95, San Francisco, USA, 1995.

[5] *Harman, Mark; Hu, Lin; Munro, Malcolm; Zhang, Xingyuan*. Side-Effect Removal Transformation. IEEE International Workshop on Program Comprehension (IWPC 2001) Toronto, Canada

[6] *Howden, W.:* An Evaluation of the Effectiveness of Symbolic Testing. Software – Practice and Experience, vol. 8, pp. 381 – 397, *1978.*

[7] *Howden, W.:* Reliability of the Path Analysis Testing Strategy. IEEE Transactions on Software Engineering, vol. 2, no. 3, pp., *1976,* 208 - 215.

[8] *Jones, B.-F.; Sthamer: H.-H.; Eyres, D.*. Automatic structural testing using genetic algorithms. Software Engineering Journal, vol. 11, no. 5, pp. 299 – 306, *1996.*

[9] *Korel, Bogdan.:* Automated Test Data Generation. IEEE Transactions on Software Engineering, vol. 16 no. 8 pp.870-879; August 1990.

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

[11] *Pargas, R., Harrold, M. and Peck, R.:* Test-Data Generation Using Genetic Algorithms. Software Testing, Verification & Reliability, vol. 9, no. 4, pp. 263 – 282, *1999.*

[12] *Pohlheim, H.*: GEATbx - Genetic and Evolutionary Algorithm Toolbox for Matlab. http://www.geatbx.com/, 1994-2001.

[13] *Pohlheim, H.*: Evolutionäre Algorithmen - Verfahren, Operatoren, Hinweise aus der Praxis. Berlin, Heidelberg: Springer-Verlag, 1999. http://www.pohlheim.com/eavoh/

[14] *Pohlheim, H, Wegener, J., Sthamer, H.*: Testing the Temporal Behavior of Real-Time Engine Control Software Modules using Extended Evolutionary Algorithms. in Computational Intelligence, VDI-Berichte 1526, Düsseldorf: pp. 61-66, 2000.

[15] *Ronald E. Prather and J.Paul Myers, Jr.*: The Path Prefix Software Testing Strategy, IEEE Transactions on Software Engineering, Vol.13 No. 7 July 1987.

[16] *Schaeffer, Marvin.*: A mathematical theory of global program optimization. Prentice-Hall Inc., 1973.

[17] *Sthamer, H.-H.*: *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD Thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.

[18] *Tracey, N., Clark, J., Mander, K. and McDermid, J.:* An Automated Framework for Structural Test-Data Generation. Proceedings of the 13th IEEE Conference on Automated Software Engineering, Hawaii, USA, *1998.*

[19] *Wegener, J.; Sthamer, H.; Jones, B.; Eyres, D.*: Testing Real-time Systems using Genetic Algorithms. Software Quality Journal, vol. 6, no. 2, pp. 127 – 135, *1997.*

[20] *Wegener, J; Sthamer, H. ; Baresel, A. (2001*): Evolutionary Test Environment for Automatic Structural Testing. Special Issue of Information and Software Technology, vol 43, pp. 851 – 854, 2001.

[21] *Wegener, J., and Grochtmann, M.*: Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing. Real-Time Systems, 15, pp. 275-298, 1998.

[22] Capability Maturity Model for Software, Software Engineering Institute, Carnegie Mellon University.

[23] IEC 65: A Software for Computers in the Application of Industrial Safety-Related Systems (Sec 122).

[24] RTCA/DO-178B Software Considerations in Airborne Systems and Equipment Certification.

[25] Vorgehensmodell zur Planung und Durchführung von Informationstechnik-Vorhaben des Bundesministeriums des Innern (Process Model for Planning and Realization of Information Technology Projects of the German Secretary of Interior).

Instrumenting Programs with Flag Variables for Test Data Search by Genetic Algorithm

Leonardo Bottaci
Computer Science Dept.
Hull University
Hull, HU6 7RX, U.K.

Abstract

Evolutionary search is potentially a powerful way of searching for software test data to satisfy various structural testing criteria. Specific test cases are evaluated by a fitness function constructed by instrumenting the program under test. The more discriminating the fitness function, the more efficient the search. When a program uses flag variables to store the results of predicate expressions, it is difficult to instrument the program effectively. The problem is examined and a solution is given for a special case. An approach for tackling the general cases is described.

1 INTRODUCTION

The testing of software is a time consuming and expensive activity and consequently the idea that it might be automated is an attractive prospect. The vast majority of testing tools in current use focus on automating the execution of test cases and on collecting coverage and output data to compare with known results. What these tools do not do, however, is generate the test data to satisfy a given criterion. In some cases, the data may be generated randomly but this is unlikely to be adequate to satisfy the criterion. After application of an initial set of tests, testers often have the problem of constructing additional tests to completely satisfy the given criterion. Even with a good knowledge of the subject program under test this can be challenging. The problem is worse when testers work, as they often do, on code written by others.

The automation of test data generation is a problem that has been tackled by a number of researchers. Ince (Ince 1987) gives an account of relatively early work in this area. The test data generation problem, for a nontrivial criterion, is undecidable in general. This, coupled with the increased awareness of the potential of heuristic search techniques has prompted researchers to use these techniques to find test data for various structural testing criteria. Korel (Korel 1990) and Ferguson (Ferguson and Korel 1996) have used function minimisation to find tests to satisfy path criteria. Jones *et al.* (Jones, Sthamer, and Eyres 1996) and Wegener *et al.* (Wegener, Sthamer, Jones, and Eyres 1997) have applied genetic algorithms (Goldberg 1989) to find test data to satisfy branch coverage and minimum and maximum execution times. Tracey *et al.* (Tracey, Clark, and Mander 1998) have used simulated annealing to search for failure conditions. A genetic algorithm is used in the GADGET (McGraw, Michael, and Schatz 1998), test data generation system. Pargas *et al.* (Pargas, Harrold, and Peck 1999) describe a test tool in which a genetic algorithm is used to search for test data that reaches a given node in the program control flow graph. The conformance of the test execution path to the control dependency conditions (Ferrante, Ottenstein, and Warren 1987) for the given node is used as the fitness function. The tool developed by Wegener *et al.* (Wegener, Baresel, and Sthamer 2001) uses evolutionary algorithms and, by combining node and path conditions, may be used to generate test data for most structural test criteria.

A common technique in the work mentioned above is the instrumentation of the subject program to produce a heuristic evaluation function or fitness function. Ultimately, the fitness function must evaluate the extent to which a given test case satisfies specific predicate expressions in the subject program. When the value of a predicate expression is stored in a flag variable there is the risk of losing the information gathered by the instrumentation code. This paper considers this problem and shows how to instrument a special case. An approach for tackling the general case is also described. The techniques described in this paper have

been implemented in a prototype test data generation tool.

2 INSTRUMENTING A PREDICATE EXPRESSION TO SEARCH FOR TEST DATA

Consider the control flow graph of a subject program. The nodes are the basic blocks of the subject and the edges are the possible transitions between basic blocks. The conditional transitions are associated with a branch predicate. There is a distinguished start node and a distinguished exit node. Many test criteria require a test case to execute a given statement. If the particular path is unimportant then the control dependency predicate path (or sometimes paths) of the goal statement specifies the required value for each critical branch predicate expression. A predicate expression may not be critical because it is not part of any path to the goal statement or because both outcomes may lead to the goal statement.

There are a number of approaches, grouped under the general heading of static methods, that attempt to calculate the appropriate condition on the input data by analysis of the program. One of these is symbolic evaluation (Clarke 1976) (Howden 1977). As the name suggests, the program itself is not executed but instead a description is constructed of how execution along a particular path would affect a set of variables. This description cannot give the precise values of variables but instead provides constraints on their possible values, expressed, ultimately on the values of the input variables. In general, however, the relationship between the input data and the values of internal variables at the point where they are used in a predicate expression may not be readily analysable because of the presence of loops and computed storage locations, e.g arrays and pointers.

Dynamic analysis is an alternative to static methods. In this approach, the subject program is instrumented in order that it may reveal, during execution, the information that can be used to guide the search towards the required test case. In the most basic instrumentation, a record is kept of the values of all branch predicate expressions executed. A cost for the given input is computed by counting the number of predicate conditions in the control dependency path of the goal statement that have not been satisfied by the execution of the program. This is the method described by Pargas *et al.* (1999). A zero cost indicates that a solution has been found whereas a nonzero cost indicates that an undesired branch was taken at some predicate.

Although a count of undesired branch decisions provides some guidance to the search; in some situations, the first few conditions of the control dependency path will be easily satisfied but the next condition may be quite difficult to satisfy. Our experience is that during genetic search in this situation, the entire population of test cases all quickly evolve to the same fitness which is that obtained by satisfying only the easy predicates. At this point, the search space has become flat and the search becomes random.

As an example, consider the program fragment below.

```
...
if (a <= b)
   ...          // EXECUTION REQUIRED
                // TO ENTER THIS BRANCH
```

Suppose we are seeking a test case that will cause execution of the true branch of the conditional shown above. If the required branch is difficult to enter, many test cases will cause `a <= b` to be false. To discriminate between these tests, the program in instrumented to calculate a cost measure that penalises those tests that may be considered to be "far from" satisfying `a <= b`. For this expression a suitable cost measure would be $a - b$. A test that has a zero cost ($a - b = 0$) "just" satisfies the condition. A positive cost indicates that the predicate expression is false.

The subject program is instrumented at the point a particular condition is required to hold, in our example at `a <= b`. Through instrumentation, the subject program has in effect been converted into another program that computes a function that we seek to minimise to zero. This method has been used by Korel (Korel 1990), Tracey *et al.* (Tracey, Clark, Mander, and McDermid 1998), Wegener *et al.* (Wegener, Baresel, and Sthamer 2001) and others.

Below are shown the typical relational predicate cost formulae. a, b are numbers and ϵ is the smallest positive constant in the domain (i.e. 1 in the case of integer domains and the smallest number greater than zero in the particular real number representation).

Predicate expression	Cost of not satisfying predicate expression
$a \leq b$	$a - b$
$a < b$	$a - b + \epsilon$
$a \geq b$	$b - a$
$a > b$	$b - a + \epsilon$
$a = b$	$abs(a - b)$
$a \neq b$	$\epsilon - abs(a - b)$

The cost formulae must be extended beyond the re-

lational predicates to the logical predicates to provide a cost for branch predicate expressions such as `a <= b and not(x > 0)`. As a simple case, consider the logical negation operator. Presented with the argument *true* with cost c it must return $false$ with cost $-c+\epsilon$. This cost formula for negation follows from consideration of the costs of $a \leq b$ and $a > b$. Possible cost tables for *or* and *and* are given in Table 1 where c_a is the cost representation of a boolean value a. In the table below, c_a and c_b are always positive so that the four rows correspond to the four rows of the classical truth table for two boolean values.

Table 1: Logical Operator Cost Table

a	b	a *or* b	a *and* b
c_a	c_b	$min(c_a, c_b)$	$max(c_a, c_b)$
c_a	$-c_b$	$-c_b$	c_a
$-c_a$	c_b	$-c_a$	c_b
$-c_a$	$-c_b$	$min(-c_a, -c_b)$	$max(-c_a, -c_b)$

The intuition behind the cost formula for *or* is that if either of the c_a or c_b costs are to be incurred we need incur only the least cost hence the *min* function. When both costs must be incurred, we are obliged to accept at least the maximum cost[1].

Tracey *et al.* (Tracey, Clark, Mander, and McDermid 1998) use essentially the same cost functions although their's are restricted to nonnegative values. A notable difference, however, is the use of + rather than *max* for *and*. One can argue that when both costs must be incurred, we are obliged to accept them both. There are also situations in which + is a better operator than *min* for *or* and the above truth table is presented only as a heuristic; the precise cost formulae used is not relevant to the subject of this paper.

3 FLAG VARIABLE PROBLEM

Given an effective cost function, problems can nonetheless arise in trying to use it. A particular problem is that sometimes the point in the program where a predicate expression is evaluated, this is the point at which a cost may be calculated dynamically, may not be the point at which the predicate value is used, which is where the cost value is required. In the software testing literature, this is often referred to as the boolean flag problem.

[1]Note that the above formulas are consistent with De Morgan's laws. For example, the cost of the negation of *a or b* is $-min(c_a, c_b) + \epsilon$ which is equal to $max(-c_a + \epsilon, -c_b + \epsilon)$ which is the cost of *not a and not b*. This cost equality is a stronger condition than is necessary, we require only that truth values be preserved.

3.1 SPECIAL CASE

The following code fragment illustrates a special case of this problem.

```
flag := a <= b;  // COST a - b CALCULATED
...
if (flag)        // COST a - b REQUIRED
   ...           // AS EXECUTION REQUIRED
                 // TO ENTER THIS BRANCH
```

In the predicate expression of the condition, we require `flag = true` but this is not a useful expression to instrument because a boolean variable can provide only one of two values leading to an ineffective cost function with a flat surface.

Since the problem arises because the information that is used to compute the boolean value of the flag is discarded once the flag is set, the solution described in this paper is to retain this information so that it may be used later in the execution when, for example, the flag is evaluated as part of a conditional. In this way, when the flag variable is evaluated in the predicate of the conditional, it may be associated with the cost of the expression `a <= b` computed at the time that the flag was set earlier, in the execution.

A prototype test data generation tool has been constructed in which the above scheme has been implemented. All predicate expressions in the subject program are instrumented to compute the costs described in the previous section. In addition, whenever any variable is assigned the value of a predicate expression, the cost of that expression is associated with the variable. In the example above, if `flag` is set then the cost $a - b$ is associated with `flag`. When the variable `flag` is used in the if-statement, the saved cost is retrieved and associated with the if-statement.

There may be a number of statements where a flag variable may be set and used during the execution of a program as for example

```
if (...)
   flag := a <= b;   // COST a - b IF EXEC
else
   flag := a >= c;   // COST c - a IF EXEC
...
flag := not(flag) and (x > 0);  // COST OF
                                // flag USED
                                // AND SAVED
```

Whenever a variable is set to the value of a predicate expression, the variable is also associated with the cost

of that expression and so the cost of any expression involving flag variables may be calculated.

Note that the above scheme allows for the cost of a predicate expression to be associated with the assignment of a value to any variable, even a computed variable such as an array element or pointer reference. Pointer variables have not yet been implemented in the prototype but they will not present a problem for this scheme.

3.2 GENERAL CASE

The above technique fails, however, when the boolean expression that in effect determines the flag value is not directly assigned to the flag but is used to control the assignment of a "summary" value, as is shown in the following fragment.

```
flag := false;
...
if (a <= b) {      // COST KNOWN HERE
   ...
   flag := true;
}
...
if (flag)       // COST POSSIBLY USEFUL HERE
   ...          // DESIRED BRANCH
```

When the flag is false and it is desired to set it true there is no cost value associated with the flag that can be use to guide the search towards satisfying `a <= b`. The cost of this expression is computed, however, but it is associated only with the first conditional statement.

It is not at all clear how the computed cost can be propagated usefully in this situation. To establish that this cost is even relevant to the problem it is necessary to recognise that the second assignment to `flag` is relevant to the selection of the required branch. Data dependence analysis (Aho, Sethi, and Ullman 1986) could be used to do this. We might then identify the conditional statement closest to the unexecuted assignment to the flag, i.e. that conditional which controls entry to the basic block that contains the assignment. If in the code `flag` is set true then the cost of the conditional predicate expression should be associated with the use of the flag in the second conditional statement and conversely if in the code `flag` is set false (and a false value is required) then the flag should be associated with the negation of the cost of the conditional predicate expression.

In the above fragment, we have the benefit of knowing that if the flag is set, it is set true, in general, unless the statement is executed, the value of the flag is unknown. With an unknown flag value there is the danger that the cost of the conditional predicate expression is not useful since it may guide the search towards the execution of a statement that does not change the value of the flag to the required value. A more difficult case is shown in the fragment below.

```
...
if (a <= b) {       // COST KNOWN
   ...
   if (a > c) {     // COST KNOWN IF a <= b
      ...
      flag := true;
```

The variable `flag` is set true only when both predicate expressions `a <= b` and `a > c` are true. This suggests that the cost to be associated with the value of `flag` at the assignment is the cost of `a <= b` *and* `a > c` but if `a <= b` is false then there is no cost for `a > c`.

In general, flag values can be known only when they are set and cost information can be collected only when predicate expressions are executed. The need to execute code in order to analyse it is a fundamental limitation of dynamic program analysis in general.

Ferguson*et al.* (Ferguson and Korel 1996) also tackle the problem of generating test data in a program with flag variables. They do not associate costs with flag values. The approach they take when a search fails to find a test case that will execute a required branch is to identify the statements which could affect the value of the flag. Data dependence analysis (Aho, Sethi, and Ullman 1986) is used to do this. Once these statements are identified, their execution become subgoals of the test generator. In this way, the search for statements to execute is goal directed and depth first in that a subgoal is pursued before a sibling goal.

Clearly, program execution must be directed to currently unexecuted parts of the program. It may not be necessary, however, to use a focussed technique to identify the specific statements that may affect the predicate expression under consideration. In any case, it is not possible to tell if the execution of the statement will solve the problem unless the statement is executed. A much simpler approach, for example, would be to attempt to execute all acyclic paths that reach the problem node.

A genetic algorithm is well suited to exploring many areas of the search space in parallel (breadth first search on a serial machine). The initial population of random test cases could be separated into separate subpopulations or islands. In each island there would

be a search for a specific acyclic path. Given the use of a genetic algorithm as a search tool for this work it seems sensible to investigate if the ability of the genetic algorithm to search different parts of the search space breadth first can be exploited to solve this problem. It may turn out that a search for test cases that will execute all acyclic paths to the required branch is reasonably efficient in practice in which case it is possible to avoid the implementation complexities of a more goal directed search.

4 IMPLEMENTATION

A prototype test data generation system has been written (using CMU Common Lisp) to apply the ideas described in this paper to example programs. The prototype has two main modules. One module is concerned with the instrumentation of the subject program and the other, smaller, module is responsible for searching for test data using a genetic algorithm.

The subject program is parsed into an abstract syntax tree[2]. From this tree is generated the instrumented subject program. The subject and mutant program is instrumented as follows. Each conditional statement has a fixed length FIFO queue[3] in which predicate expression costs are saved as they are calculated. In addition, the conditional retains the lowest positive cost produced (recall that the cost is positive when the predicate is false) and the highest non-positive cost (recall that the cost is zero or negative when the predicate is true). In this way it is possible to determine if both branches have been taken. Each variable also has a similar fixed length FIFO queue in which is saved the cost of any predicate expression value assigned.

In addition to the subject program, the only additional information that the user may supply is a constraint and probability distribution on the subject program input domain. This is done by defining subsets of the input domain and assigning a probability to each subset. The input domain definition and probability distribution is used to create the initial random population of test cases. For each designated subset, the user may also specify the parameters of the genetic mutation operator (uniform or Gaussian distribution and variance).

Test inputs are coded not as binary strings but as strings of atomic values of the common programming language data types, i.e. integer, float, etc. The genetic algorithm is of the steady-state variety and similar to Genitor (Whitley 1989). Reproduction takes place between two individuals who produce one or two offspring (depending on the choice of reproduction operator). These offspring are then immediately inserted into the population expelling the one or two least fit. The population is kept sorted according to cost and the probability of selection for reproduction is based on rank in this ordering.

5 CONCLUSIONS

The instrumentation of predicates in a program under test is a common technique for guiding the search for test data. The presence of flag variables, however, has long been recognised as an impediment to such instrumentation. The approach described in this paper is to propagate the cost information from the predicate expression instrumentation, i.e. the statement in the program where it is calculated, to the conditional where it is required. This technique cannot solve the general problem with flag variables but here it is proposed to use the genetic algorithm to search among the relevant acyclic paths.

[2] Currently the subject program must be hand translated into a common input language. The common input language is an expedient that for the purpose of research avoids the need to construct a parser for the language of the subject program. It has not yet proved to be a handicap since it is not difficult to find quite small subject programs that provide the test generator with a difficult challenge. The common input language is procedural and block structured although not all the features found in this type of language are as yet available. In particular, pointers are absent and the array is the only aggregate data type.

[3] A queue of costs is required to tackle the problem of instrumenting code within loops, a problem not relevant to this paper.

References

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers: Principles, Techniques and Tools.* Addison - Wesley.

Clarke, L. A. (1976, September). A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering SE-2*(3), 215–222.

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

Ferrante, J., K. J. Ottenstein, and J. D. Warren (1987, July). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems 9*(3), 319–349.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning.*

Addison Wesley.

Howden, W. (1977). Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering SE-4*(4), 266–278.

Ince, D. C. (1987). The automatic generation of test data. *The Computer Journal 30*(1), 63–69.

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

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

McGraw, G., C. Michael, and M. Schatz (1998). Generating software test data by evolution. Technical Report RSTR-018-97-01, RST Corporation, Suite 250, 21515 Ridgetop Circle, Sterling VA 20166.

Pargas, R. P., M. J. Harrold, and R. P. Peck (1999). Test-data generation using genetic algorithms. *Software Testing, Verification and Reliabillity 9*, 263–282.

Tracey, N., J. Clark, and K. Mander (1998, March). Automated program flaw finding using simulated annealing. *Software Engineering Notes 23*(2), 73–81.

Tracey, N., J. Clark, K. Mander, and J. McDermid (1998). An automated framework for structural test data generation. *Procceedings of the 13th IEEE Conference on Automated Software Engineering*.

Untch, R. H., A. J. Offutt, and M. J. Harrold (1993). Mutation analysis using mutant schemata. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis ISSTA 1993*, New York, NY, USA, pp. 139–147. ACM.

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

Wegener, J., H. Sthamer, B. F. Jones, and D. Eyres (1997). Testing real-time systems using genetic algorithms. *Software Quality Journal 6*, 127–135.

Whitley, D. (1989). The genitor algorithm and selective pressure: why rank based allocation of reproductive trials is best. *Proceedings of the Third International Conference GAs.*, 116–121.

GPTesT: A Testing Tool Based on Genetic Programming

Maria Cláudia Figueiredo Pereira Emer
UFPR - Curitiba CP: 19081,
81531-970, Brazil
mpereira@inf.ufpr.br

Silvia Regina Vergilio
UFPR - Curitiba CP: 19081,
81531-970, Brazil
silvia@inf.ufpr.br

Abstract

Genetic Programming (GP) has recently been applied to solve problems in several areas. It has the goal of inducing programs from test cases by using the concepts of Darwin's evolution theory. On the other hand, software testing, that is a fundamental and expensive activity for software quality assurance, has the objective of generating test cases from the program being tested. In this sense, a symmetry between induction of programs based on GP and testing is noticed. Based on such symmetry, this work presents GPTesT, a testing tool based on GP. Fault-based testing criteria generally derive test data using a set of mutant operators to produce alternatives that differ from the program under testing by a simple modification. GPtesT uses a set of alternatives genetically derived, which allow the test of interactions between faults. GPTesT implements two test procedures respectively for guiding the selection and evaluation of test data sets. Examples with these procedures show that the approach can be used as a testing criterion.

1 INTRODUCTION

The use of software products in most areas of human activities has generated a growing interest in software quality assurance. Software Engineering techniques and tools were proposed with the goal of increasing the quality of the software being developed. In this context, the software testing activity has gained importance during the last decade and is considered fundamental.

In the literature, there are three groups of testing techniques proposed to reveal a great number of faults with minimal effort and costs: 1) functional technique: uses functional specification of a program to derive test cases; 2) structural technique: derives test cases based on paths in the control flow graph of the program; 3) fault-based technique: derives test cases to show the presence or absence of typical faults in a program.

These techniques are generally associated to the testing criteria. A criterion is a predicate to be satisfied to consider the testing activity ended, that is, to consider a program tested enough [16, 25]. It helps the tester in two major tasks: test case selection and test case evaluation.

Fault-based criteria consider that most programmers do their programs very similar to the correct program, according to a specification. This fact is known as "competent programmer hypothesis [8]". When the users test a program, they use the correct program that they have in mind, and if the program P being tested is not correct, there is a set of alternatives for P that can include at least one correct program. The fault-based criteria explore the use of alternatives for testing P [8, 9, 14, 20]. In most cases, the alternative program differs from P by a simple syntactic modification, that is, only a fault at a time is introduced. They assume that complex faults are detected by analyzing simpler faults, this assumption is named "coupling effect" [8, 22].

Some works [9, 14] assume only necessary conditions for discovering faults; that is, to reveal a fault is necessary to produce only an intermediate different state in the program and in its alternative, after the modified statement. This is assumed because determining sufficient conditions, which are the conditions to produce different final states, is undecidable (task related to the term coincidental correctness [3]). However, Morell[20] points out that these assumptions ignore the global effect of faults or interactions of modifications in the

program.

Genetic Programming (GP) is a field of the called Evolutionary Computation. The term was popularized by Koza in 1992 [15]. The goal is to use the concepts of Darwin's evolution theory [6] for computer program induction. The concepts are usually applied by genetic operators such as: selection, crossover, mutation and reproduction. During the last decade, GP has received significant attention and been used to solve a large number of problems, mainly in Artificial Intelligence and Engineering Areas [1].

Some authors mention that there is a symmetry between the testing activity and the induction of programs [2, 3, 28]. In this sense, testing is an activity that generates test cases from a program being tested, and GP is a technique that generates programs from test cases.

Based on such symmetry, this work describes GPTesT tool, that supports a GP-based test approach. The alternatives are generated using GP and can differ from P by more than simple modifications. GPTesT guides the tester in two tasks: selection and evaluation of test cases. It allows the test of C programs and uses Chameleon [26], a GP tool. The paper is organized as follows. Section 2 shows aspects related to GP and the Chameleon tool. Section 3 presents a review about the test activity. Section 4 describes GPTesT and Section 5 illustrates the mentioned test procedures. Finally, Section 6 concludes the paper.

2 ABOUT CHAMELEON

Genetic Programming (GP) was introduced by John Koza [15], based on the idea of Genetic Algorithms presented by John Holland [13]. Instead of a population of beings, GP works with a population of computer programs. The goal of the GP algorithm is to naturally select the program that better solves a given problem, through recombination of "genes",. A special heuristic function called fitness is used to guide the algorithm in the process of selecting individuals. This function receives a program and returns a number that shows how close this individual is to the desired solution. First, an initial population of computer programs is randomly generated (Generation 0). After that, the GP algorithm enters a loop that is ideally executed until a desired solution is found.

In this paper, the tool Chameleon [26] illustrates the use of GP for software testing. Chameleon implements a grammar-oriented approach and evolves C programs. It represents the programs using grammar-based derivation trees.

Through the evolution process, genetic operators recombine programs by making modifications directly on their derivation trees. In reproduction, no change is made: the individual is simply replicated to the next generation. It is equivalent to the asexual reproduction of beings. Mutation is the addition of a new segment of code to a randomly selected point of the program.

Crossover is the operator that truly performs recombination of computer programs. This operation takes two parents to generate two offspring. A random point of crossover is selected on each parent and the subtrees below these points are exchanged. It is equivalent to the sexual reproduction of beings. When grammars are used, the crossover operator is restricted and only allows the exchange of tree branches that have been generated using the same production rule.

To execute Chameleon, the user needs to provide the grammar correspondent to the problem to be solved and an initial configuration I of parameters. The parameters are related to the genetic operations as mutation and crossover rates; to the number of runs and size of population; to the derivation tree; and to the name of a file that contains a set T of test cases. These test cases are used to calculate the fitness value of each individual. The number of runs is used to end the process. The individual (or program) with better fitness value is selected. The selection can also be random.

Figure 1 shows an example with the initial configuration I of parameters, including the grammar, for language C, adopted to the problem of calculating the common minimum multiple of two given numbers (*cmm* problem). Chameleon finds, among other, the solution presented in Figure 2.

3 ABOUT TESTING

The main goal of testing is to find an unrevealed fault [21]. Hence, how to derive test cases for revealing as many faults as possible is an important question. This is because it is related to some factors such as efficacy, costs, limitations to automate the testing activity, etc. Other question to be considered is to known whether a program has being tested enough or how to evaluate a data test set T. These two questions, related to generation and evaluation of test cases, are discussed by Rapps and Weyuker in [24].

In order to guide the testing activity and answer the above questions, different testing criteria were proposed. They consider different aspects to derive the test data. Functional criteria use functional specification of a program to derive test cases. Boundary Value Analysis and Cause-Effect Graphs [23] are ex-

```
[begin]
[parameters]
population size=500
number of runs=10
tournament size = 10
maximum depth for initial random programs = 15
maximum depth  during the run = 30
crossover rate = 90
mutation rate = 0
elitist = N
threshold = 0.01
[compiler]
cl -nologo -G6 -MT -Fepop.exe
[result-producing branch]
terminal set = {X,Y}
function set = {%, !=, *, /}
output variable = Z
[result-producing branch productions]
<code>   -> <def> <prog> <result>
<def>    -> float R = 1, A = X, B = Y;
<result> -> Z = (A<op>B) <op> <var>;
<prog>   -> if (<expc>) {<prog1>} else {<atr>}
<prog1>  -> do {<bloco>} while (<expc>);
<bloco>  -> <exp>
<bloco>  -> <bloco> <exp>
<exp>    -> <var> = <var> <opm> <var>;
<exp>    -> <var> = <var>;
<expc>   -> <var> <opc> <cte>
<atr>    -> <var> = <cte>;
<opm>    -> %
<op>     -> *
<op>     -> /
<opc>    -> !=
<var>    -> X
<var>    -> Y
<var>    -> R
<cte>    -> 0
[fitness cases]
source -> cmm.dat
[end]
```

Figure 1: Initial Configuration for Chameleon

amples of functional criteria. Structural criteria derive test cases based on paths in the control-flow graph of the program. The best known structural criteria are control-flow and data-flow based criteria [16, 24, 27]. Fault-based criteria derive test cases to show the presence or absence of typical faults in a program, considering common errors in the software development process. The best known fault-based criterion is Mutation Analysis [8].

This work focuses on fault-based testing, and the Mutation Analysis criterion will be described in more detail. It consists basically of generating mutant programs for the program P being tested. Mutation Analysis considers two assumptions [8]: 1) the hypothesis of the competent programmer: "Programmers do their programs very similar to the correct program"; 2) cou-

```
cmm (int X, int Y)
{
   int A=X, B=Y, R=1;
   if (Y!=0){
     do {
       R=Y;
       Y=X%Y;
       X=R;
     } while (Y!=0)
   }
   else {
     X=0;
   }
   return (A*B)/R;
}
```

Figure 2: A Possible Solution for the cmm Problem

pling effect: "Tests designed to reveal simple faults can also reveal complex faults". It is also based on a set of mutation operators. A mutant is represented by a single mutation in the original program established by a mutation operator.

All mutants are executed using a given input test case set T. If a mutant M presents different results from P, it is said to be dead, otherwise, it is said to be alive. In this case, either there are no test cases in T that are capable to distinguish M from P, or M and P are equivalent. To satisfy the criterion, we have to find a test case set able to kill all non-equivalent mutants; such a test case set T is considered adequate to test P. Then, a mutant will be considered dead if its behavior concerning a test case is different from that of the original program. The Mutation Score, obtained by the relation between the number of mutants killed and the total number of non-equivalent mutants generated, allows the evaluation of the adequacy of the used test case set. The number of equivalent mutants generated is not determined automatically; it is obtained interactively as an entry from the tester, since the equivalence question is, in general, undecidable [5, 8].

In the literature, there are many testing tools. However, the complete automation of testing activity is not possible due to many testing limitations: infeasible paths, equivalent mutants, etc. In general, there is no algorithm to generate a test set that satisfies a given criterion. It is not even possible to determine if such set exists [12]. In spite of these limitations, there are in the literature many works addressing test data generation for satisfying testing criteria. Most recent studies have been exploring Genetic Algorithms [4, 17, 18, 19].

Proteum [10] and Mothra [7] are examples of testing tools based on mutation testing. These tools gener-

ate mutants by using mutation operators. Proteum has a set of 71 mutation operators and supports test of C programs. Mothra supports testing of Fortran programs. Different operators are, in general, defined for different programming languages and the mutants differ from the program being tested by a simple modification. Morell[20], however, points out that such fact ignores the global effect of faults or interactions of modifications in the program.

This work proposes the use of GP to derive the mutants. This can produce alternatives that are very different from the original program and consequently can test global effects of faults. These aspects are discussed in the following section.

4 GPTesT

In this section we describe GPTesT (**G**enetic **P**rogramming-based **Tes**ting **T**ool) implemented to support GP based testing. It implements two test procedures for selection and evaluation of test cases, showing that the approach can be used in the same way as a testing criterion, such as Mutation Analysis.

Figure 3 contains the Use Case Diagram for GPTesT. Next, we present a brief description and purpose of each use case and describe the main GPTesT functionalities.

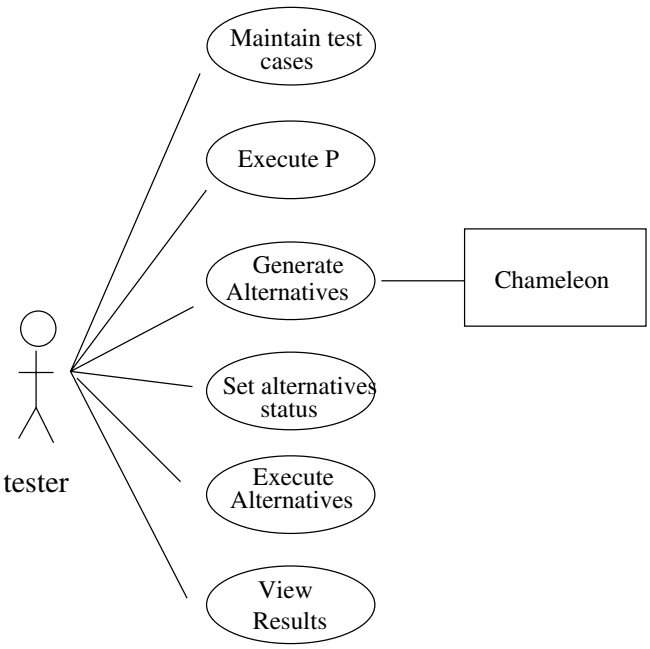


Figure 3: GPTesT: Use Case Diagram

- Maintain test cases: this use case is related to different functions for test case maintenance. The tester can add a new test case, delete or disable an existent, as well, visualize the obtained output after execution of the program P being tested. GPTesT saves all the given test cases as part of the testing session for P.

- Execute P: this use case executes P with all the non-executed test cases and saves the obtained output. The tester analyzes the output. If the output is different from the expected, a failure occurred. In this case, the tester must correct P and start a new testing session.

- Generate alternatives using Chameleon tool: this use case runs the Chameleon tool with the configuration I, as illustrated in Section 2. The tester gives I as entry. GPTesT selects the programs generated by Chameleon to obtain the set A of alternatives. This selection discards some anomalous and equivalent programs that can be syntactically determined.

- Set alternative status: each alternative has a status. This status can be:
  - anomalous: the alternative has an anomalous behavior such as division by zero, loop forever, etc.
  - equivalent: the alternative computes the same function of P, producing the same outputs that P produces for any input.
  - dead: the alternative has already produced a different output for a test case when compared with P.
  - alive: the alternative has produced the same output produced by P for all enabled test cases.

  After executing the alternatives, GPTesT automatically updates the alternative status. However, the tester has to identify the equivalence of programs and to set the status of an equivalent alternative. As mentioned in Section 3, there is no algorithm to determine whether two programs compute the same function. This is an undecidable question and all fault-based testing tools have this limitation.

- Execute the alternatives: this use case executes all the alive alternatives from A with all the non-executed test cases.

- View the results: this use case allows the tester to visualize the alternatives and their status, the test

cases and a testing score, similarly to other fault-based testing tools and Proteum. To calculate the score, GPTesT uses the following formula:

$$S_M(P,T) = \frac{A_d(P,T)}{A(P) - A_e(P)}$$

where:

- P: program being tested;
- T: a test data set;
- $S_M(P,T)$: the coverage score;
- $A_d(P,T)$: total number of dead alternatives;
- $A(P)$: total number of alternatives;
- $A_e(P)$: total number of equivalent alternatives.

This initial version of GPTesT allows the unit test of programs in C language. GPTesT, as well Chameleon, are oriented to C functions, where only a C function is tested at each time. All the results are saved in files, which are in a directory. To generate the executable alternatives, GPTesT uses the compilation command from I (Chameleon configuration).

GPTesT was developed using the Unified Modeling Language (UML) and implemented in C++. Figure 4 presents the main class diagram for GPTesT. More details about GPTesT implementation are in [11].

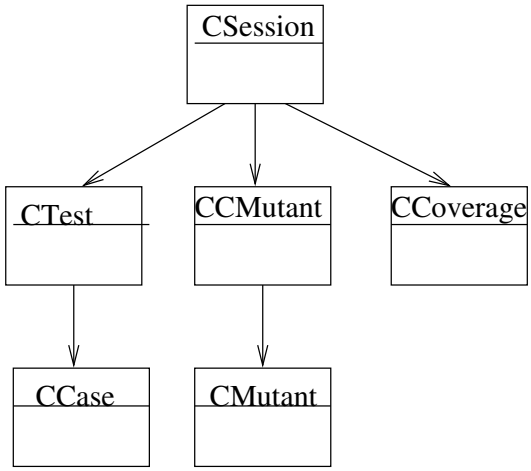


Figure 4: GPTesT: Main Classes

According to some authors [16, 24], a testing criterion or tool must support two testing procedures: selection and evaluation of test cases. Next section illustrates these procedures using GPTesT.

5 TEST PROCEDURES USING GPTEST

5.1 Selection of Test Cases

To illustrate each step of the selection procedure, we use the *cmm* program, whose source code is in Figure 5. This program prints the common minimum multiple of two given numbers.

Suppose that the tester wants to test *cmm*, and does not have any test case. GPTesT guides the tester in the task of test case selection, using GP to perform a fault-based testing. The tester takes the following steps:

```
int cmm (int a, int b)
{
    int  A, B, r;
    A = a;
    B = b;
    if (b!=0)
      do {
        r = A%B;
        A = B;
        B = r;
      } while (r!=0);
    else
      a=0;
    return (a*b)/A;
}
```

Figure 5: Source Code of *cmm* program

1. GPTesT initialization: gives initial information for GPTesT summarized in Table 1.

Table 1: Initial Information to GPTesT

Section Name	cmm
Source Code	cmm.cpp
Initial Chameleon Configuration I	illustrated in Section 2

2. Selection of alternatives: for *cmm* a set of 44 alternatives were generated. Examples of these alternatives are in Figure 6.
3. Generation of test cases to kill the alternatives: to kill the alternative, the tester has to identify a test case that produces an output that differs from P output. Observe that the test case (a=2, b=4) kills the alternative from Figure 6b. P produces 4 and the alternative produces a division by zero.

```
cmm (int X, int Y)
{
   int A=X, B=Y, R=1;
   if (X!=0){
     do {
       R=R%R;
       R=Y;
       X=X%Y;
       Y=X;
       X=R;
     } while (Y!=0)
   }
   else {
     X=0;
   }
   return (A*B)/R;
}
```

a)

```
cmm (int X, int Y)
{
   int A=X, B=Y, R=1;
   if (Y!=0){
     do {
       Y=X%Y;
       X=Y
       R=X%R;
      } while (Y!=0)
    }
    else {
      Y=0;
    }
    return (A*B)/R;
 }
```

b)

```
cmm (int X, int Y)
{
   int A=X, B=Y, R=1;
   if (Y!=0){
     do {
       R=Y;
       Y=X%Y;
       X=R;
     } while (Y!=0)
    }
   else {
     X=0;
   }
   return (A*B)/R;
}
```

c)

```
cmm (int X, int Y)
{
   int A=X, B=Y, R=1;
   if (R!=0){
     do {
       Y=X%Y;
       R=Y%R;
     } while (Y!=0)
    }
    else {
      X=0;
    }
    return (A*B)/R;
 }
```

d)

Figure 6: Examples of generated alternatives

4. Execution of the programs: using the compilation command of I and the test cases given by the tester, GPTesT executes P and the set of alternatives, producing results shown in Figure 7. The results show how many alternatives are dead, alive or equivalent and the score calculated. This final score was obtained with a set of 6 test cases.

5. Addition of new test cases: now, the tester visualizes the alive alternatives and continues the generation of test cases, repeating Steps 3 and 4 until all the non-equivalent alternatives are dead or the desired score is obtained. During this step, the tester manually identifies the equivalent alternatives. Figures 6c shows an example of equivalent alternative, that is identified by the tester.

Figure 8 presents the final status obtained for *cmm*. A score equal to 1 shows that all non-equivalent alternatives are dead using the test cases.

```
Total Number of Alternatives: 44
Anomalous Alternatives: 0
Live Alternatives: 3
Equivalent Alternatives: 0
Number of Test Cases: 6
Coverage Score: 0.931818
```

Figure 7: GPTesT Results

```
Total Number of Alternatives: 44
Anomalous Alternatives: 0
Live Alternatives: 0
Equivalent Alternatives: 3
Number of Test Cases: 6
Coverage Score: 1
```

Figure 8: Final status for program *cmm*

5.2 Evaluation of a test set

The tester also uses GPTesT for evaluation of a test set T. Consider the program P, which prints the greatest of its three inputs. There is a test set T for P, presented in Table 2. The tester desires to evaluate how good T is. GPTesT helps it in this task. The tester must follow the evaluation procedure described next. Observe that its two first steps are the same steps as the selection procedure.

1. GPTesT initialization.
2. Generation of alternatives.
3. Addition of all test cases from T.
4. Execution of P and of the alternatives using the available test cases.
5. Determination of equivalent alternatives.
6. Analysis of the score for T. The final results for T is in Figure 9

According to the tester's goals, T can be considered good "enough" and the testing activity ends. The evaluation procedure is also used to compare two test cases sets. For example, we can consider that the greater the score the better the set.

Test Case: 1)
Dead Alternatives: 9
Test Case: 2)
Dead Alternatives: 99
Test Case: 3)
Dead Alternatives: 2
Test Case: 4)
Dead Alternatives: 0
Test Case: 5)
Dead Alternatives: 0
Test Case: 6)
Dead Alternatives: 2
Execution Time: 00:01:08h

Total Number of Alternatives: 127
Anomalous Alternatives: 0
Live Alternatives: 15
Equivalent Alternatives: 0
Number of Test Cases: 6
Coverage Score: 0.88189

Figure 9: Status obtained for the test set T

When incorrect outputs of P are obtained, we have to remove the fault and continue the procedure being conducted. When we test a program, we usually follow the two procedures. We can perform the evaluation procedure with a functionally or randomly derived test set T and after this, we start the selection procedure on Step 3, to get the desired score.

Table 2: Test Case Set T

Number	a	b	c
1)	0	1	2
2)	1	2	0
3)	1	0	2
4)	4	5	6
5)	5	6	4
6)	6	4	5

6 CONCLUSIONS

This work presented a framework, named GPTesT, to support the use of Genetic Programming (GP) in the software testing activity. GPTesT allows the use of a new approach to fault-based testing.

The traditional approaches and tools are usually based on mutation operators. An operator is used to generate an alternative program that differs from the program under testing by a simple modification. GPTesT permits the alternative selection by using Chameleon, a GP-based tool. The alternatives do not necessarily differ from the original program by only one modification, and this permits to test interactions among faults, and to reveal other kind of faults than those reveled by the mutation operator approach.

The code of the program under testing is not used to derive the alternatives. This is an advantage during the maintenance phase. All alternatives continue valid. The user decides whether other alternatives will be generated. For the operator mutation approach and structural testing criteria, all the required elements must be generated again since they use the code to establish the testing requirements.

This work presents examples, showing that GPTesT supports two test procedures: test data set evaluation and selection. These procedures are a basic requirement, supported by most testing and criteria tools.

In spite of GPTesT helps the test of C programs and interacts with Chameleon, the GP approach implemented by GPTesT is independent on the used language. GPTesT implementation also permits future extensions. A possible extension is to generate alternatives using other GP tools that evolve programs written in other languages or paradigms. We intend to extend GPTesT to deal with Lisp programs.

Similar to other testing tools found in literature, GPTesT has some limitations. This happens due to the undecidibility related to the equivalence between programs and to the generation of test cases. However, in a future work we will extend GPTesT with mechanisms to reduce these limitations. The mechanisms are heuristics to determine equivalent alternatives and genetic algorithms to automatically generate test cases, helping the tester during the procedures exemplified in this paper.

References

[1] *Proceedings of Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers, 2000.

[2] F. Bergadano and D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering*. MIT Press, 1995.

[3] T. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, Vol. 18(1):31–45, November 1982.

[4] I. Chung. Automatic testing generation for mutation testing using genetic operators. In *Proceedings of SEKE*. San Francisco, June 1998.

[5] W. Craft. *Detecting Equivalents Mutants Using Compiler Optimization*. Master's Thesis, Department of Computer Science, Clemson University, Clemson-SC, 1989.

[6] C. Darwin. *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*. 1859.

[7] R. De Millo, D. Gwind, and K. King. An extended overview of the mothra software testing environment. In *Proc. of the Second Workshop on Software Testing, Verification and Analysis*, pages 142–151. Computer Science Press, Banff - Canada, July 19-21 1988.

[8] R. De Millo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, Vol. C-11:34–41, April 1978.

[9] R. De Millo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, Vol. SE-17(9):900–910, September 1991.

[10] M. E. Delamaro and J. Maldonado. A tool for the assesment for test adequacy for c programs. In *Proceedings of the Conference on Performability in Computing Systems*, pages 79–95. East Brunswick, New Jersey, USA, July 1996.

[11] M. Emer. *Seleção e Avaliação de Dados de Teste Baseadas em Programação Genética*. Master's Thesis, DInf - UFPR, Curitiba-PR, March 2002. (in Portuguese).

[12] F. Frankl. *The use of Data Flow Information for the Selection and Evaluation of Software Test Data*. PhD Thesis, Department of Computer Science, New York University, New York, U.S.A., October 1987.

[13] J. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[14] W. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, Vol. SE-8(4):371–379, July 1982.

[15] J. Koza. *Genetic Programming: On the Programming of Computers by Natural Slection*. MIT Press, Cambridge, MA, 1992.

[16] J. Maldonado, M. Chaim, and M. Jino. Briding the gap in the presence of infeasible paths: Potential uses testing criteria. In *XII International Conference of the Chilean Science Computer Society*, pages 323–340. Santiago, Chile, October 1992.

[17] G. McGraw and C. Michael. *Automatic Generation of test-cases for software testing*. Technical Report RST Corporation, 1997.

[18] C. Michael and et al. *Genetic Algorithms for Dynamic Test-Data Generation*. Technical Report RST-003-97-11 Corporation, 1997.

[19] C. Michael, G. McGraw, and M. Schatz. Generating software test data by evolution. *IEEE Trans. on Soft. Engin.*, Vol 27(12):1085–1110, Dec. 2001.

[20] L. J. Morell. Theoretical insights into fault-based testing. In *Proc. of Workshop on Software Testing, Verification and Analysis*, pages 45–62. Banff, Canada, 1988.

[21] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[22] A. Offut. The coupling effect: Fact or fiction? In *Proc. of Workshop on Software Testing, Verification and Analysis*, pages 131–140. 1989.

[23] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New-York, EUA, third edition, 1992.

[24] S. Rapps and E. Weyuker. Data flow analysis techniques for test data selection. In *Proceedings of International Conference on Software Engineering*. Tokio - Japan, September 1982.

[25] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

[26] E. Spinoza and et al. Chameleon: A generic tool for genetic programming. In *Proceedings of the Brazilian Computer Society Conference*. Fortaleza, Brazil, August 2001.

[27] H. Ural and B. Yang. A structural test selection criterion. *Information Processing Letters*, 28(3):157–163, July 1988.

[28] E. Weyuker. Assessing test data adequacy through program inference. *ACM Trans. on Programming Languages and Systems*, Vol. SE-5(4):641–655, 1983.

A New Representation and Crossover Operator for Search-Based Optimization of Software Modularization

Mark Harman and Robert Hierons
Brunel University,
Uxbridge, Middlesex, UB8 3PH.
{Mark.Harman,Rob.Hierons}@brunel.ac.uk

Mark Proctor
CISCO Systems, 250 Longwater Avenue,
Reading, UK.
Mark.proctor@bigfoot.com

Abstract

This paper reports experiments with automated software modularization and re-modularization, using search-based algorithms, the fitness functions of which are derived from measures of module granularity, cohesion and coupling. The paper introdeuces a new representation and crossover operator for this problem and reports initial results based on simple component topologies.

1 INTRODUCTION

It is well established in the software engineering community that good modularization of software leads to system which are easier to design, develop, test, maintain and evolve. Given a set of program components, there are many ways in which the module boundaries can be drawn, each of which corresponds to a different 'modularization' of the software. The problem is a graph partitioning problem, which is known to be NP hard and therefore seems suited to a metaheuristic search-based approach.

Macoridis et al. [15] showed that the problem of modularizing software can be reformulated as a search problem. Initially, they used an exhaustive search and hill climber [15], but later experimented with a simple genetic algorithm [4] to search the space of possible modularizations. Mitchell [16] provides a survey of work on modularization together with experience using exhaustive search, hill climbing and genetic algorithms. Mitchell reports that exhaustive search becomes impractical for networks of more than 15 components. He describes the Bunch [14] a modularization tool which uses a hill climbing approach to implement search based modularization. The Bunch tool uses hill climbing, rather than a genetic algorithm, because it was found that the hill climber produced more consistently high quality results [16]. Mitchell indicates that the genetic approach requires more work.

This paper attempts to further explore the application of genetic algorithms to the problem, and makes two modest contributions to the application of genetic algorithms to the modularization problem.

- We introduce a new representation which allows only one representation per modularization.
- We introduce a new crossover operator which attempts to preserve building blocks.

In our work, we were principally concerned with the problem of reverse engineering a system whose modularization has degraded as the system is maintained. For such a system some components may no longer be in suitable modules and re-modularization of the system might be appropriate. The granularity of a modularization is the number of modules it uses. Our problem is therefore to search the space of possible modularizations around the current granularity to see if there exists a better allocation of components to modules.

The rest of the paper is organized as follows. Section 2 describes the approach adopted to formulating modularization as a search problem. Section 3 presents the results of applying the genetic algorithm to example modularization problems. Section 4 presents related work and Section 6 concludes.

2 REPRESENTATION, FITNESS AND OPERATORS

The starting point for the application of search-based techniques to software engineering is the definition of a suitable representation, fitness function and operators

[7]. This section introduces a new representation for the allocation of components to modules and a new crossover operator. We also describe our approach to the definition of the fitness function, as this differs from that used in previous work [15, 4, 16].

2.1 REPRESENTATION

The first problem which presents itself when attempting to formulate a software engineering problem as a search-based problem, is that of representation. In the case of modularization there is a need to identify each possible way of modularizing a system in a *unique* way so that there is only one representation per modularization. Non-unique representations of modularizations artificially increase the search space size, inhibiting search-based approaches to the problem.

The approach we adopted was to normalize the representation in the following way: Modules are numbered, and elements allocated to module numbers using a simple look-up table. Component number one is always allocated to module number one. All components in the same module as component number one are allocated to module number one. Next, the lowest numbered component, n, not in module one, is allocated to module number two. All components in the same module as component number n are allocated to module number two. This process is repeated, choosing each lowest numbered unallocated component as the defining element for the module.

This representation must be renormalized when components move as the result of mutation and crossover, but it has the significant advantage that a particular allocation of components to modules has but one single representation.

2.2 FITNESS FUNCTION

Following Constantine and Yourdon [2], approaches to modularization typically attempt to maximize cohesion and minimize coupling in line with software engineering principles which indicate that this leads to good quality results. Constantine and Yourdon defined seven levels of coupling and seven levels of cohesion. These seven levels of cohesion provide a qualitative measure of a systems overall cohesion and coupling. Unfortunately the levels are of little use as an input to a fitness function as they are too subjectively defined. Lakhotia [10] formalised the seven levels within a dependency analysis framework. This work allows the levels to form the input to a fitness function. However, Lakhotia's measure would yield only a seven point scale, resulting in a fitness landscape which would be coarse and would, therefore, be inappropriate for a search-based solution.

In this paper cohesion and coupling will be measured simply in terms of dependencies between the components of a module. The module's components will be assumed to be a set of procedures, functions and variables. Dependence arises between procedure (or function) p and procedure (or function) p' iff p calls p'. Similarly, a dependence (or association) arises between a procedure (or function) p and variable v iff p reads from or writes to v.

The problem of finding good modularization is therefore a graph theoretic problem of finding subgraphs with the maximum connectivity (cohesion), with the minimal association between subgraphs (coupling), and for a desired number of identified subgraphs (target granularity).

Cohesion for a network is measured as the average number of associations per module with respect to the maximum possible number of associations. More formally, let $\mathcal{A}$ be a function from modules to the number of associations within the module. Let N be a function from a module to the number of components in the module and let K be the number of modules in the network. The cohesion $C(m)$, of a module m is defined as follows:

$$\mathrm{C}(m) = \begin{cases} 1 & \text{if N(m)} = 1 \\ \frac{\mathcal{A}(m)}{\mathcal{N}(m).(\mathcal{N}(m)-1)} & \text{otherwise} \end{cases}$$

The cohesion of the system, S containing K modules is defined as follows:

$$\mathrm{Cohesion(S)} = \frac{\sum_{m \in S} C(m)}{K}$$

This value is renormalized to a percentage (by multiplying by 100), so that 100% indicates that all components are related to all others within their own module.

The coupling *unfitness*[1], of the system is expressed as the total number of inter-module associations divided by the total number possible for the network. Coupling fitness is simply the inverse of coupling unfitness. Once again this is renormalized to a percentage, where 100% indicates that there is no coupling between modules.

In order to capture the additional requirement that the modularization produced has a granularity not too dissimilar to the current granularity, a polynomial punishment factor was introduced into the fitness function,

[1] recall that coupling is considered to be bad, so it is to be minimized, hence 'unfitness'.

to reward solutions polynomially as they approach the target value for granularity of the modularization.

The aim is to allow some deviation from the target granularity, where this can allow dramatic improvement in cohesion and coupling values for the overall system, but to encourage the search to consider solutions on and around the target granularity.

The granularity component of the fitness function is calculated in terms of the actual granularity of the modularization AG and the target granularity TG. Once again this is normalized to a percentage, so that 100% represents the situation where the actual and target values of granularity are identical. The value of the actual granularity is allowed to range from 0 to twice the target granularity, which each of these extreme values scoring zero fitness for the granularity component of the fitness function.

The three fitness components: cohesion, coupling and granularity are each given equally weight in computing the overall fitness of the system.

2.3 CROSSOVER

To attempt to promote the formation, retention and propagation of good building blocks [5, 23] within the genetic algorithm, a crossover operator was defined which attempts to preserve partial module allocations from parents to children.

Rather than selecting an arbitrary point of crossover within the two parents, an arbitrary parent is selected and one of its arbitrarily chosen modules is copied to the child. This results in a partial allocation of components to modules in the child. The components allocated are removed from both parents. This removal can be thought of as a form of 'pre-emptive repair' as it prevents duplication of components in the child when further modules are copied from one or other parent to the child.

The process of selecting a module from a parent and copying to the child is repeated and the components copied are removed from both parents until the child contains a complete allocation (that is, when both parents have no modules left to copy).

This approach ensures that at least one (randomly chosen) module from the parents is preserved (in entirety) in the child, and that parts of other modules will also be preserved.

A standard genetic algorithm was implemented with single point crossover, to allow comparison with the novel crossover operator.

Mutation was set to an unusually low value (after crossover, an individual chromosome had only a 5% chance of mutation). The population size was also relatively low at 30 individuals, in keeping with prior work [16]. This allows the possible effects of the crossover operator to dominate our results, facilitating a comparison of the novel and standard crossover techniques.

3 RESULTS

Figures 1 and 2, show the results of applying the two genetic algorithms and hill climbing against a random search baseline. The labellings are 'Random' for the random search, 'HC' for the Random Mutation Hill Climbing search, 'GA' for the simple genetic algorithm with standard single point crossover and 'GA+' for the genetic algorithm with the novel crossover operator. Results are averaged over five separate runs for each search algorithm to account for random effects.

The left hand section of the figure shows the parameters for the problem and illustrates the components and their associations, depicted in a manner which suggests the 'ideal' modularization (namely, that which maximizes fitness).

The right-hand section of the figure shows the corresponding results for the three search-based algorithms, together with random search (as a base-line performance). The graphs plot average fitness (over five runs) against generation number. Recall that fitness is denoted by a percentage, where 100% is the maximum fitness obtainable for a 'perfect' modularization. A perfect modularization has reached exact target granularity, no associations between modules and has every component within a module related to every other component. Such a perfect fitness may be prohibited by the structure of the associations between components and so 100% is not reached by any of the search algorithms in some cases.

Figure 1 shows the behaviour of the search algorithms when the target granularity is appropriate. That is, the target granularity is set to the number of modules in the 'ideal' modularization. Figure 2 shows the behaviour when the target granularity is misleading.

3.1 APPROPRIATE TARGET GRANULARITY

The results for these simple networks confirm that search-based algorithms outperform random search[2]

[2]In the most simple problem (at the top of the figure), the genetic algorithm with the novel crossover technique is

and that the gap between the search-based algorithms and random search increases with the size of the problem.

As expected, GA+, the genetic algorithm with the novel crossover operator outperforms the simple genetic algorithm, GA, which uses a single point crossover operator. However, the novel crossover quickly becomes trapped by a local optimum.

For these simple networks, hill climbing outperforms all other techniques. This is not surprising since the genetic algorithms are focused upon the use of crossover and allow very little mutation.

3.2 MISLEADING TARGET GRANULARITY

The results for a selection of simple problems, where the value of target granularity is set to a misleading value were also collected and are depicted in Figure 2. These results still show that the hill climber and simple genetic algorithm out-perform random search and that hill climbing is far superior, but they indicate a worse performance for the GA+ search algorithm which employs the novel crossover technique. It is worth noting that that GA+ still performs well where the modules are very clearly defined by the associations (the third case).

These results suggest that GA+ is more sensitive to inappropriate choices of target granularity than any of the other approaches. This sensitively can be exploited, because it will suggest more radical re-modularization for very badly degraded, heavily maintain software, where a complete repartitioning of the system will produce better results than merely moving a few components between modules, or perhaps adding or removing a module or two.

4 RELATED AND FUTURE WORK

The work reported here is most closely related to work on the Bunch tool, by Macoridis et al. [15, 4, 16, 14], who introduced the search-based approach to software modularization. Macoridis et al. use a standard genetic algorithm with single point crossover and a representation which allocates a module number to each component. This representation has the drawback that it allows many representations of a single modularization, for example the strings (1,1,2,1,2,3,3), (3,3,1,3,1,2,2) and (2,2,3,2,3,2,2) all represent a modularization consisting of three modules, which places components 1,2 and 4 in one module, components 3 and 5 in another module and which places components 6 and 7 in a third module.

The principal difference between our work and that of Macoridis et al. lies in the novel crossover technique and the instruction of a normalized representation for the modularization problem.

A related problem of hierarchical decomposition of software is considered by Lutz [13]. Lutz is concerned with the problem of decomposition of software into hierarchies at different levels of abstraction, whereas the present work is concerned with only a single level of abstraction (the implementation level). Lutz therefore considers designs rather than code. However, there is no reason, in principle, why the approach adopted by Lutz could not be also applied to the modularization problem considered in the present paper.

The approach adopted by Lutz differs strongly from the approach adopted in the present paper with regard to the choice of fitness function. The fitness function used by Lutz is based upon an information-theoretic formulation inspired by Shannon [19]. The function awards high fitness scores to hierarchies which can be expressed most simply (in information theoretic terms), with the aim of rewarding the more 'understandable' designs. Such a fitness function is possibly more semantic than the comparatively structural approaches adopted in the present paper and in the work of Macoridis et al. More work is required to compare the results produced by these two approaches.

Other work on software re-modularization has adopted analytical solutions based upon formal concept analysis and clustering metrics [9, 21, 11] and sets of heuristic rules [18]. More work is required to assess the comparative performance of these non search-based approaches with the search-based strategy introduced here.

Some metrics for coupling and cohesion [20, 17, 1, 8] have attempted to give a more 'continuous' real-valued quantitative metric based on a variety of criteria, derived from program slicing [3, 22, 6]. Using these metrics, it is possible to allocate weights to associations between components. This would allow the search to be more attuned to the relative impact of a particular association between modules. This would be particularly useful in systems where there are many associations, and so clustering based merely upon the presence or absence of an association becomes relatively arbitrary.

In order to exploit association weights, a search based clustering algorithm such as the GGA [12] is required.

worse than random, but in all other cases all search algorithms are better than random.

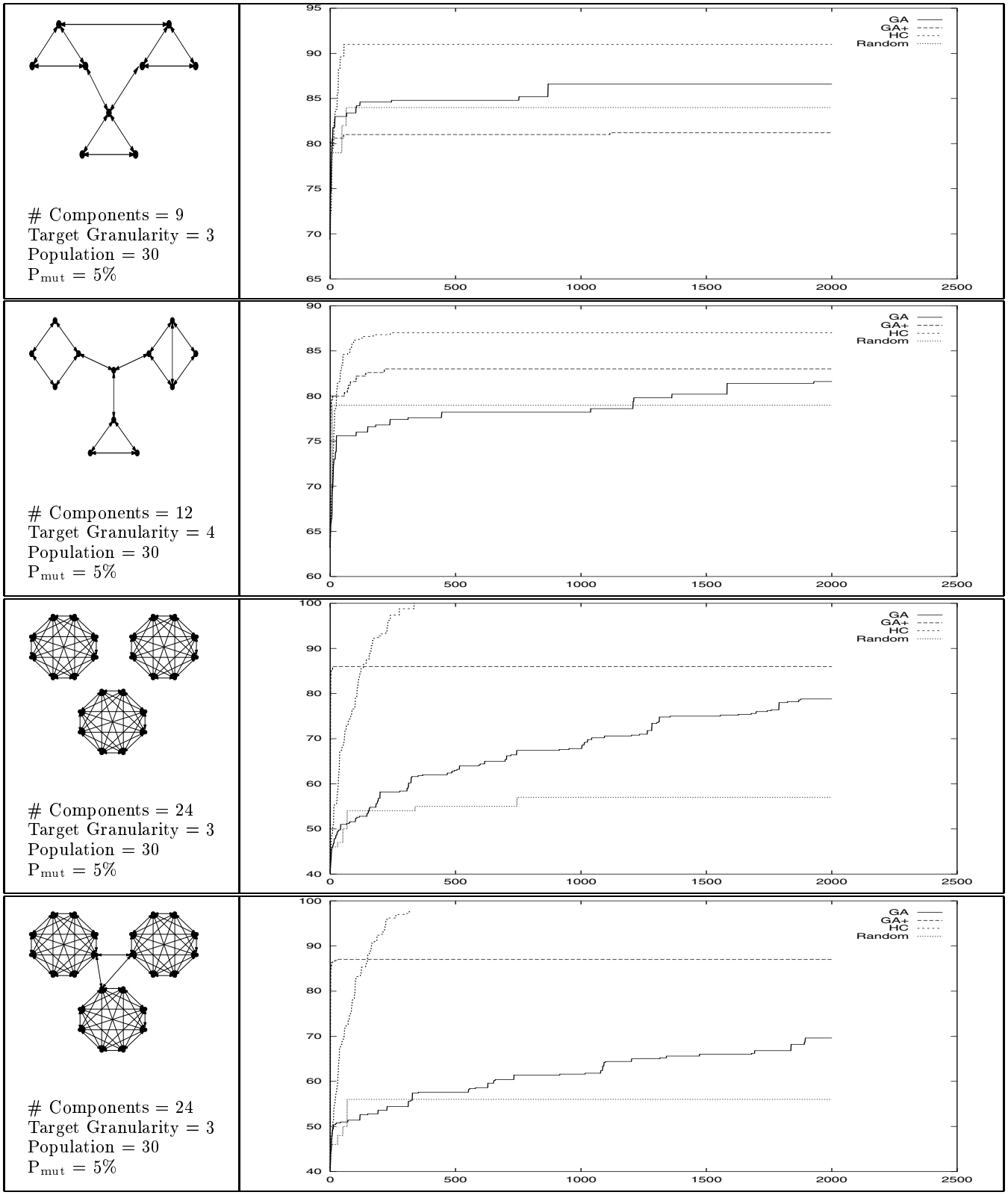


Figure 1: Guiding Value for Target Granularity

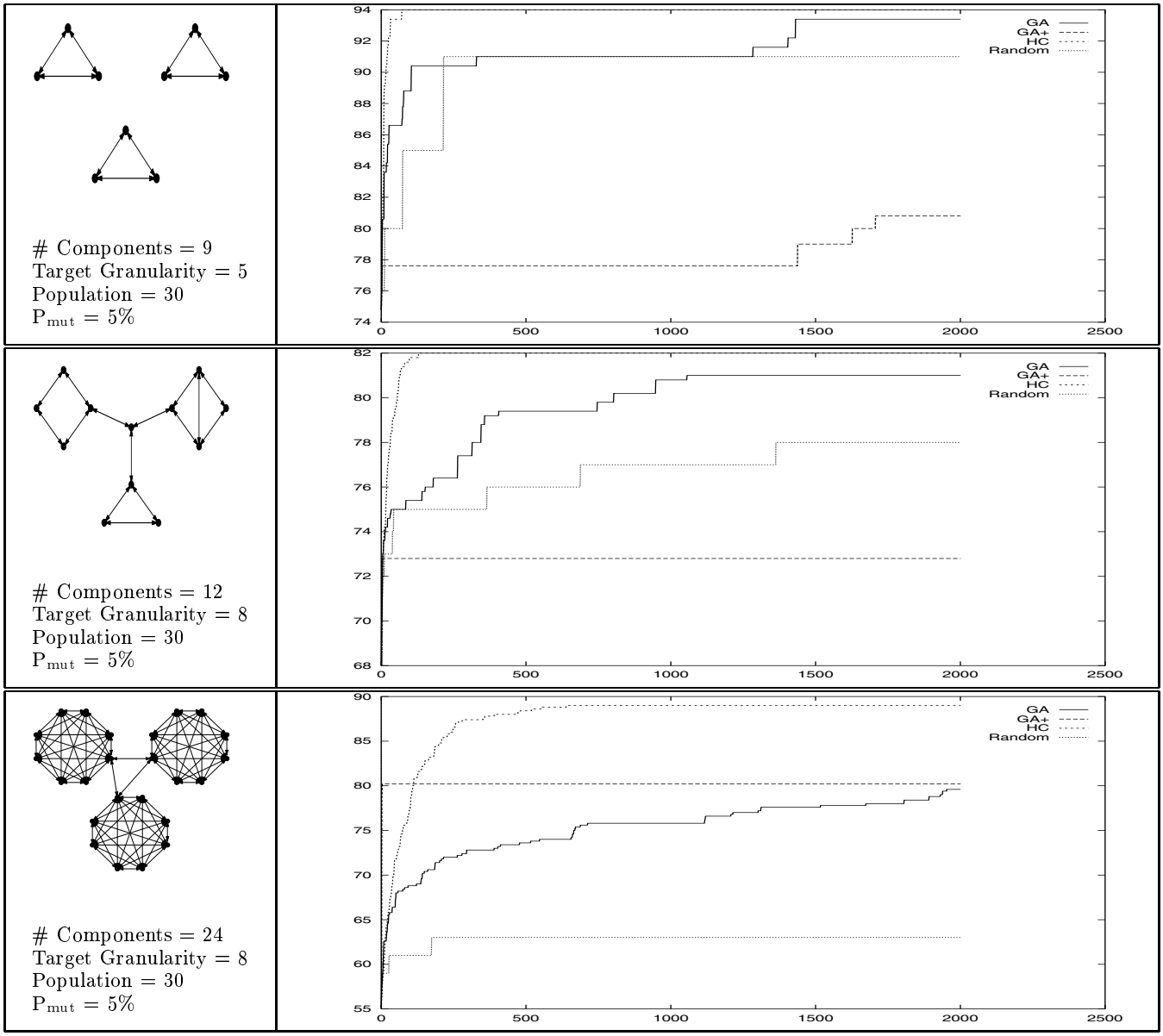


Figure 2: Misleading Value for Target Granularity

The GGA clusters n-dimensional nodes according to the internode distance in the n dimensional space. The modularization problem is a special case of this, where the nodes are components and the distance is the weight of association. In the present paper this 'weight' has simply been 0 (indicating no association) or 1 (indicating the presence of an association). Using the GGA and slice-based measurement of cohesion and coupling, a more 'impact sensitive' approach can be pursued. The authors intend to explore this possibility in future work.

5 ACKNOWLEDGEMENTS

The authors benefitted greatly from discussion of earlier versions of this work with members of the EPSRC-funded SEMINAL network. In particular, Vic-Rayward Smith (at the University of East Anglia, UK) pointed out the importance of a normalized representation. The authors would also like to thank Stephen Swift, Xiaohui Liu and Allan Tucker (at Brunel) for helpful comments on this work and for suggesting and offering the use of the GGA in future work and Spiros Mancoridis and Brian Mitchell (at Drexel) for detailed discussion about their BUNCH system.

6 CONCLUSION

This paper makes a modest contribution to work on the software modularization problem. Previous work in this area has either not used search-based techniques at all, or has largely been concerned with hill-climbing and exhaustive searches.

We introduce a normalized representation for a software modularization, which will reduce the size of the search space and may improve results for genetic algorithms which are known to perform poorly where there is a many-to-one mapping from genotype to phenotype. We also suggest a new crossover operator which is designed to promote the formation and retention of building blocks. Initial work suggests that this crossover technique may be better suited to genetic approaches than standard crossover.

References

[1] Bieman, J. M., and Ott, L. M. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20*, 8 (Aug. 1994), 644–657.

[2] Constantine, L. L., and Yourdon, E. *Structured Design*. Prentice Hall, 1979.

[3] De Lucia, A. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation* (Florence, Italy, 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 142–149.

[4] Doval, D., Mancoridis, S., and Mitchell, B. S. Automatic clustering of software systems using a genetic algorithm. In *International Conference on Software Tools and Engineering Practice (STEP'99)* (Pittsburgh, PA, 30 August - 2 September 1999).

[5] Goldberg, D. E., and Sastry, K. A practical schema theorem for genetic algorithm design and tuning. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)* (San Francisco, California, USA, 7-11 July 2001), L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds., Morgan Kaufmann, pp. 328–335.

[6] Harman, M., and Hierons, R. M. An overview of program slicing. *Software Focus 2*, 3 (2001), 85–92.

[7] Harman, M., and Jones, B. F. Search based software engineering. *Information and Software Technology 43*, 14 (Dec. 2001), 833–839.

[8] Harman, M., Okunlawon, M., Sivagurunathan, B., and Danicic, S. Slice-based measurement of coupling. In 19^{th} *ICSE, Workshop on Process Modelling and Empirical Studies of Software Evolution* (Boston, Massachusetts, USA, May 1997), R. Harrison, Ed.

[9] Hutchens, D., and Basili, V. System structure analysis: clustering with data bindings. *IEEE Transactions on Software Engineering SE-11*, 8 (1985), 749–757. The use of cluster analysis as a tool for system modularization is examined. It appears that the clustering of data bindings provides a meaningful view of system modularization.

[10] Lakhotia, A. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)* (1993), pp. 34–44.

[11] Lindig, C., and Snelting, G. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 1997 International Conference on Software Engineering* (1997), ACM Press, pp. 349–359.

[12] Liu, X., Swift, S., and Tucker, A. Using evolutionary algorithms to tackle large scale grouping problems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)* (San Francisco, California, USA, 7-11 July 2001), L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds., Morgan Kaufmann, pp. 454–460.

[13] Lutz, R. Evolving good hierarchical decompositions of complex systems. *Journal of Systems Architecture 47* (2001), 613–634.

[14] Mancoridis, S., Mitchell, B. S., Chen, Y., and Gansner, E. R. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings; IEEE International Conference on Software Maintenance* (1999), IEEE Computer Society Press, pp. 50–59.

[15] Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y., and Gansner, E. R. Using automatic clustering to produce high-level system organizations of source code. In *International Workshop on Program Comprehension (IWPC'98)* (Ischia, Italy, 1998), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 45–53.

[16] Mitchell, B. S. *A Heuristic Search Approach to Solving the Software Clustering Problem.* PhD Thesis, Drexel University, Philadelphia, PA, Jan. 2002.

[17] Ott, L. M., and Thuss, J. J. The relationship between slices and module cohesion. In *Proceedings of the* 11^{th} *ACM conference on Software Engineering* (May 1989), pp. 198–204.

[18] Schwanke, R. W. An intelligent tool for reengineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering* (May 1991), pp. 83–92.

[19] Shannon, C. E. A mathematical theory of communication. *Bell System Technical Journal 27* (July and October 1948), 379–423 and 623–656.

[20] Thuss, J. J. An investigation into slice–based cohesion metrics. Master's thesis, Michigan Technological University, 1988.

[21] van Deursen, A., and Kuipers, T. Identifying objects using cluster and concept analysis. Tech. Rep. SEN-R9814, Centrum voor Wiskunde en Informatica (CWI), Sept. 1998.

[22] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

[23] Wu, A. S., and Lindsay, R. K. A comparison of the fixed and floating building block representation in the genetic algorithm. *Evolutionary Computation 4*, 2 (1997), 169–193.

Improving Evolutionary Testing by Flag Removal

Mark Harman, Lin Hu and Robert Hierons
Brunel University,
Uxbridge, Middlesex, UB8 3PH, UK
Mark.Harman@brunel.ac.uk

André Baresel and **Harmen Sthamer**
DaimlerChrysler AG, Research and Technology
Alt-Moabit 96a, D-10559 Berlin
Andre.Baresel@daimlerchrysler.com

Abstract

This paper argues that Evolutionary testing can be improved by transforming programs with flags into flag free programs. The approach is evaluated by comparing results from the application of the Daimler-Chrysler Evolutionary Testing System to programs with flags and their transformed flag-free counterparts. The results of this empirical study are very encouraging. Programs which could not be fully covered become fully coverable and the number of generations required to achieve full coverage is greatly reduced.

1 INTRODUCTION

Evolutionary testing generates test data to cover certain structural program features, using evolutionary algorithms to search the space of possible program inputs. Evolutionary testing has been shown to be an effective way of automatically generating test data for white box (or structural) test adequacy criteria [13, 21, 19, 15, 11]. The approach works well for well-behaved programs, but for certain programming language features the approach performs poorly.

One such problem arises with programs which use flag variables. A flag variable is one whose value is either `true` or `false`. Flags typically 'flag' the presence of some special condition if interest. The use of flag variables with current approaches to fitness function definition, yields a coarse fitness landscape with a single super-fit plateau and a single super-unfit plateau (corresponding to the two possible values of the flag variable). This causes the search to degenerate to a random search. Where the super-fit plateau is small, such a random search fails to find suitable test data, reducing the coverage achieved by the approach.

Embedded systems, such as engine controllers, typically make extensive use of flag variables to record state information concerning the devices controlled. Such systems can therefore be hard to test using evolutionary testing approaches to automated test data generation. This is a serious problem, since generating such test data by hand is prohibitively expensive, yet the correct operation of such embedded systems is clearly of paramount importance.

This paper presents a transformation-based approach, which addresses the problem. The approach allows certain forms of commonly arising flags to be transformed out of the program, thereby dramatically improving the results of evolutionary testing. The rest of the paper is organised as follows: Section 2 gives a brief overview of evolutionary testing, while Section 3 explains the flag problem. Section 4 describes our solution to the flag problem and Section 5 presents and discusses the results of applying this solution to typical flag-based programs.

2 APPLYING EVOLUTIONARY ALGORITHMS TO SOFTWARE TEST DATA GENERATION

Evolutionary testing designates the use of metaheuristic search methods for test case generation. The input domain of the test object forms the search space in which one searches for test data that fulfill the respective test goal. Due to the non-linearity of software (if-statements, loops etc.) the conversion of test problems to optimisation tasks mostly results in complex, discontinuous, and non-linear search spaces. Neighbourhood search methods like hill climbing are not suitable in such cases. Therefore, metaheuristic search methods are employed, e.g. evolutionary algorithms, simulated annealing or tabu search. In this work, evo-

lutionary algorithms will be used to generate test data, since their robustness and suitability for the solution of different test tasks has already been proven in preceding work [13, 21]. The only prerequisites for the application of evolutionary tests are an executable test object and its interface specification. In addition, for the automation of structural testing, the source code of the test object must be available to enable its instrumentation.

In order to automate software tests with the aid of evolutionary algorithms, the test goal must itself be transformed into an optimisation task. For this, a numeric representation of the test goal is necessary, from which a suitable fitness function for evaluation of the generated test data can be derived. Depending on which test goal is pursued, different fitness functions emerge for test data evaluation. For structural testing the fitness functions can be based on computation of a distance for each individual that indicates how far it is away from executing the program predicate in the desired way [13, 21, 19].

For example, if a branching condition `x==y` needs to be evaluated as `true`, then the fitness function may be defined as $|\,\mathtt{x}-\mathtt{y}\,|$ (provided that the fitness values are minimised during the optimisation). Each individual of the population represents a test datum with which the test object is executed. For each test datum the execution is monitored and the fitness value is determined for the corresponding individual.

The approach adopted in the work reported in the present paper, used the DaimlerChrysler Evolutionary Testing System. Multiple strategies and competitions between these were used. All experiments used a population of 300 individuals split into 6 subpopulations of 50 individuals. In order to combine the multiple strategies, migration was introduced to permit an exchange of the best individuals between subpopulations at regular intervals. The details of the implementation of the evolutionary approach to software testing are described elsewhere [13, 21, 19, 20]. In the present paper, the focus is upon the way in which transformation can be used to improve the behaviour of these established techniques.

3 THE FLAG PROBLEM

A flag variable will be taken to mean any variable, the type of which is boolean, but the transformations presented here may well extend to other variables which are assigned one of a small number of possible scalar values.

Generating test data using evolutionary testing [10, 20, 19, 15, 11] has been shown to be successful. However, evolutionary testing relies upon a fitness function which uses the predicate which controls a branch. Where such a predicate is simply a reference to a flag variable, the search has little information to guide it, making the evolutionary technique perform poorly. More precisely, using an evolutionary algorithm, the presence of flag variables (and unordered enumeration types in general) can create a coarse fitness landscape.

This reduces the effectiveness of the search. That is, the fitness landscape consists of two plateaus, corresponding to the two possible flag values. One of these plateaus will be super-fit and the other super-unfit. A search-based approach, such as evolutionary testing, will not be able to locate the super-fit plateau any better than a random search, because the fitness landscape provides no guide to direct the search from unfit to fit regions of the landscape. Where the fit plateau may be very small relative to the unfit plateau, this makes the program hard to test. A similar problem is observed with n–valued enumeration types, whose fitness landscapes contains n discrete values, as n becomes larger the program becomes progressively more testable, as the landscape becomes progressively more smooth and therefore, more guidance is available.

Figure 1 illustrates the transformation approach to the flag variable problem. The original program (in column (a)) is hard to test using currently defined fitness functions for evolutionary testing. The first dotted section indicates code which does not assign to `n`, the second dotted section of code does not assign to `flag`. Suppose `n` is an unsigned integer value. The value of `n` required to cause the second conditional to follow the `true` branch must be odd and less than four, namely it must be either 1 or 3. Random testing is very unlikely to 'stumble' across these two values, so a more intelligent search is required. This is where evolutionary testing could help. Unfortunately, the presence of the flag variable inhibits the search, because the fitness landscape is insufficiently smooth to guide the search. Therefore, it is difficult to cover both branches of the final `if` statement.

4 A TRANSFORMATION-BASED SOLUTION

A program transformation [5, 16, 17, 2] is a rule which defines the way in which a program can be modified. It can be thought of as a function from program syntax to program syntax. Some transformation rules have side conditions. These are conditions which must be true for the transformation to be correct. As a simple ex-

(a) Original	(b) Single flag assignment	(c) Independent Assignment	(d) Flag removed

`flag = n<4;`
`...`
`if (n%2==0) flag = 0;`
`...`
`if (a[i]!='0' && flag)`
`...`
	`...`
	`flag=(n%2==0)?0:(n<4);`
	`...`
	`if (a[i]!='0' && flag)`
	`...`
		`...`
		n' `= n;`
		`flag=(`n'`%2==0)?0:(`n'`<4);`
		`...`
		`if (a[i]!='0' && flag)`
		`...`
			`...`
			n' `= n;`
			`flag=(`n'`%2==0)?0:(`n'`<4);`
			`...`
			`if (a[i]!='0' &&`
			`(`n'`%2==0)?0:(`n'`<4))`
			`...`

Figure 1: Flag Removal

ample, consider the simple transformation rule 'reverse if' which reverses the branches of an `if-then-else` statement, and negates the predicate. This transformation produces an equivalent program while altering the structure of the original. Such a transformation will be denoted like this[1]:

$$\texttt{if } E \texttt{ then } S_1 \texttt{ else } S_2 \Rightarrow \texttt{if not(}E\texttt{) then } S_2 \texttt{ else } S_1$$

Many transformation rules are extremely simple. On their own they achieve little of value. However, when combined into sequences of transformations, or into mini-programs, called transformation tactics, the combined effect on the program under consideration can be startling. A set of transformation tactics is typically collected together into a transformation strategy; an algorithm for manipulating the subject program into a semantically equivalent, but more syntactically and structurally amenable form.

Transformation has been applied to many problems including automatic parallelization [12, 23] program comprehension [3, 18, 9], reverse and re-engineering [16] and efficiency improvement [1]. In this paper transformation will be used to remove the flag problem, by transforming predicates which contain flags into flag-free predicates.

Our approach to the flag problem will be to transform a flag-based program into an equivalent flag free version. The transformations we use will preserve the branches of the original program, so that test data will achieve branch coverage for the original program if and only if it does so for the transformed program. This allows us to replace the (harder) problem of generating test data for the flag-based program with the (easier) problem of generating test data for the transformed, flag-free version. Once we have generated the test data (using the transformed program) we have no further need of the transformed flag-free program, and it is discarded. This application of program transformation differs from conventional transformation in this regard: For us, transformation is a means to an end, rather than an end in itself. The transformed program is of use only to the evolutionary testing system and is never presented to the human.

Consider, the program from Figure 1. The version in column (d) is equivalent, but easier to test because the use of the flag variable `flag` has been replaced by an expression which denotes its value at the point of use. Thus, this version of the program produces a smoother fitness landscape at the second predicate, thereby guiding the search toward the two fittest values sought. Columns (b) and (c) show the intermediate transformation steps required to reach the result in column (d). In column (b) assignments to the flag variable have been collected together. In column (c) a temporary variable, n', is used to capture the current value of the variable `n`. Finally, in column (d) the expression denoting the value of `flag` is substituted for the single use which it reaches.

Unfortunately, space restrictions prevent a full treatment of the flag removal algorithm. Figure 2 presents a sketch of the algorithm. One transformation step that we found particularly useful is that known as 'program slicing' [22, 14, 6, 8], and in particular its amorphous formulation [7]. Slicing removes parts of the program which cannot affect a particular variable of interest. Slicing is useful in flag removal, because it allows us to isolate the code that captures the computation on the flag variable.

5 RESULTS

The DaimlerChrysler Evolutionary Testing system was used to generate test data for flag-based programs and these results were compared with those obtained from running the testing system with identical parameters on the transformed, flag-free versions of the programs.

[1] In general, this paper will adopt the convention that $A \Rightarrow B$ denotes the fact that code fragment A can be transformed to code fragment B.

In this section we present three indicative experiments, which illustrate various incarnations of the flag problem and the effect upon evolutionary test data generation of their removal. The figures show the results obtained on the left hand side against the relevant fragments of the corresponding programs on the right-hand side. The program fragments are shown to illustrate the particular flavour of flag problem considered. However, when using the system, the human need not be aware either of the flag-free version of the program, nor indeed of the evolutionary process itself. The user simply submits a program (possibly with flags) and obtains a set of optimised test data.

The results plot the coverage achieved (for six separate executions of the evolutionary testing systems) against the number of fitness evaluations. They are therefore a measure of effectiveness against effort.

A test goal consists of attempting to optimise test data to cover a particular branch. The coverage for each trial therefore increases in steps, as each test goal is satisfied. In all examples we present, a test set which achieves full branch coverage *exists* (there are no infeasible branches).

5.1 Triangle Classification Program

The classify triangle program is widely used as a benchmark in software testing. The program has three variables (`a`, `b` and `c`), which represent the side lengths of a figure. The goal of the program is to determine whether the three side lengths represent a triangle, and if they do, to categorise the triangle type.

Input values are `double` values within range -1000 to 20000 with a precision of 0.00001. This gives a search space of size of approximately 10^{27}. We experimented with two versions of the a 'Validity check' program and a 'Special Value' program. These two variants of the triangle program illustrate the range of difficulty introduced by flags from none (Validly Check) through to severe (Special Value). The results for each variant are shown in Figures 3 and 4.

In the 'Validity Check' variant, the flag is assigned a value which represents a set of validity checks on inputs. There are many sub-criteria (boolean terms), many inputs which satisfy each sub-criteria and many which fail to satisfy each. Therefore, the fitness landscape does not contain a small high fitness plateau. Furthermore, each of the sub-criteria is also checked later on in the program by a separate conditional and so each sub-criteria also forms a separate test goal. In this situation the presence of flags presents no difficulty.

By contrast the 'Special Value' variant of the triangle program represents the *worst* form of flag-based program. The flag variable is set to true by only very few inputs, creating a tiny plateau of high fitness. Furthermore, the sub-conditions mentioned in the boolean expression assigned to the flag variable are not tested anywhere else in the program. In such a situation evolutionary testing degenerates to random testing.

The results show that for the 'Validity Check' version of the program, the removal of flags makes practically no difference, with all trials reaching maximum fitness, and with all doing so with a similar spread of effort. On the other hand, the 'Special Value' variant shows how bad the flag problem can be. After 40,000 fitness evaluations, none of the trail runs has risen above a coverage of .86 and after 120,000 evaluations none has risen above 0.92. No trial reached the maximum possible fitness. However, for the flag free version, after only 25,000 fitness evaluations, *all* of the trail runs has reached a coverage of more than 0.86 and after only 80,000 evaluations all have reached maximum possible coverage (1.0).

5.2 Calendar Program

The calendar program computes dates, but takes account of special days and date corrections which have taken place throughout the centuries. These special dates are denoted by flags in the program.

In 1751, the British Parliament passed

> "An Act for Regulating the Commencement of the Year, and for Correcting the Calendar Now In Use."[4]

The act became known as the 'Calendar Act'. One of the aspects of this act was that the date of September the 2^{nd}, the following year was to be immediately followed by September the 14^{th}. An decision which caused much consternation and a demand for the return of the 'stolen 11 days'. These stolen days form a special case in the calendar program which is denoted by a flag.

The calendar program is a typical flag-based program which tests for an 'unusual' condition and sets the value of a flag according to this test. This is typical because flags often test for exceptional cases. That is, the value assigned is far more likely to take one of the two possible values than the other (because the condition tested is 'unusual'). In this case, the program contains 10 character variables, which take values within range 0 to 10. This gives a search space of approximately 10^{10}, with a flag representing the 11 stolen days.

The results of evolutionary test data generation for the calendar program, together with the relevant fragments of code are shown in Figure 5. The flag-free code has been simplified for readability. The actual transformed program produced by the flag-removal algorithm contains many temporary variables. Of course, the fact that the transformed version has poor readability is not an issue for this work (unlike most work on transformation) because the transformed program is not read by a human.

The results show that flag removal helps in this instance, because all of the runs achieve the maximum possible fitness using the flag-free version, while none does so using the original program. It can also be seen from the growth of coverage for each run, that using the flag-free version we obtain higher coverage, faster than using the flag-based version.

6 CONCLUSION

This paper has introduced a transformation-based approach which improves evolutionary testing in the presence of flag variables. Flag variables inhibit the successful application of evolutionary techniques to automated structural software test data generation. The transformation based approach, removes the reliance upon flag variables, thereby improving both the time to produce test data and the coverage achieved.

Flag variables are very common in embedded systems. The correct behaviour of these systems is also a paramount concern because they control real-world devices, the failure of which can lead to severe consequences. The results presented show that flag removal works well, when applied to typical flag-based programs.

7 ACKNOWLEDGEMENTS

The authors benefitted greatly from discussion of this work with members of the EPSRC-funded SEMINAL network.

References

[1] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, techniques and tools.* Addison Wesley, 1986.

[2] Baxter, I. D. Transformation systems: Domain-oriented component and implementation knowledge. In *Proceedings of the Ninth Workshop on Institutionalizing Software Reuse* (Austin, TX, USA, Jan. 1999).

[3] Bennett, K., Bull, T., Younger, E., and Luo, Z. Bylands: reverse engineering safety-critical systems. In *IEEE International Conference on Software Maintenance* (1995), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 358–366.

[4] Calendar Act. Calendar Act, Anno vicesimo quarto George II, cap. xxiii., 1751.

[5] Darlington, J., and Burstall, R. M. A tranformation system for developing recursive programs. *J. ACM 24*, 1 (1977), 44–67.

[6] De Lucia, A. Program slicing: Methods and applications. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation* (Florence, Italy, 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 142–149.

[7] Harman, M., and Danicic, S. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprenhesion (IWPC'97)* (Dearborn, Michigan, USA, May 1997), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 70–79.

[8] Harman, M., and Hierons, R. M. An overview of program slicing. *Software Focus 2*, 3 (2001), 85–92.

[9] Harman, M., Hu, L., Zhang, X., and Munro, M. Side-effect removal transformation. In 9^{th} *IEEE International Workshop on Program Comprehension (IWPC'01)* (Toronto, Canada, May 2001), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 310–319.

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

[11] Pargas, R. P., Harrold, M. J., and Peck, R. R. Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability 9* (1999), 263–282.

[12] Ryan, C., and Walsh, P. The evolution of provable parallel programs. In *Genetic Programming 1997: Proceedings of the Second Annual Conference* (Stanford University, CA, USA, 13-16 July 1997), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds., Morgan Kaufmann, pp. 295–302.

[13] Sthamer, H. *The Automatic Generation of Software Test Data Using Genetic Algorithms.* PhD thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.

[14] Tip, F. A survey of program slicing techniques. Tech. Rep. CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam, 1994.

[15] Tracey, N., Clark, J., and Mander, K. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)* (January 1998), IFIP, pp. 169–180.

[16] Ward, M. Reverse engineering through formal transformation. *The Computer Journal 37*, 5 (1994).

[17] Ward, M., and Bennett, K. A practical program transformation system. In *Working Conference on Reverse Engineering* (Baltimore, MD, USA, May 1993), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 212–221.

[18] Ward, M., Calliss, F. W., and Munro, M. The maintainer's assistant. In *Proceedings of the International Conference on Software Maintenance 1989* (1989), IEEE Computer Society Press, Los Alamitos, California, USA, p. 307.

[19] Wegener, J., Baresel, A., and Sthamer, H. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms 43*, 14 (2001), 841–854.

[20] 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.

[21] Wegener, J., Sthamer, H., Jones, B. F., and Eyres, D. E. Testing real-time systems using genetic algorithms. *Software Quality 6* (1997), 127–135.

[22] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

[23] Williams, K. P. *Evolutionary Algorithms for Automatic Parallelization.* PhD thesis, University of Reading, UK, Department of Computer Science, Sept. 1998.

The essential aim of the algorithm, is to reduce a program with flags into one with a single assignment to the flag variable, which can be substituted for the use, within a predicate under test. For example:-

```
flag = a==0;                                        flag = a==0;
  .                                                   .
  .  /* no assignments to flag or a */      ⇒        .  /* no assignments to flag or a */
if(flag) ...                                        if(a==0) ...
```

Where there is a single assignment, which cannot be substituted because of the presence of intervening assignments to other variables needed by the definition of the flag variable, temporary variables are used to reduce the problem to that previously considered. For example:-

```
                          Ta = a;                   Ta = a;
flag = a==0;              flag = a==0;              flag = a==0;
  .                         .                         .
  .                         .                         .
a=a+1;            ⇒       a=a+1;            ⇒       a=a+1;
  .                         .                         .
  .                         .                         .
if(flag) ...              if(flag) ...              if(Ta==0) ...
```

Where there are multiple assignments, these are gathered together into a single assignment, which can then be handled by the approach above. For example:-

```
x = y+1;                  x = y+1;                      x = y+1;
y = x*2;                  y = x*2;                      y = x*2;
flag = x>y;          ⇒    flag = x>y;             ⇒     flag=x>y || (y+x>y) ==0;
y = y + flag;             flag=flag || (y + flag) ==0;  x = y*x;
flag=flag || y ==0;       x = y*x;
x = y*x;
```

Where the multiple assignments to the flag variable occur on different branches in an acyclic control flow graph which defines the flag variable, these are transformed into conditional assignments, which can then be treated using the approach above.

Figure 2: Sketch of the Flag Removal Algorithm

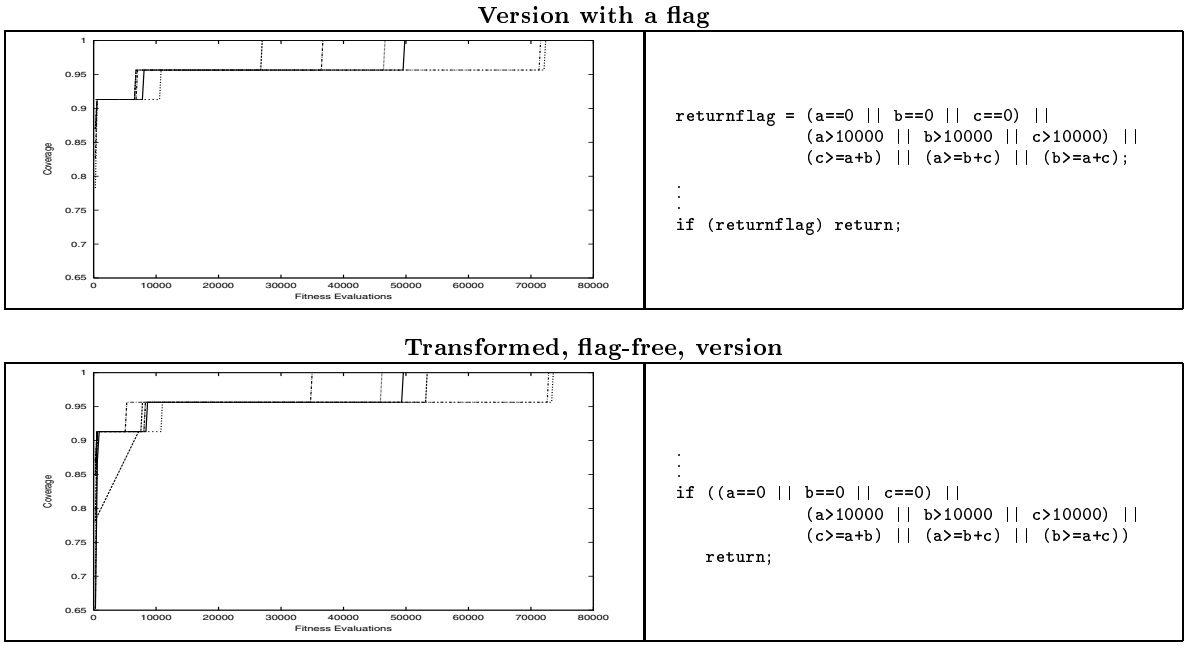
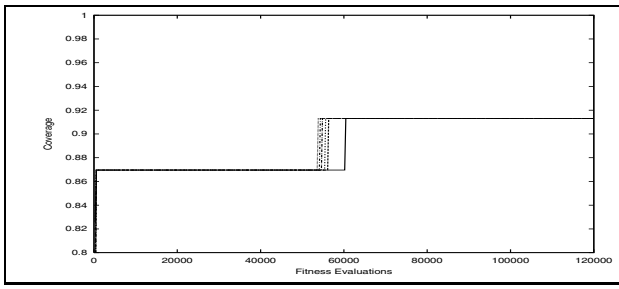


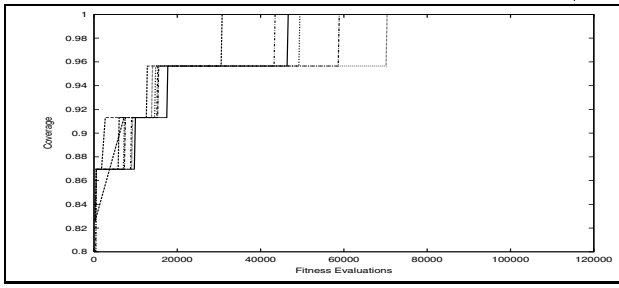
Figure 3: Results for the 'Validity Check' Triangle program

Version with a flag




```
returnflag = (a==99999 || b==99999 || c==99999) ;
.
.
.
if (specialflag) return;
```

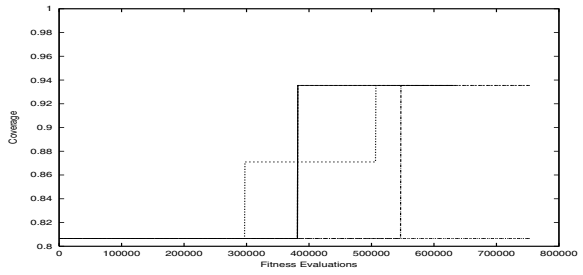
Transformed, flag-free, version




```
.
.
.
if ( (a==99999 || b==99999 || c==99999))
   return;
```

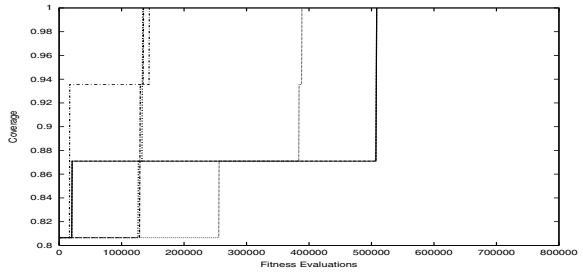
Figure 4: Results for the 'Special Value Check' Triangle program

Version with a flag




```
.
.
.
  /* date correction for september 1752 */
if(special_days)
   result = "Day did not exist.";
else
   if (leapflag && is_september && day>13)
      result = dayName((addMonths(month,year)+(--day)
                        +firstJanuary(year)+10)%7);
   else
.
.
.
```

Transformed, flag-free, version




```
.
.
.
  /* date correction for september 1752 */
if(year = 1752 && month==9 && day >=3 && day <= 13)
   result = "Day did not exist.";
else
   if (year = 1752 && month==9 && day>13)
      result = dayName((addMonths(month,year)+(--day)
                        +firstJanuary(year)+10)%7);
   else
.
.
.
```

Figure 5: Results for the Leap Year Program

Search Heuristics, Case-Based Reasoning and Software Project Effort Prediction

Colin Kirsopp, Martin Shepperd, John Hart

Empirical Software Engineering Research Group
School of Design, Engineering & Computing
Bournemouth University
Royal London House
Bournemouth, BH1 3LT
UK

Abstract

This paper reports on the use of search techniques to help optimise a case-based reasoning (CBR) system for predicting software project effort. A major problem, common to machine learning (ML) techniques in general, has been dealing with large numbers of case features, some of which can hinder the prediction process. Unfortunately searching for the optimal feature subset is a combinatorial problem and therefore NP-hard. This paper examines the use of random searching, hill climbing and forward sequential selection (FSS) to tackle this problem. Results from examining a set of real software project data show that even random searching was better than using all available features (average error 35.6% rather than 50.8%). Hill climbing and FSS both produced results substantially better than the random search (15.3 and 13.1% respectively), but FSS was computationally more efficient. Providing a description of the fitness landscape of a problem along with search results is a step towards the classification of search problems and their assignment to optimum search techniques. This paper attempts to describe the fitness landscape of this problem by combining the results from random searches and hill climbing, as well as using multi-dimensional scaling to aid visualisation. Amongst other findings, the visualisation results suggest that some form of heuristic-based initialisation might prove useful for this problem.

1 BACKGROUND TO PROJECT EFFORT PREDICTION

An important problem in the field of software engineering is finding how best to make predictions concerning size of, and effort required for, software development projects. These predictions must be made at an early stage during a project, working primarily from feasibility and requirements specification documents. Despite a significant amount of research effort over the past 30 years, no one method has been found to be consistently effective.

Early prediction techniques such as Boehm's COCOMO prediction system (Boehm 1984) (and more recent variants) were algorithms that sought to relate predicted source code size together with a large number of cost drivers to effort and nominal duration. Unfortunately, there is little independent evidence that this type of universal approach yields consistently useful results. It would seem that despite the cost drivers and various parameters COCOMO is over adapted to the data set from which it was developed. Alternative approaches include the use of simple statistical techniques such as stepwise regression to develop models that have local predictive value only, for instance (Kok, Kitchenham *et al.* 1990). More recently there has been considerable interest in a variety of ML methods that are trained on local data. This has included work with artificial neural nets, for example (Finnie, Wittig *et al.* 1997), rule induction algorithms (Mair, Kadoda *et al.* 2000) and genetic programming systems to search for functions that fit the data (Burgess and Lefley 2001; Dolado 2001). Whilst some quite accurate results have been reported, all these techniques suffer from the problem of poor explanatory value. In other words they are able to provide a prediction but not necessarily offer a justification that is helpful for a project manager who will need, in some sense, to trust the prediction if these techniques are to be deployed in practice. Another ML technique is case-based reasoning (CBR). This has an advantage in that there is substantial evidence (Klein 1998) that humans make use of analogies or prototypes when solving problems.

The remainder of this paper is organised as follows. The next section reviews the use of CBR for effort prediction. We then address the feature subset selection problem which causes particular difficulties for problems characterised by large numbers of features, few cases and

limited domain understanding. We briefly review the range of approaches to feature subset selection and in particular how it may be viewed as a search problem. We then turn to our effort prediction case study and show that feature subset selection contributes significantly to prediction accuracy for our CBR approach. We compare three search techniques: random, steepest ascent hill climbing and forward sequential selection. Next we try to explain our results in terms of the fitness landscape and the problems of visualisation. We conclude with a discussion of the significance of our results, the extent to which they might be generalised and areas requiring further investigation.

2 CASE-BASED REASONING

A number of research groups including ourselves have been investigating applying CBR to software project prediction since the mid 1990s (Prietula, Vincinanza *et al.* 1996; Shepperd, Schofield *et al.* 1996). The basic approach is that each completed project is considered as a separate case and added to a case base. Each case is characterised by n features which might be continuous, discrete or categorical. The choice of features is somewhat arbitrary and will depend upon what is available. Example features might include the number of interfaces, the level of code reuse and the design method employed. Clearly, a requirement is that these features must be known (or reliably estimated) at the time of prediction. A new project, for which a prediction is required (known as the target case), is also characterised by the same feature set and plotted in standardised n-dimensional feature space. Distance, usually a modified form of Euclidean distance, is used to identify the most similar cases to the target and these, since they have known values for effort, are used as the basis of the prediction.

There have been some differences in approach on how the analogues are used to calculate the prediction, for instance Prietula *et al.* make substantial use of adaptation rules whilst our work is closer to a k nearest neighbour (k-NN) method. We believe our approach to have the advantage of being more flexible since we are not restricted to a particular set of features which is a requirement for adaptation rules. For a thorough review of CBR the reader is referred to (Kolodner 1993).

The flexibility of the k-NN approach enabled us to develop ANGEL, a software estimation CBR tool that has a shell structure that can deal with arbitrary sets and types of features. The ANGEL tool makes a prediction in the following way. As described above, there is an existing case base of projects each characterised by a set of n features and for which the actual effort value is known. The target case is also characterised in terms of the same set of features. The feature space is created in such a way that the influence of a feature is not related to the choice of unit. This is achieved by normalising the difference between maximum and minimum observations so that all values lie between 0 and 1. This results in the feature space being represented as a unit-sided n dimensional hyper-cube. The Euclidean distance between the target case and each of the other cases is calculated and the closest k cases identified. The predicted effort[1] for the target case is then the average (or weighted average) of the actual effort values for the k closest cases. For more details see (Shepperd, Schofield *et al.* 1996).

In general the results have been sufficiently encouraging — we found that ANGEL performed as well or better than a stepwise regression model across 9 data sets (Shepperd and Schofield 1997) — to generate significant interest.

3 FEATURE SUBSET SELECTION PROBLEM

Searching for useful feature subsets has been recognised as a challenge for the ML community as a whole for a number of years. This is because all techniques — and not just CBR — are potentially vulnerable to erroneous, irrelevant or redundant data. Approaches to searching for subsets fall into two categories: filters and wrappers (Kohavi and John 1997). Filters operate independently of the ML algorithm reducing the number of features prior to training. By contrast, wrappers use the ML algorithm itself on some sample of the data set in order to determine the fitness of the subset. This tends to be computationally far more intensive, but can find better subsets than the filter methods. In this paper we focus on wrappers. This is because our goal is prediction of a continuous variable rather than classification and filter methods look for features strongly correlated with the dependent variable and orthogonal to the independent variables. Given our need to re-scale features to overcome problems of differing units, orthogonality is not necessarily a desirable characteristic of a feature subset.

Various wrapper methods have been investigated by a number of researchers. The original version of ANGEL addressed the problem of searching for the optimal feature subset by an exhaustive search using a jack knife on the case base in order to determine fitness. However, the search is approximately $O(2^n)$ so once n exceeds 15-20 this becomes computationally intractable. Other approaches have included different variants of hill climbing algorithms (Skalak 1994), simulated annealing algorithms (Debuse and Rayward-Smith 1997), sequential feature selection algorithms, both forward and backward (Aha and Bankert 1996) and genetic algorithms (Whitley, Beveridge *et al.* 1997). These have generally been reported to lead to improvements in accuracy without the prohibitive computational cost of an exhaustive search.

[1] For some prediction techniques the the exact nature and definition of effort would be significant. However, since CBR bases its prediction on effort values for previous projects, the prediction will be in the same units as these supplied effort values. The predicted value will also (indirectly) assume the same counting rules that were used to calculate the effort for the known cases. For this reason the precise definition of effort is not significant and may vary from dataset to dataset (though they should be consistent within any one dataset).

Essentially all these methods have a search component to generate candidate subsets from the space of all possible subsets and a fitness function which is a measure of the error deriving from the ML algorithm using the subset, trained on a sample from the data set and validated on a holdout sample. Typical sampling techniques are the jack knife and *n*-fold validation. The fitness function is generally a measure of error and as such is a cost that should be minimised. The exact nature of the measure will depend upon the nature of what is being predicted but is usually either based on the cost of misclassifications or the sum of absolute residuals.

4 PREDICTION CASE STUDY

This case study examines the use of different search algorithms to optimise the feature subset used to build effort prediction systems for software development projects. We look at the accuracy of the prediction systems built using different feature subsets selected using these algorithms. The results discussed in this study were generated by the ArchANGEL CBR tool[2]. The same tool settings (except feature subset selection algorithm) were used for all runs. Predictions were made using an inverse distance weighted average of the effort values for the 3 nearest neighbours, i.e. k=3. The data set was jack-knifed to produce a prediction for each case and the sum of absolute residuals for these predictions was used as the accuracy indicator for each prediction system. In other words this is a hold out one project, validation strategy.

The data used for the case study is the so-called 'Finnish dataset' derived from project data collected by a software house. This dataset contains 407 cases described by 90 features including metrics such as function points. Software project effort data sets are characterised by relatively few cases (almost invariably under 500 and typically less than 50). Therefore the data set used in this paper is at the large end of this spectrum. The features are a mixture of continuous, discrete and categorical. However, there are a number of missing data values and also some features that would not be known at prediction time and so should not be included in a prediction system. Removing features with missing values or after-the-event data leaves a subset of 44 features that are actually used in the case study. The data set also exhibits significant multi-collinearity, in other words there are strong relationships between features as well as with the feature to be predicted, namely effort.

The number of possible combinations of feature subsets is 2^n where n is the number of features. In this case study there are 44 features, however, one is used as the target feature, namely total project development effort. This leaves 43 features as the input to the subset search, giving $8.8\ e^{12}$ possible combinations. Clearly an exhaustive search is not feasible so other search strategies must be considered.

This case study examines three alternative strategies:

1. Random feature subset selection
2. Multi-start steepest ascent hill climbing
3. Forward sequential selection

We restricted our choice for two reasons. First, other groups have had some success with these algorithms for finding good feature subsets. Second, there seems no purpose in examining more complex search strategies if the problem can be effectively solved using a hill-climber (Juels and Wattenberg 1994). Also, in order to analyse the value of feature subset selection we used the accuracy from using all features as a comparative baseline.

As a search problem we need to consider two additional issues: representation of solutions and measurement of fitness. Fortunately, for feature subset selection problems the set of candidate features can be simply represented as a bit string where 1 denotes selected and 0 excluded. This also provides a view of neighbourhood which is defined as any move derived from mutating a single bit, in other words moving any one feature into, or out of the selected set. Fitness, or strictly speaking cost[3], is a little more complex. As stated in the previous section we are concerned with wrappers. Informally we prefer a feature subset that leads to more accurate project effort prediction. This is defined as jack-knifing across the entire data set so that we obtain a predicted effort value $\hat{e}$ for every case. This can be compared with the true effort value e, in order to derive an absolute residual r which is $|\hat{e}-e|$ since we are indifferent to the direction of error. For the entire data set we sum the absolute residuals. Note that this is a more neutral view of prediction error compared with, say, the sum of the squares of the residuals which adopts a more risk averse stance since a few extreme errors will dominate the fitness measure.

4.1 RANDOM FEATURE SELECTION

A simple approach to finding a suitable feature subset in such a large search space is to collect results from a large number of randomly generated feature subsets. Table 1 shows the summary of results from 4028 randomly sampled feature subsets.

It is worth noting that using all features gives a result close to the mean of the random feature subsets. This implies that a randomly chosen subset of the features is likely to be just as good as using all available features. The best solution found has an accuracy value of 56988 compared with 88522 using all features. This is a significant improvement (recall that low values of sum(|r|) are preferred), however we next turned to a more systematic search strategy in order to seek better results.

[2] ArchANGEL is the most recent version of the ANGEL software tool for project prediction. It may be downloaded from http://dec.bmth.ac.uk/ESERG/ANGEL/

[3] Whilst we actually wish to minimise the fitness function we will retain the usual terminology of hill-climbing, peaks and so forth.

Table 1: Summary statistics for random feature selection

Count	4028
Mean	91272
Median	91831
Min	56988
Max	135586

4.2 HILL CLIMBING

A commonly used search strategy is hill climbing. In this case study the algorithm used was multi-start steepest ascent hill climbing where each climb has a new, randomly selected starting point (or initial feature set). The algorithm is steepest ascent because the entire neighbourhood is evaluated and the move with the best result is used as the next base position. The neighbourhood is defined as any new feature set that can be obtained from toggling a single bit. This definition of the neighbourhood means that any feature set has 43 neighbours that must be evaluated for each step in the climb.

Table 2: Summary statistics for 113 hill climbs

Count	113
Mean	47803
Median	51909
Min	29916
Max	61049

Table 2 gives the summary statistics for the accuracy levels achieved by the 113 hill climbs. We see that the best feature subset found by hill climbing gave a result of 29916. This is a significant improvement over either using all features or random searching (88522, 56988), see also Table 4. Also worthy of note is that the maximum (worst) value of a peak found by hill climbing was 61049. This is only slightly worse than the best of the random results (56988). In fact, all but 8 of the 113 climbs produced results better than all 4028 random selections.

Figure 1 shows the distribution of results from Table 2. Note the bi-modal nature of the hill climbing results. There are a number of significantly better solutions that are clearly separated from the main distribution of results. When the distinction between this set of outlying good solutions and the other results was investigated it was found that the best solutions used fewer features.

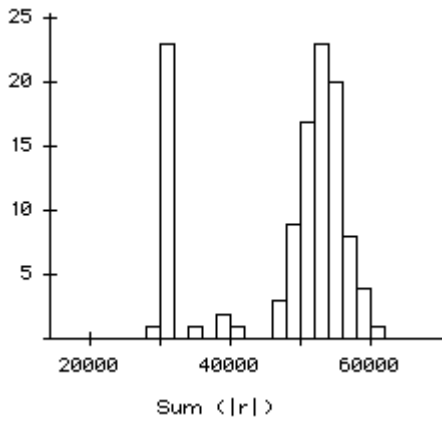


Figure 1: Distribution of Results from Hill Climbing

Figure 2 shows a scatter plot of sum(|r|) against the number of features included in the feature subset. A slight trend is notable in the main cloud of results but it is the lower valued outliers that most clearly show this effect. No good solution was found that contains more than 15 features (out of 43).

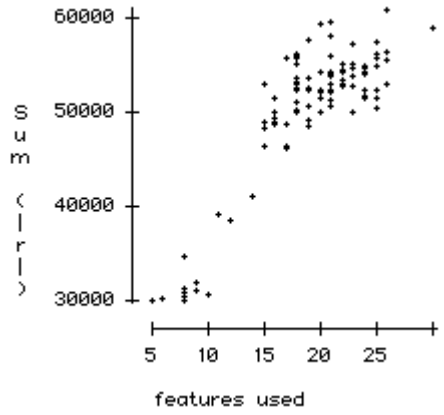


Figure 2: Scatter plot of accuracy against number of features used

4.3 FORWARD SEQUENTIAL SELECTION

Another form of search that can be used in this situation is a sequential feature selection algorithm. There are two main types of sequential selection algorithms, forward sequential selection (FSS) and backward sequential selection (BSS). In FSS, features are added one by one starting from any empty feature set. In BSS, features are removed one by one from a complete feature set. We used an FSS algorithm due to the observation that the best

solutions from hill climbing were relatively feature-sparse (see Figure 2)[4].

The FSS algorithm used works in a broadly similar way to steepest ascent hill climbing but with two main differences. The first difference is that it starts from having no features selected rather than a random selection. Secondly, features are only added (never removed). The algorithm starts by finding the best single feature for prediction and adding this to the empty feature set. An attempt is then made to add another feature. This is done by evaluating the accuracy of predictions obtained using a combination of the current feature set with each of the other features in turn. The best of these combinations becomes the current feature set. This process is repeated, adding one feature at each stage, until none of the combinations yields a better result than the current feature set.

The FSS result was 30202. This is only slightly worse than the result from the best hill climb (by less than 1%) and this result was reached far more quickly. The FSS result was obtained by evaluating just 243 feature combinations. A single hill climb required an average of 688 evaluations and only 3 of 113 hill climbs produced results equal to or better than the FSS result. This means that hill climbing would, on average, require around 19000 evaluations to find an equal or better result.

5 FITNESS LANDSCAPE

The close relationship between the form of the fitness landscape and the performance of search algorithms has been noted by many researchers, e.g. (Crisan and Muhlenbein 1998; Reeves 2000). Since the 'no free lunch' theorem (Wolpert and Macready 1997), researchers realise that is not possible to say that a particular search algorithm is always better than another. Research now concentrates on trying to show that a particular search algorithm is better than another for a restricted class of search problem. Difficulties with this approach include trying to define a class of search problem and trying to identify that class of problem *a priori*.

A step towards identifying which search algorithms are best for which class of problem is to identify characteristics of the fitness landscape that aid a particular algorithm. Work has been done towards this on a theoretical basis (Sharpe 1998). This section attempts to describe the landscape for this particular search problem and to comment on how this may have affected the results from the search algorithms used. However, there are several challenges. First, and obviously, there is the sheer size of the landscape. Second, there is a visualisation problem. It is convenient to think in terms of a landscape in which the x and y co-ordinates represent the search space and the height the quality or fitness of the solution. Thus we use such abstractions as hill climbing and peaks. For our problem we must confront 43 dimensions each of extreme coarseness i.e. binary. Consequently simple representations are inapplicable.

The random search can be thought of as sampling from the entire landscape. The results from this sampling could therefore be used to make inferences about the underlying population where the underlying population is the set of results obtained from an exhaustive search of the feature space (if that were computationally possible).

The results in Figure 3 show the distribution of 'heights' in the fitness landscape (but not their positions). Values follow an approximately normal distribution centred on a mean of just over 91000 with approximate maximum and minimum values of 135000 and 57000. This gives some idea of typical values, although clearly the whole point of a hard search is that we are interested in the extreme outliers. So we have some idea of the proportions of the fitness landscape that lie at particular heights, but no information on how individual points are positioned in the fitness landscape. The landscape could be entirely chaotic or organised as a single smooth peak. There is no way of knowing the structure of the landscape from these results.

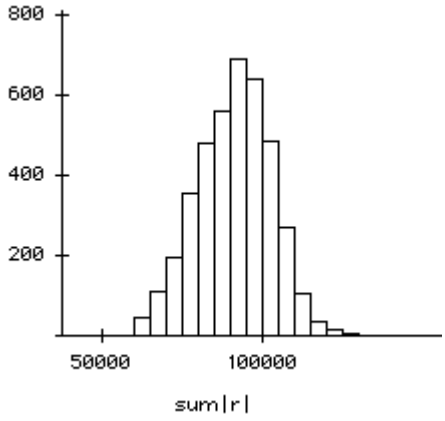


Figure 3: Distribution of results for randomly selected feature sets

The results from the hill climbing also help provide information concerning the fitness landscape. Hill climbing algorithms depend on certain assumptions about the nature of the fitness landscape for their operation. They exploit local correlations in the structure of the search space, i.e., they assume that there is likely to be a better point near to a good point. A hill climbing algorithm would not work in a chaotic landscape. The significantly better results produced by the hill climbing (over the random selection) suggest that there is structure

[4] Normally, BSS is recommended rather than FSS. This is because BSS evaluates features to remove in the presence of all the other features that may be included in the final solution. This allows it to take advantage of any interaction between the features when making the decision on which feature to remove. However, it has been suggested that BSS 'is more easily confused by large numbers of features' and that FSS 'is preferred when the optimal number of selected features is small', see for example (Aha and Bankert 1996). Since this search has a large number of features and the results from hill climbing suggest that the better results have a small number of features we chose FSS.

in the landscape that is exploited by hill climbing. The length of the climbs also supports this view. The less structured the landscape the shorter individual climbs would be. The median climb length was 15 steps and the maximum climb length was 31 steps. Compare this with the 43 steps that is needed to traverse the entire breadth of the state space and there are clearly large-scale structures in the fitness landscape.

What else can the hill climbing results tell us about the structure of the fitness landscape? Firstly, there are a large number of distinct peaks in the landscape (highly multi-modal). The 113 hill climbs found 98 distinct peaks - only 15 climbs found a previously visited peak. Interestingly, all of the peaks that were visited more than once were one of the ten highest peaks. This suggests that the better peaks in this landscape have larger basins of attraction.

If we combine the hill climbing results with the random results another observation can be made. The best of 4028 random selections was better than only the worst 8 hill climbs. There is very little chance of finding a good result by random searching. This implies that little of the feature space lies near the top of the hills, i.e., the peaks are very sharp.

As well as the local structures within the landscape that aid individual hill climbs there could also be larger scale structures or trends. To try to identify any such trends requires a way of visualising the position and height of points in 43 dimensional binary space. A multi-dimensional scaling (MDS) algorithm was used to achieve this visualisation. MDS is used to compress high dimensional data into lower dimensional space while attempting to retain the relative distances between the data points. We used the MDS algorithm provided by the statistics package SPSS to convert the similarity in feature subsets corresponding to each peak into co-ordinates in 2 dimensional space. These points were then plotted and their height or fitness denoted by different symbols. Note that we have no data on intervening points so we cannot draw contours.

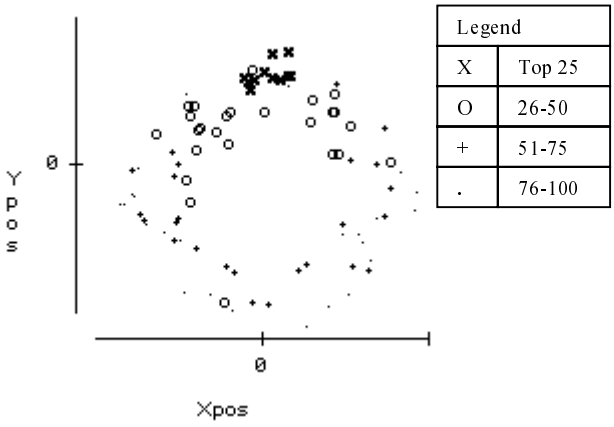


Figure 4: MDS of hill climbing results

The results for the best 100 hill climbs are shown in Figure 4. There is a general tendency for the better peaks to be in the upper part of the figure and the worse peaks to be in the lower part of the figure. More notable is the tight cluster of Xs representing the 25 best climbs (10 best peaks). These observations show that there are some global trends within the feature space. This may be a trend in just the height of the peaks or a general raising of the landscape. To investigate this issue a second random sampling was done that was constrained to the region of the highest 10 peaks. Figure 5 shows the results from this localised random sampling.

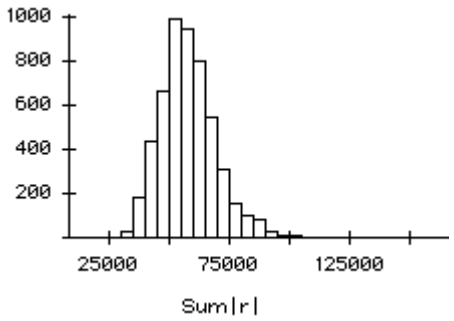


Figure 5: Distribution of restrained random search

From the clear differences in central tendency between the constrained and the global random samples, there appears to be a tendency for the average height to increase towards a particular region in the feature space. To continue the landscape analogy, although there are many local peaks throughout the landscape, in general the landscape rises towards a localised massif. This massif has a higher average level of fitness as well as the highest peaks. It does, however, still contain some deep sinkholes (see Table 3 where the lowest fitness was in excess of 143000).

Table 3: Summary statistics for restrained random search

Count	5358
Mean	58095
Median	56714
Min	31353
Max	143824

In some ways FSS can be thought of as a single steepest ascent hill-climb with a fixed starting point and some moves made taboo, since once features are included in a subset they cannot subsequently be removed. The best hill climbs are all in a region of sparse feature subsets and the FSS path also operates in this region of the fitness landscape. The value of a good starting point can be seen in the short length of the path (5 steps) and consequent efficiency gains.

To summarise, we know the landscape is multi-modal, that the peaks have steep slopes and small summits and somewhat surprisingly there are some very large-scale structures as evidenced by the large basins of attraction for the better solutions. This last observation was unexpected. What we do not know is whether there exists some higher peak, or peaks, but with a very small basin of attraction.

6 DISCUSSION

This paper has looked at the performance of a number of search techniques to solve a real world software engineering problem. The problem was the optimisation of the feature set used by a *k*-NN system to predict software project effort. Table 4 shows a summary of the results for these techniques. The table shows accuracy of the best feature set found by each method (measured as the sum of the absolute residuals). It also shows the maximum and mean values for techniques where multiple results were produced. To provide an intuitive feel for the practical value of the prediction systems produced, an alternative measure of accuracy — mean absolute relative error — is also included[5]. Finally, as a means of indicating efficiency, we provide the number of different feature sets evaluations required by each technique.

Table 4: Summary of results

	All	Random	Hill	FSS
Evaluations	1	4028	74433	243
Min sum|r|	88522	56988	29916	30202
Max sum|r|		135586	61049	
Mean sum|r|		91272	47803	
% Error	50.8 %	35.6 %	15.3 %	13.1 %

The simplest and quickest technique is to use all of the features, as this requires a single evaluation. However the accuracy results obtained are very poor. Prediction made using this system had an average error of 50.8%. This is intended as a base line against which we see all other techniques yield accuracy benefits.

The results show that a random search is clearly better than simply using all features. The best feature set found with this method resulted in a prediction system with an average error of 35.6%. Longer runs of random trials could bring this value down further but more intelligent search techniques would probably be preferred.

Hill climbing found the best solution known to us (as measured by sum|r|[6]). This result was equivalent to an average error on predictions of about 15%. However, hill climbing is very computationally intensive. The table shows that a total of 74433 different feature sets were evaluated during the 113 hill climbs. This form of multiple-start hill climbing (like the random search) has no in-built end point. It can be run indefinitely but is likely to show diminishing returns as it converges on an exhaustive search.

Forward sequential selection yielded a result only fractionally worse than the best hill climbing result in terms of residuals and slightly better in terms of relative error (13.1%). The major advantage of the FSS technique was only 243 evaluations were required to reach this result

Following this study the authors would make the following recommendations to anyone trying to optimise a feature subset for case-based prediction of software development effort.

1. If the feature set is small (<15-20 features) use an exhaustive search. This will always find the optimum feature subset.
2. If the feature set has more than 20 features an exhaustive search will not be computationally possible. Based on the work presented in this paper (albeit a single dataset) the authors would recommend trying FSS since it is significantly more efficient than a hill-climber and there was little difference in accuracy.

It is worth noting that the inherently noisy nature of such real world data means that there may be little scope for further improvement. Also, for our problem we are only seeking good engineering approximations. Thus we can view a solution as good enough. Any "real world" project effort prediction will be vulnerable to changes in environment, requirements, staff and so on, thus in practice ultra-high levels of accuracy are somewhat illusory. If readers are considering applying more complex algorithms to tackle this type of problem they should consider whether it would be worthwhile in the light of the relatively small improvement that might potentially be yielded.

As well as applying a set of search techniques to a particular software engineering problem, we have also attempted to describe the fitness landscape of the problem. Given that the fitness landscape is comprised of 43 binary dimensions, describing the landscape is a difficult task. A number of techniques were employed to help describe and visualise the landscape. Firstly, the result from the random search were interpreted as samples from the fitness landscape and used to find how much of the landscape was at various heights. Secondly, hill climbing was used to look for the presence of structure in the landscape and to gauge the number and height of peaks in the landscape. Multi-dimensional scaling was used as a means of visualising the relative position of the various peaks found by hill climbing. The results from the MDS helped to identify global trends in the landscape.

[5] We did not use mean absolute relative error (sometimes referred to as MMRE in the software engineering literature) as our fitness function since it is asymmetric and heteroscedastic.

[6] From table 4 it can be seen that although hill climbing has a lower sum|r| than FSS, it has a higher % error. This rank reversal is due to different accuracy indicators measuring different aspects of error.

Localised exhaustive or random searches can also be used to further investigate areas of interest within the fitness landscape.

The result of the investigation of the fitness landscape for this problem showed it to be highly multi-modal. The landscape contained sharp peaks and troughs and rose towards a multi-peaked 'massif' that contained all of the better results found.

To conclude, we have described the successful use of search techniques on a real world software engineering problem. By means of the search for better feature subsets, software project effort prediction error has been reduced from 50.8 % to 13.1 %. We have also suggested techniques to help visualise fitness landscapes and used them on a real example. Lastly we have noted that applying search techniques to engineering problems is not necessarily the search for an optimum value. A good enough result reached in a timely fashion may be better than a prolonged search for an optimum result. The successful use of simple search techniques such as FSS and hill climbing suggest that researchers should try such methods before resorting to more complex techniques and that as a side effect these techniques also reveal useful information about the nature of the fitness landscape.

Acknowledgments

The authors are indebted to STTF Ltd for making the Finnish data set available.

References

Aha, D. W. and R. L. Bankert (1996). A comparative evaluation of sequential feature selection algorithms. *Artificial Intelligence and Statistics V*. D. Fisher and J.-H. Lenz. New York, Springer-Verlag.

Boehm, B. W. (1984). "Software engineering economics." *IEEE Transactions on Software Engineering* **10**(1): 4-21.

Burgess, C. J. and M. Lefley (2001). "Can genetic programming improve software effort estimation? A comparative evaluation." *Information & Software Technology* **43**(14): 863-873.

Crisan, C. and H. Muhlenbein (1998). The frequency assignment problem: A look at the performance of evolutionary search. *Artificial Evolution: Lecture Notes in Computer Science, Vol. 1363:***:** 263-273.

Debuse, J. C. W. and V. J. Rayward-Smith (1997). "Feature subset selection within a simulated annealing data mining algorithm." *J. of Intelligent Information Systems* **9**: 57-81.

Dolado, J. J. (2001). "On the problem of the software cost function." *Information & Software Technology* **43**(1): 61-72.

Finnie, G. R., G. E. Wittig and J.-M. Desharnais (1997). "A comparison of software effort estimation techniques using function points with neural networks, case based reasoning and regression models." *J. of Systems Software* **39**: 281-289.

Juels, A. and M. Wattenberg (1994). Stochastic hillclimbing as a baseline method for evaluating genetic algorithms, Technical Report, Univ. of California at Berkeley.

Klein, G. (1998). *Sources of Power : How People Make Decisions*. Cambridge, Ma, MIT Press.

Kohavi, R. and G. H. John (1997). "Wrappers for feature selection for machine learning." *Artificial Intelligence* **97**: 273-324.

Kok, P., B. A. Kitchenham and J. Kirakowski (1990). The MERMAID approach to software cost estimation. *Esprit Technical Week*.

Kolodner, J. L. (1993). *Case-Based Reasoning*, Morgan-Kaufmann.

Mair, C., G. Kadoda, M. Lefley, K. Phalp, C. Schofield, M. Shepperd and S. Webster (2000). "An investigation of machine learning based prediction systems." *J. of Systems Software* **53**(1): pp23-29.

Prietula, M. J., S. S. Vincinanza and T. Mukhopadhyay (1996). "Software Effort Estimation With a Case-Based Reasoner." *J. Experimental & Theoretical Artificial Intelligence* **8**: 341 - 363.

Reeves, C. R. (2000). Fitness landscapes and evolutionary algorithms. *Artificial Evolution: Lecture Notes in Computer Science, Vol. 1829***:** 3-20.

Sharpe, O. (1998). Beyond NFL: A few tentative steps. *3rd Conf. on Genetic Programming (GP'98)*, Madison, Wisconsin., Morgan Kaufman.

Shepperd, M. J. and C. Schofield (1997). "Estimating software project effort using analogies." *IEEE Transactions on Software Engineering* **23**(11): 736-743.

Shepperd, M. J., C. Schofield and B. A. Kitchenham (1996). Effort estimation using analogy. *18th Intl. Conf. on Softw. Eng.*, Berlin, IEEE Computer Press.

Skalak, D. B. (1994). Prototype and feature selection by sampling and random mutation hill climbing algorithms. *11th Intl. Machine Learning Conf. (ICML-94)*, Morgan Kauffmann.

Whitley, D., J. R. Beveridge, C. Guerra-Salcedo and C. Graves (1997). Messy genetic algorithms for subset feature selection. *International Conference on Genetic Algorithms, ICGA-97.*

Wolpert, D. H. and W. G. Macready (1997). "No Free Lunch Theorems for Search." *IEEE Transactions on Evolutionary Computation* **1**(1): 67-82.

Using Heuristic Search Techniques to Extract Design Abstractions from Source Code

Brian S. Mitchell and Spiros Mancoridis
Department of Mathematics & Computer Science
Drexel University, Philadelphia, PA 19104
{bmitchel, smancori}@mcs.drexel.edu

Abstract

As modern software systems are large and complex, appropriate abstractions of their structure are needed to make them more understandable and, thus, easier to maintain. Software clustering tools are useful to support the creation of these abstractions. In this paper we describe our search algorithms for software clustering, and conduct a case study to demonstrate how altering the clustering parameters impacts the behavior and performance of our algorithms.

1 Introduction & Background

Software supports many business, government, and social institutions. As the processes of these institutions change, so must the software that supports them. Changing software systems that support complex processes can be quite difficult, as these systems can be large (*e.g.*, thousands or even millions of lines of code) and dynamic.

There is a need to develop sound methods and tools to help software engineers understand large and complex systems so that they can modify the functionality or repair the known faults of these systems. Understanding how the software is structured – at various levels of granularity – is one of several kinds of understanding that is important to a software engineer. As the software structure can itself be very complex, the appropriate abstractions of a system's structure must be determined. Techniques and tools can then be designed and implemented to support the creation of these abstractions.

Automatic design extraction methods have been proposed to create abstract views – similar to "road maps" – of a system's structure. Such views help software engineers cope with the complexity of software development and maintenance. Design extraction starts by parsing the source code to determine the components and relations of the software. The parsed code is then analyzed to produce a variety of views of the software structure, at varying levels of abstraction.

Detailed views of software structure are appropriate when the software engineer has isolated the subsystems that are relevant to his or her analysis. However, abstract (architectural) views are more appropriate when the software engineer is trying to understand the global structure of the software. Software clustering is used to produce such abstract views. These views feature module-level components and relations contained within subsystems. The source code level components and relations can be determined using source code analysis tools. The subsystems, however, are not found in the source code. Rather, they are inferred from the source code components and relations either automatically, using a clustering tool, or manually, when tools are not available.

The problem of automatically creating abstract views of software structure is very computationally expensive (NP-hard) (Garey and Johnson, 1979), so a hope for finding a general solution to the software clustering problem is unlikely. Nevertheless, several heuristic approaches to solving this problem have been proposed. These approaches generally differ in the way that they create subsystems. For example, some popular clustering techniques use source code component similarity (Hutchens and Basili, 1985; Schwanke, 1991; Choi and Scacchi, 1999; Müller *et al.*, 1992), concept analysis (Lindig and Snelting, 1997; Deursen and Kuipers, 1999), or implementation information such as module, directory, and/or package names (Anquetil *et al.*, 1999) to derive the subsystems. Our clustering approach differs from the others as it is based on heuristic search techniques (Mancoridis *et al.*, 1998; Doval *et al.*, 1999; Mancoridis *et al.*, 1999; Mitchell *et al.*, 2001).

In this paper we examine the latest features of the software clustering techniques that are implemented in our clustering tool named Bunch. We will focus on the enhancements that we have made to our hill-climbing clustering algo-

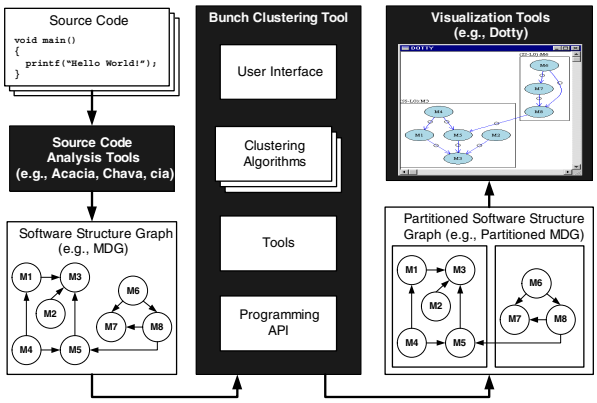


Figure 1: Bunch's Design Extraction Process

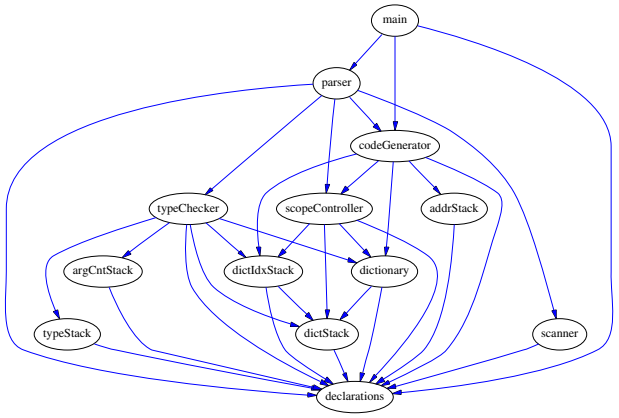


Figure 2: The MDG for a Small Compiler

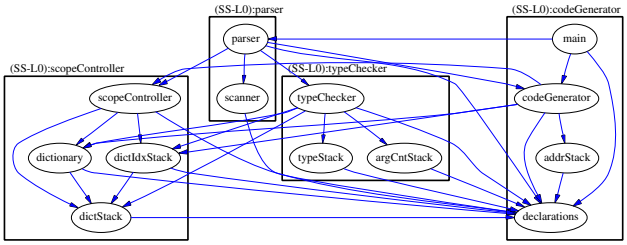


Figure 3: The Partitioned MDG for a Small Compiler

rithm since it was first published in the 1998 IWPC proceedings (Mancoridis *et al.*, 1998). We also conduct a case study to demonstrate how the various clustering parameters supported by our latest hill-climbing algorithm impact the clustering results when applied to several systems of varying size. Our goal is to help Bunch users understand the parameters of our clustering algorithms.

2 Design Extraction using Bunch

The first step in our design extraction process (see Figure 1) is to determine the resources and relations in the source code and store the resultant information in a database. Readily available source code analysis tools – supporting a variety of programming languages – can be used for this step (Chen, 1995; Korn *et al.*, 1999). After the resources and relations have been stored in a database, the database is queried and a Module Dependency Graph (MDG) is created. For now, consider the *MDG* to be a directed graph that represents the software modules (e.g., classes, files, packages) as nodes, and the relations (e.g., function invocation, variable usage, class inheritance) between modules as directed edges. Once the *MDG* is created, Bunch's clustering algorithms can be used to create the partitioned *MDG*. The clusters in the partitioned *MDG* represent subsystems that contain one or more modules, relations, and possibly other subsystems. The final result can be visualized and browsed using a graph visualization tool such as dotty (North and Koutsofios, 1994).

An example *MDG* for a small compiler developed at the University of Toronto is illustrated in Figure 2. We show a sample partition of this *MDG*, as created by Bunch, in Figure 3. Notice how Bunch automatically created four subsystems to represent the abstract structure of a compiler. Specifically, there are subsystems for code generation, scope management, type checking, and parsing.

The center of Figure 1 depicts additional services that are supported by the Bunch clustering tool. These services are discussed thoroughly in Bunch's user and programmer documentation, which can be accessed on the Drexel University Software Engineering Research Group web page (SERG, 2002).

3 Bunch's Updated Clustering Algorithms

The goal of Bunch's clustering algorithms is to partition the Module Dependency Graph (MDG) so that the clusters represent meaningful subsystems. We formally define the *MDG* to be a graph $MDG = (M, R)$, where M is the set of named modules in the software system, and $R \subseteq M \times M$ is a set of ordered pairs of the form $\langle u, v \rangle$ which represents the source-level relationships that exist between module u and module v. Also, the *MDG* can have weighted edges to establish the "strength" of the relation between a pair of modules. An example *MDG* consisting of 8 modules is shown on the left side of Figure 4.

Once the *MDG* of a software system is determined, we search for a "good" partition of the *MDG*. We accomplish this task by using heuristic searches whose goal is to maximize the value of an objective function that is based on a formal characterization of the trade-off between coupling (*i.e.,* connections between the components of two distinct clusters) and cohesion (*i.e.,* connections between the components of the same cluster). We refer to our objective function as the *Modularization Quality* (*MQ*) of a partition of an *MDG*. This measurement represents the "quality" of a system decomposition. *MQ* adheres to our fundamental

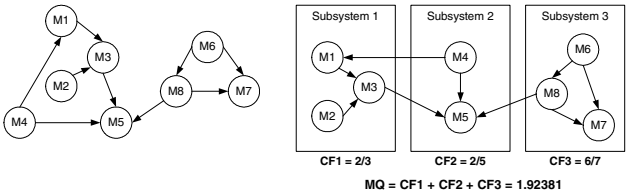


Figure 4: Modularization Quality Example

assumption that well-designed software systems are organized into cohesive clusters that are loosely interconnected. The *MQ* expression is designed to reward the creation of highly cohesive clusters yet penalize excessive inter-cluster coupling.

The *MQ* for an *MDG* partitioned into k clusters is calculated by summing the *Cluster Factor* (*CF*) for each cluster of the partitioned *MDG*. The Cluster Factor, CF_i, for cluster i ($1 \leq i \leq k$) is defined as a normalized ratio between the total weight of the internal edges (edges within the cluster) and half of the total weight of external edges (edges that exit or enter the cluster). We split the weight of the external edges in half in order to apply an equal penalty to both clusters that are are connected by an external edge. We refer to the internal edges of a cluster as intra-edges (μ_i), and the edges between two distinct clusters i and j as inter-edges ($\varepsilon_{i,j}$ and $\varepsilon_{j,i}$ respectively). If edge weights are not provided by the *MDG*, we assume that each edge has a weight of 1. Also, note that $\varepsilon_{i,j} = 0$ and $\varepsilon_{j,i} = 0$ when $i = j$. Below, we define the *MQ* calculation:

$$MQ = \sum_{i=1}^{k} CF_i \qquad CF_i = \begin{cases} 0 & \mu_i = 0 \\ \dfrac{\mu_i}{\mu_i + \frac{1}{2}\sum\limits_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & otherwise \end{cases}$$

Figure 4 illustrates an example *MQ* calculation for an *MDG* consisting of 8 modules that are partitioned into 3 subsystems. The value of *MQ* is approximately 1.924, which is the result of summing the Cluster Factor for each of the three subsystems. The larger the value of *MQ*, the better the partition. For example, the *CF* for Subsystem 2 is 0.4 because there is 1 intra-edge, and 3 inter-edges. Applying these values to the expression for *CF* results in $CF_2 = 1/(1 + \frac{3}{2}) = 2/5 = 0.4$.

The *MQ* measurement described above is the most recent objective function that we have integrated into the Bunch tool. Our older objective function, which was described in an earlier paper (Mitchell *et al.*, 2001) worked well, but we noticed after significant testing that the clusters produced by Bunch tended to minimize the inter-edges that exited the clusters, and not minimize the number of inter-edges in general. Also, the above described *MQ* measurement supports *MDGs* that contain edge weights, which is a feature that was not supported by our original *MQ* measurement (Mancoridis *et al.*, 1998).

3.1 Clustering Algorithms

One way to find the best partition of an *MDG* is to perform an exhaustive search through all of the valid partitions, and select the one with the largest *MQ* value. However, this is often impossible because the number of ways the *MDG* can be partitioned grows exponentially with respect to the number of its nodes (modules) (Mancoridis *et al.*, 1998).[1] Because discovering the optimal partition of a *MDG* is only feasible for small software systems (*e.g.,* fewer then 15 modules), we directed our attention, instead, to using heuristic search algorithms that are capable of discovering approximation results quickly. The approximation search strategies that we have investigated and implemented in the Bunch (Mancoridis *et al.*, 1999) clustering tool, are based on hill-climbing (Mancoridis *et al.*, 1998) and genetic (Doval *et al.*, 1999) algorithms.

3.1.1 Hill-Climbing Algorithm

Bunch's hill-climbing clustering algorithms start by generating a random partition of the *MDG*. Modules from this partition are then rearranged systematically in an attempt to find an "improved" partition with a higher *MQ*. If a better partition is found, the process iterates, using the improved partition as the basis for finding even better partitions. This hill-climbing approach eventually converges when no additional partitions can be found with a higher *MQ*.

Neighboring Partitions

Our hill-climbing algorithms move modules between the clusters of a partition in an attempt to improve *MQ*. This task is accomplished by generating a set of neighboring partitions (*NP*).

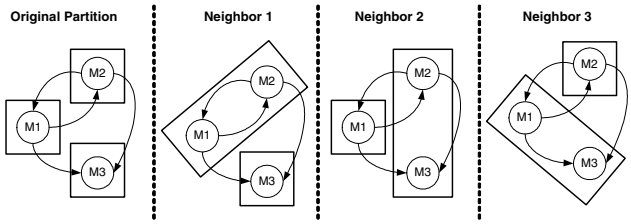


Figure 5: Neighboring Partitions

We define a partition *NP* to be a neighbor of a partition P if and only if *NP* is exactly the same as P except that a single element of a cluster in partition P is in a different cluster

[1] It should also be noted that the general problem of graph partitioning (of which software clustering is a special case) is NP-hard (Garey and Johnson, 1979).

in partition *NP*. If partition P contains m nodes and k clusters, the total number of neighbors is $O(n \cdot k)$. Figure 5 illustrates an example partition, and all of its neighboring partitions.

3.1.2 Genetic Algorithm

Hill-climbing search algorithms suffer from the well-known problem of "getting stuck" at local optimum points, and therefore possibly missing the global optimum (best solution). To address this concern we have investigated other search algorithms such as Genetic Algorithms (GA) (Goldberg, 1989; Mitchell, 1997), and applied these algorithms to the software clustering problem. We have found that the results produced by Bunch's hill-climbing algorithm are typically better than the Bunch GA (Doval *et al.*, 1999). Upon studying this outcome, we concluded that further work on our encoding and crossover techniques are necessary.

4 Hill-Climbing Algorithm Enhancements

The previous section of this paper describes the clustering algorithms that are supported by Bunch. The emphasis since the Bunch project was started has been on our hill-climbing clustering algorithms. The rest of this section describes various enhancements that we recently made to the hill-climbing algorithm described in Section 3.1.1.

4.1 Adjustable Hill-Climbing Threshold

The hill-climbing algorithm described in Section 3.1.1 starts with a generated random partition of the *MDG*. It then iterates using our neighboring partition strategy[2] to find an "improved" partition using the *MQ* objective function. During each iteration several options are available for controlling the behavior of the hill-climbing algorithm:

1. The neighboring process uses the first partition that it discovers with a larger *MQ* as the basis for the next iteration.

2. The neighboring process examines all neighboring partitions and selects the partition with the largest *MQ* as the basis for the next iteration.

3. The neighboring process ensures that it examines a minimum number of neighboring partitions during each iteration. If multiple partitions with a larger *MQ* are discovered within this set, then the partition with the largest *MQ* is used as the basis for the next iteration. If no partitions are discovered that have a larger *MQ*, then the neighboring process continues and uses the next partition that it discovers with a larger *MQ* as the basis for the next iteration.

[2]The hill-climbing algorithm examines the set of neighboring partitions in random order.

To address the last option from the above list, the latest version of the hill-climbing algorithm uses a threshold η ($0\% \leq \eta \leq 100\%$) to calculate the minimum number of neighbors that must be considered during each iteration of the hill-climbing process. A low value for η generally results in the algorithm taking more "small" steps prior to converging, and a high value for η results in the algorithm taking fewer "large" steps prior to converging.

Our experience has shown that examining many neighbors during each iteration (*i.e.*, using a large threshold such as $\eta \geq 75\%$) increases the time the algorithm needs to converge to a solution. One obvious question is if the increased runtime increases the likelihood of finding a better solution than if the first discovered neighbor with a higher *MQ* ($\eta = 0\%$) is used as the basis for the next iteration of the hill-climbing algorithm. Answering this question is one of the goals of the case study that is described in Section 5.

4.2 Simulated Annealing

A well-known problem of hill-climbing algorithms is that certain initial starting points may converge to poor solutions (*i.e.*, local optima). To address this problem, our hill-climbing algorithm does not rely on a single random starting point, but instead uses a collection of random starting points.

Another way to overcome the above described problem is to use Simulated Annealing (SA) (Kirkpatrick *et al.*, 1983). SA algorithms are based on modeling the cooling processes of metals, and the way liquids freeze and crystalize. When applied to optimization problems, SA enables the search algorithm to accept, with some probability, a worse variation as the new solution of the current iteration. As the computation proceeds, the probability diminishes. The slower the *cooling schedule*, or rate of decrease, the more likely the algorithm is to find an optimal or near-optimal solution. SA techniques typically represent the cooling schedule with a *cooling function* that reduces the probability of accepting a worse variation as the optimization algorithm runs.

We defined a cooling function that establishes the probability of accepting a worse, instead of a better partition during each iteration of the hill-climbing algorithm. The idea is that by accepting a worse neighbor, occasionally the algorithm will "jump" to explore a new area in the search space. Our cooling function is designed to respect the properties of the SA cooling schedule, namely: (a) decrease the probability of accepting a worse move over time, and (b) increase the probability of accepting a worse move if the rate of improvement is small. Below we present our cooling function that is designed with respect to the above requirements.

$$P(A) = \begin{cases} 0 & \triangle MQ \geq 0 \\ e^{\frac{\triangle MQ}{T}} & \triangle MQ < 0 \end{cases} \qquad T(k+1) = \alpha \cdot T(k)$$

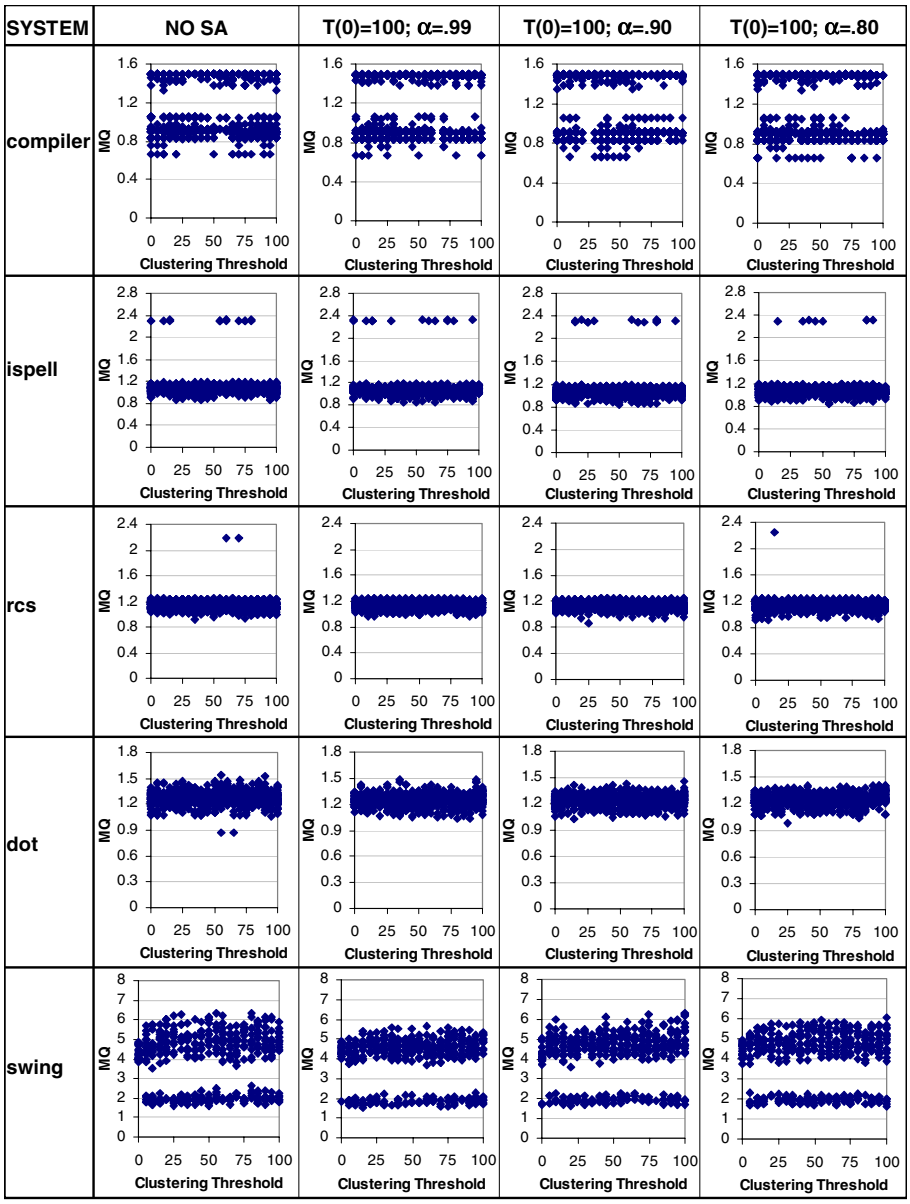


Figure 6: Case Study Results – MQ versus η Scatter Plot

Each time the cooling function is evaluated, $T(k)$ is reduced. The initial value of T (*i.e.*, $T(0)$) and the rate of reduction constant α are established by the user. Furthermore, $\triangle MQ$ must be negative, which means that the *MQ* value has decreased. Once the probability of accepting a partition of the *MDG* with a lower *MQ* is calculated, a uniform random number between 0 and 1 is chosen. If this random number is less than the probability $P(A)$, the partition is accepted.

5 Case Study

In the previous section we described some recent enhancements that were made to Bunch's hill-climbing clustering algorithms. These new features require the user to set configuration parameters to guide the clustering process. In this section we present a case study to investigate the impact of altering these parameters on systems of various size.

Table 1 describes the 5 systems that we used in our case study. We selected these systems because they vary in size and complexity. Our basic test involved clustering each system 1,050 times, consisting of 21 tests, with 50 runs in each test. The first 50 clustering runs were executed with the adjustable clustering threshold η set to 0%. The next set of 20 tests (with 50 runs in each test) involved incrementing η by 5% until η reached 100%.

We then repeated the above test for each of the systems described in Table 1, this time using the simulated annealing

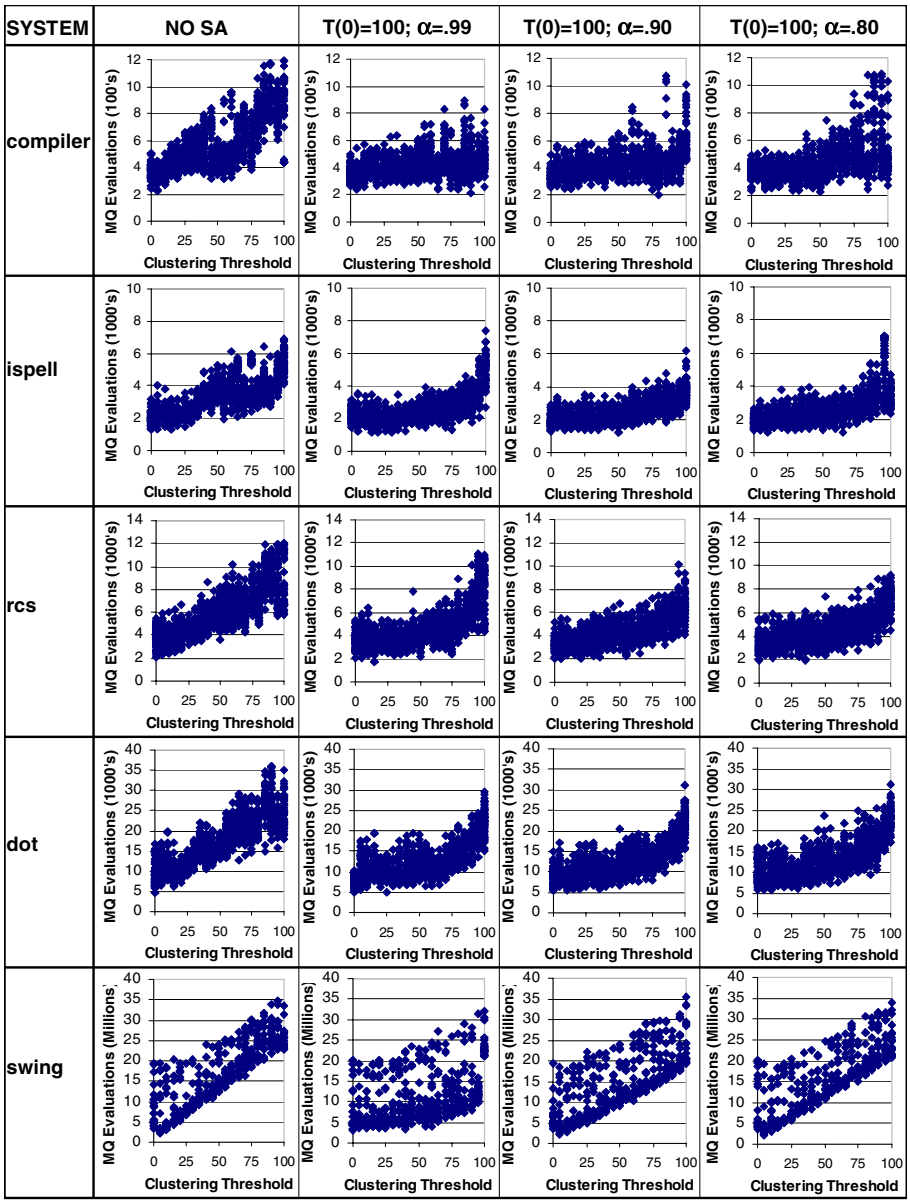


Figure 7: Case Study Results – MQ Evaluations versus η Scatter Plot

feature. Each of these tests altered the parameters used to initialize the *cooling function* that was described in Section 4.2. For these tests we held the initialization value for the starting temperature constant, $T(0) = 100$, and varied the cooling rate as follows: $\alpha = \{0.99, 0.9, 0.8\}$.

The following observations were made based on the data collected in the case study:

- As expected, the clustering threshold η had a direct and consistent impact on the clustering runtime, and the number of *MQ* evaluations. As η increased so did the overall runtime and the number of *MQ* evaluations. This behavior is illustrated consistently in Figure 7.
- Figure 6 shows that although increasing η increased the overall runtime and number of *MQ* evaluations, altering η did not appear to have an observable impact on the overall quality of the clustering results. The data in Figure 6 also shows that our simulated annealing implementation did not improve *MQ*. However, the simulated annealing algorithm did appear to help reduce the total runtime needed to cluster each of the systems in this case study.

  Figure 7 shows the number of *MQ* evaluations performed for each of the systems in the case study. Since the overall runtime is directly related to the number of *MQ* evaluations, it appears that the use of our SA cool-

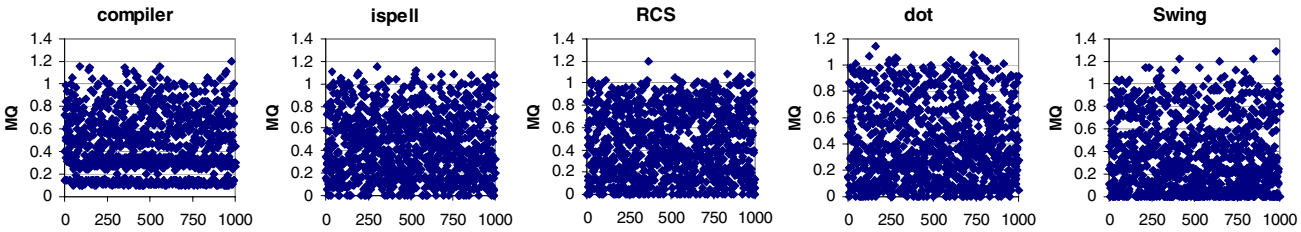


Figure 8: Case Study Results – Random Partition Scatter Plot

Table 1: Systems Examined in the Case Study

System Name	System Description & MDG Size
compiler	The compiler shown in Figure 2. MDG: modules = 13, relations = 32
ispell	An open source spell checker. MDG: modules = 24, relations = 103
rcs	An open source version control system. MDG: modules = 34, relations = 163
dot	A graph drawing tool (Gansner *et al.*, 1993). MDG: modules = 42, relations = 256
swing	The Java user interface class library. MDG: modules = 413, relations = 1513

ing technique is a promising way to reduce the clustering time. Table 2 compares the average number of *MQ* evaluations executed in each SA test to the average number of *MQ* evaluations executed in the non-SA test. For example using $T(0) = 100$ and $\alpha = .99$ reduced the number of *MQ* evaluations needed to cluster the swing class library by an average of 32%.

- Figure 6 indicates that the hill-climbing algorithm converged to a consistent solution for the ispell, dot and `rcs` systems.

- Figure 6 shows that the hill-climbing algorithm converged to one of two families of related solutions for the compiler and swing systems. For the compiler system, 53.7% of the results were found in the range $0.6 \leq MQ \leq 1.0$, and 46.3% of the results were found in the range $1.3 \leq MQ \leq 1.5$. For the swing system, 27.3% of the results were found in the range $1.5 \leq MQ \leq 2.5$, and 72.7% of the results were found in the range $3.75 \leq MQ \leq 6.3$.

- Figure 6 shows two interesting results for the ispell and rcs systems. For the ispell system, 34 out of 4,200 samples (0.8%) were found to have an *MQ* around 2.3, while all of the others samples had an *MQ* value in the range $0.8 \leq MQ \leq 1.2$. For the rcs system, 3 out of 4,200 samples (0.07%) were found to have an *MQ* around 2.2, while all of the other samples had *MQ* values concentrated in the range of $1.0 \leq MQ \leq 1.25$. This outcome highlights that some rare partitions of an *MDG* may be discovered if enough runs of our hill-climbing algorithm are executed.

- Figure 8 illustrates 1,000 random partitions for each system examined in this case study. The random partitions have low *MQ* values when compared to the clustering results shown in Figure 6. This result provides some confidence that our clustering algorithms produce better results than examining many random partitions, and that the probability of finding a good partition by means of random selection is small.

- As α increased, so did the number of simulated annealing (non-improving) partitions that were incorporated into the clustering process. In Figure 9 we show the number of SA partitions integrated into the clustering process for the swing class library. As expected, the number of partitions decreased as α decreased. Although we only show this result for swing, all of the systems examined in this case study exhibited this expected behavior.

Table 2: Reduced Percentage of *MQ* Evaluations Associated with using Simulated Annealing

	Simulated Annealing Parameters		
System	$T(0) = 100$ $\alpha = .99$	$T(0) = 100$ $\alpha = .90$	$T(0) = 100$ $\alpha = .80$
compiler	27%	26%	23%
ispell	22%	25%	21%
rcs	25%	27%	26%
dot	28%	29%	25%
swing	32%	15%	10%

6 Conclusions

This paper describes the latest enhancements that we made to our hill-climbing clustering algorithm, and examined several configuration parameters that impact the overall runtime and quality of the clustering results. A case study was also conducted to evaluate the impact of altering some of our clustering parameters when the Bunch tool was used to cluster several systems of varying sizes.

Figure 9: Case Study Results – Simulated Annealing Partitions used for Clustering swing

Our case study produced some interesting results, some of which were surprising. We expected that altering the clustering threshold η would either improve *MQ* or reduce variability in the clustering results, neither was found to be true. Also, although our simulated annealing technique did not impact *MQ* it did reduce the number of *MQ* evaluations, and therefore the overall clustering runtime for the systems that we examined. We also observed that our clustering algorithm tended to converge to a single "neighborhood" of related solutions for the ispell, rcs, and dot systems, and to two "neighborhoods" of related solutions for the compiler and swing systems. Finally, some rare solutions, with a significantly higher *MQ*, were discovered for the ispell and rcs systems.

As future work we intend to reevaluate our genetic algorithm in order to make some of the improvements that were suggested in this paper, and to investigate some alternative SA cooling functions.

7 Acknowledgements

This research is sponsored by grants CCR-9733569 and CISE-9986105 from the National Science Foundation (NSF). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

N. Anquetil, C. Fourrier, and T. Lethbridge. Experiments with hierarchical clustering algorithms as software remodularization methods. In *Proc. Working Conf. on Reverse Engineering*, October 1999.

Y. Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6, pages 177–208. John Wiley & Sons, New York, 1995.

S. Choi and W. Scacchi. Extracting and restructuring the design of large systems. In *IEEE Software*, pages 66–71, 1999.

A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *International Conference on Software Engineering, ICSM'99*, pages 246–255. IEEE Computer Society, May 1999.

D. Doval, S. Mancoridis, and B.S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of Software Technology and Engineering Practice*, August 1999.

E.R. Gansner, E. Koutsofios, S.C. North, and K.P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman, 1979.

D. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison Wesley, 1989.

D. Hutchens and R. Basili. System Structure Analysis: Clustering with Data Bindings. *IEEE Transactions on Software Engineering*, 11:749–757, August 1985.

S. Kirkpatrick, C.D. Gelatt JR., and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

J. Korn, Y. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of java applets. In *Proc. Working Conference on Reverse Engineering*, October 1999.

C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proc. International Conference on Software Engineering*, May 1997.

S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. 6th Intl. Workshop on Program Comprehension*, June 1998.

S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of International Conference of Software Maintenance*, pages 50–59, August 1999.

B. S. Mitchell, M. Traverso, and S. Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *The Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, August 2001.

M. Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, Cambridge, Massachusetts, 1997.

H. Müller, M. Orgun, S. Tilley, and J. Uhl. Discovering and reconstructing subsystem structures through reverse engineering. Technical Report DCS-201-IR, Department of Computer Science, University of Victoria, August 1992.

S. North and E. Koutsofios. Applications of graph visualization. In *Proc. Graphics Interface*, 1994.

R. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. 13th Intl. Conf. Software Engineering*, May 1991.

SERG, 2002. The Drexel University Software Engineering Research Group (SERG). http://serg.mcs.drexel.edu.

Code Factoring and the Evolution of Evolvability

Terry Van Belle
Department of Computer Science
University of New Mexico
Albuquerque, New Mexico, USA
vanbelle@cs.unm.edu
505-277-7833

David H. Ackley
Department of Computer Science
University of New Mexico
Albuquerque, New Mexico, USA
ackley@cs.unm.edu
505-277-9149

Abstract

Evolvability can be defined as the capacity of a population to evolve. We show that one advantage of Automatically Defined Functions (ADFs) in genetic programming is their ability to increase the evolvability of a population over time. We observe this evolution of evolvability in experiments using genetic programming to solve a symbolic regression problem that varies in a partially unpredictable manner. When ADFs are part of a tree's architecture, then not only do average populations recover from periodic changes in the fitness function, but that recovery rate itself increases over time, as the trees adopt modular software designs more suited to the changing requirements of their environment.

1 EVOLUTION OF EVOLVABILITY

Evolutionary biologists have explored the sometimes controversial notion that beyond merely producing organisms adapted to their environments, the forces of evolution may operate to improve the adaptation process itself (Dawkins 1989, for example). The idea is that over relatively long time periods populations can become *more capable of adaptation* in the face of future environmental change, a process described as "the evolution of evolvability."

While it may appear only logical that a 'more adaptable' population would ultimately outcompete and displace some other population that is less so, it seems a good deal less obvious when imagining how that competition would actually have to play out. If two individuals are equally fit in their environment there will be no direct pressure favoring the survival of one over the other, even if they vary drastically in how well-suited their designs would be for evolutionary adaptation to future changes. Even worse, if there is any near-term fitness cost associated with being adaptable for the future, we would expect such adaptability to dwindle rather than proliferate.

1.1 EVOLVABILITY IN ARTIFICIAL LIFE

Artificial life researchers have developed models in which the capacity of populations to adapt improves. Some approaches have involved mechanisms such as encoding the mutation rate inside of the genotype (Fogel, Fogel & Atmar 1991), using 'locking bits' to turn on and off the mutability of individual data bits (Turney 1999), and allowing variable gene ordering to encourage modularity (Pepper 2000). Although not providing experimental results, (Altenberg 1994) suggested that genetic programming (GP) experiments can exhibit an increase in evolvability through the proliferation of favorable blocks of code.

On the other hand, given an unchanging fitness function most typical genetic algorithms will converge to a small number of genotypes. As that happens, the average fitness of the population rises, but the ability of the population to adapt further declines, because less and less of the overall solution space remains easily reachable by applications of the genetic operators. In such cases, shorter-term fitness optimization is antagonistic to longer-term maintenance of evolvability.

1.2 CODE FACTORING

In software engineering, *code factoring* refers to the process of reorganizing the code within a program to improve its 'internal structure' in some manner, generally without changing what the program actually does (Fowler, Beck, Brant, Opdyke & Roberts 1999). Code factoring is used, for example, to merge duplicated pieces of code, and to separate more volatile code elements from a more stable code base. In this paper we explore a model that displays

an increase in evolvability based on evolutionary code factoring.

We applied genetic programming to a simple symbolic regression task with a repeated term. Unlike typical GP experiments, we periodically varied the value of this repeated term over the course of a run, thus changing the GP population's environment. We report on three experiments comparing trees that contained an ADF (Koza 1994) with trees that did not. We found that while both populations only adapted in temporarily constant environments, the ADF population also evolved forms capable of adapting more quickly to a changed environment—the evolvability of the ADF population evolved in a positive direction.

2 BACKGROUND

2.1 BIOLOGY AND SOFTWARE

(Kirschner & Gerhart 1998) point out how some biological mechanisms—such as versatile protein elements, weak linkage, compartmentation, redundancy, and exploratory behavior—can improve the evolvability of multicellular organisms. Such mechanisms can reduce the interdependence between components, allowing functionally independent traits to vary without adversely affecting each other.

At the same time, sometimes decreasing flexibility can be advantageous. (Dawkins 1996) cites bilateral symmetry as an example of a constraint that can improve evolvability. If longer legs, say, would improve fitness, a developmental process structured so that both legs lengthened equally as the result of a single mutation could have an advantage over another ontogeny that required a separate mutation for each leg to occur simultaneously. The former approach sacrifices the added flexibility of differing limb lengths, but we imagine that would be generally detrimental anyway. Evolvability is inherently about placing both flexibility and constraint where they are likely to help more than hurt, and involves betting on how the future is likely to be different from the present. If there are patterns in the environmental changes, evolvability may be able to gain traction.

Similar issues arise in software design (Altenberg 1994, for example). The environment to which the software must adapt includes the changing requirements of the software users (Nehaniv 2000, Stiemerling & Cremers 2000). Such changes are not completely random, and good software design tries to anticipate them.

For example, when designing code for a graphical user interface that contains some clickable buttons, one possible approach would be to write a completely separate module for each button. This allows great flexibility in that everything about the appearance and behavior of one button could be changed without any effect on the others, but essentially no software designer would even consider such an approach. Buttons in GUIs generally behave in largely similar fashions, so the code responsible for each button can be very similar as well. Software designers separate attributes that distinguish the buttons—location, label, and action, for example—and provide them as parameters modifying the behavior of a single piece of code. The designer can then maintain the common code for all buttons simultaneously.

Although both solutions—lots of nearly duplicated code versus parameterized common code—could precisely satisfy the immediate behavioral needs, one solution is likely to be more evolvable than the other. Designers of durable systems strive not only to satisfy current requirements, but also to be adaptable along the dimensions in which they believe the requirements will vary in the future. This, we hypothesize, is a key part of the distinction between a program which merely solves a problem and one which solves it with *good design*: The latter is more evolvable.

In both biological and computational systems, the environment within which an organism or piece of software must function is likely to be more constant along some dimensions, and more variable along others. Systems with high evolvability will be adaptable along the variable dimensions of the environment without disrupting those design elements that have adapted to the environment's constant dimensions.

2.2 MODULARITY IN GP

(Koza 1994) introduced a variant to genetic programming, known as "Automatically Defined Functions" (ADFs), to introduce modularity into genetic programming, which previously had typically involved only a single block of code per organism. With ADFs, each genotype contains multiple blocks of code. One is designated the "Result Producing Branch" (RPB), and is the code that calculates the tree's result. The RPB can make use of special nonterminals that make calls to other blocks of code—the ADFs—that are by convention identified `ADF0`, `ADF1`, ... Each ADF contains a pre-set number of arguments, usually defined by the experimenter, which it can access through special argument terminal nodes, conventionally labelled `ARG0`, `ARG1`, ... The function $(x-1)^2+(x-1)$, for example, could be represented by:

```
ADF0:  (- X 1)
RPB:   (+ (* ADF0 ADF0) ADF0)
```

In this example, `ADF0` takes no arguments and computes $x-1$; the RPB computes the final function by calling `ADF0` several times. The combination of `ADF0` and the RPB constitutes a single genome.

(Koza 1994) demonstrated that introducing ADFs produced a near-universal improvement both in computational effort and in code size over equivalents without ADFs, provided the problem complexity exceeds a certain threshold. In this paper we show that ADFs can also improve a GP genome's evolvability over time.

3 EXPERIMENTAL FRAMEWORK

To investigate evolvability, we need both a dynamic environment and some evolutionary organisms. Here we present the models we used for each.

3.1 A DYNAMIC ENVIRONMENT

The environmental task was to perform symbolic regression on the function

$$y = A\sin(Ax) \tag{1}$$

where A is a constant, and x ranges from -1 to 1. Although A was held constant during fitness evaluations, it was changed periodically on a longer time scale, causing the fitness function to vary along a single dimension, while keeping all other dimensions constant. As software designers—seeing that A appears twice in the objective function and armed with the foreknowledge that A will vary over evolutionary time—we can readily conclude that it would be advantageous to factor out the computation of A and reuse that code; the question was whether 'blind evolution' would be able to see that as well.

Every generation, 200 x values were generated uniformly at random from the interval $[-1,1)$, and the corresponding y values were obtained from Equation 1 using the current A. A tree's fitness was calculated by evaluating it on the 200 current x's, and summing the absolute values of the differences between the correct y value and what the tree produced. For display purposes, we also computed the number of *hits* of a tree—the number of sample points where the calculated result was within 0.1 of the correct y value. Any tree scoring 200 hits was considered a *correct* tree, regardless of how it accomplished that performance.

Each run consisted of 1000 generations, evenly divided into *epochs* of L generations each. At the end of each epoch, a new value for A was selected uniformly at random from the range $[0,6)$.[1] Fitness of the best of generation was reported at the start of each epoch (immediately after A had been selected), and at the end of each epoch (after L generations of evolution had elapsed). Figure 1 summarizes the procedure used for the dynamic environment.

[1] Since $-A\sin(-Ax) \equiv A\sin(Ax)$, negative values of A would not add any problem complexity.

1. Update A
2. Generate 200 random sample points
3. Calculate fitnesses at the start of the epoch
4. Loop for L generations:
   4.1. Evolve for a generation
   4.2. Regenerate 200 random sample points
5. Calculate fitnesses at the end of the epoch
6. Calculate evolvability
7. Go to step 1

Figure 1: Experimental framework for a dynamic environment. See text for details.

3.1.1 Evolvability Defined

Within that environment, we defined the evolvability of the population at each epoch to be the following:

$$E = \frac{F_e - F_s}{L} \tag{2}$$

where F_s was the population's best fitness (measured in hits) just after A was changed and F_e was the best fitness at the end of L generations of evolution beyond that point.

3.2 EVOLUTIONARY ARCHITECTURES

We compared a 'monolithic' tree architecture containing no ADFs with an 'ADF' tree architecture that provided a single zero-argument ADF. In Experiment 1, below, the terminal node 'x' was not allowed in the ADF, so the ADF could only produce a constant value. In the monolithic configuration, a single branch calculated the entire function.

We used `lil-gp 1.1` (Punch & Goodman 1995), to run the experiments. A summary of the details can be found in Table 1. The percentage indicated for each operator gives the probability it will be used to produce an offspring. The 'Best' operator simply reproduces the best genome from the previous generation, then the second best, and so on for as many times as it is called, providing a kind of probabilistic elitism.

4 RESULTS

For each experimental configuration, we collected statistics such as the fitness (measured in hits) at the beginning and end of each epoch, the evolvability E, and the sizes of the various branches in numbers of nodes. All values reported

Population Size	1000
Generations	1000
Function	$y = A\sin(Ax)$
Fitness	sum of absolute value of error for 200 points
Fitness Reported	number of hits (max 200)
Operators	Crossover (80%), Mutation (10%), Reproduction (5%), Best (5%)
Crossover	Branch Typing
Selection	Generational, Tournament, Tournament Size = 7
Non-terminals	$\sin, \cos, \log, \exp, +, -, *, /$
Terminals	Random constant in $[-1, 1]$. RPB of ADF: x, ADF0. Monolithic: x.
Architecture	Monolithic vs. one 0-arg ADF
Update of A	Uniform random from $[0, 6)$
Epoch Length	$L = 5$ generations (except in Section 4.3)

Table 1: Details of Experiment 1

were measured from the best of generation, and averaged over 100 runs.

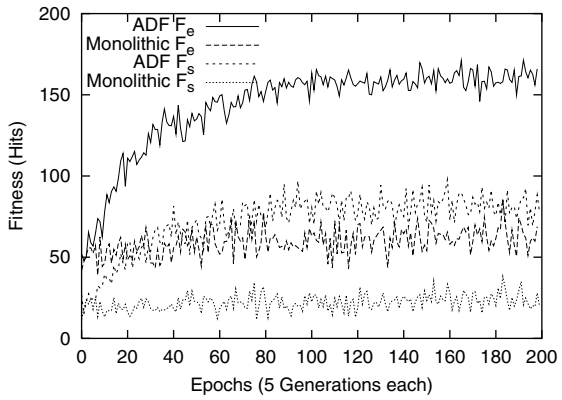


Figure 2: Average fitness values at the start (F_s) and end (F_e) of each epoch when regressing to $y = A\sin(Ax)$. A is selected at the start of each epoch uniformly from the range $[0, 6)$.

4.1 ADFS AND EVOLVABILITY

The first experiment asked if ADFs can increase average evolvability of a population compared to their absence. The results can be seen in Figures 2 through 5. The x axis is measured in epochs.

Figure 2 shows fitnesses, while Figure 3 presents the data

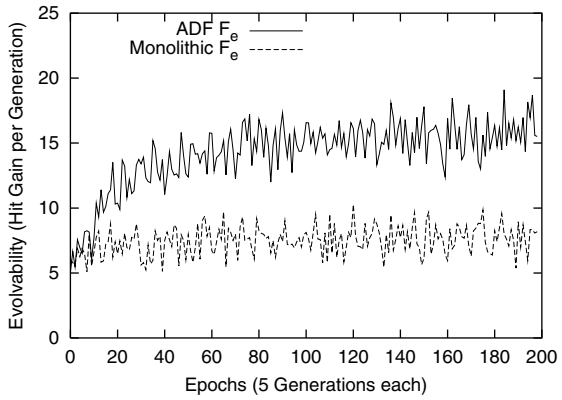


Figure 3: Average evolvabilities for each epoch, regressing to $y = A\sin(Ax)$.

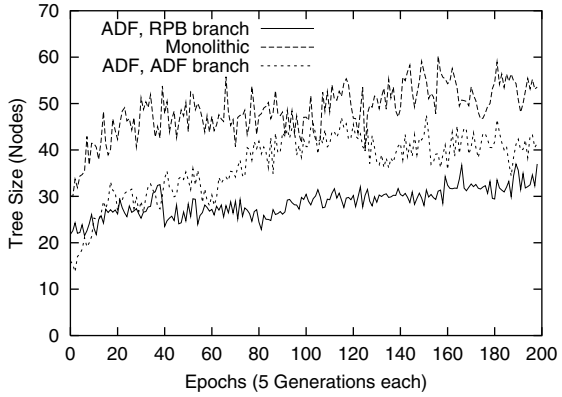


Figure 4: The average sizes of the three major branches. The ADF case shows sizes for both the RPB and the ADF; the monolithic size is the entire tree.

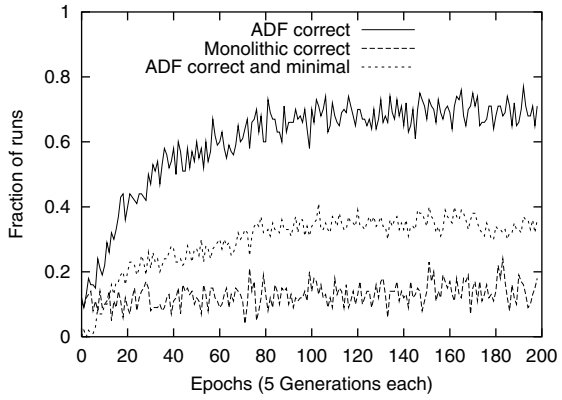


Figure 5: Fraction of runs, at each epoch, that contained correct solutions (hits = 200). Also plotted is the percent of ADF runs that contains a minimal (RPB size 6) correct solution—the tree size of Equation 3.

expressed as evolvabilities. The gap between the start (F_s) and end (F_e) fitnesses increases over time when ADFs are part of the tree architecture, but shows little to no increase when they are not; only when the trees contained ADFs did the best of generation average evolvability increase substantially over time.

An unexpected result visible in Figure 2 is that F_s increased over time when ADFs were part of the tree architecture. This is somewhat surprising, since F_s is calculated immediately after a new random value had been chosen for A, and before any evolution using that value has occurred. The ADF populations not only improved their capacity to track the changing environment, but as time went on, the populations became seeded with good solutions along the dimension of varying A.

Figure 4, showing the average tree sizes over time, suggests that the environmental change also had the effect of curbing any substantial tree size increases, so 'code bloat' (Blickle & Thiele 1994) was not a problem. The average sizes of all branches increased at most slowly, staying below 60 nodes.

A 'poster child' best-of-generation solution in the ADF case, taken from the end of the last epoch of run 10, looks like this:

```
RPB:  (* ADF0 (sin (* ADF0 X))
ADF0: (+ -0.52751 (- 0.03383
         (+ (sin -0.84486) -0.07376)))
```

This is an example of an ideally structured RPB. The calculation of the constant A has been moved to the `ADF0` branch, and the form of the RPB:

$$y = \mathrm{ADF0}() \cdot \sin(\mathrm{ADF0}() \cdot x) \tag{3}$$

matches that of Equation 1. The RPB contains six nodes (two multiplications, two ADF calls, one $\sin()$ call and one access to x), which is minimal for a correct general solution, given the set of terminals and non-terminals available. 36% of the runs ended with a solution of 200 hits and an RPB of size 6 at the end of the last epoch. Figure 5 shows that both the percent of correct solutions, and of correct minimal solutions rises substantially over the course of the ADF run, while staying fairly constant in the monolithic case.

Not all minimal correct trees will represent an ideal solution like Equation 3—it is possible to generate a non-general correct solution of size 6. However, manual examination of the last generation of the ADF case indicates that all but one of the runs was an ideal solution, or one of the form

```
RPB:  (/ (sin (/ X ADF0)) ADF0)
```

where the ADF calculates the reciprocal of A. The one exception had had the correct form at the end of the penultimate epoch, but had succumbed to an A-specific approximation during the last epoch.

By way of contrast, we ran 100 runs using a static fitness function, where the value for A never varied from 3 for the entire run. In this case, only 2 runs provided solutions of the ideal form *at any point* in the run.

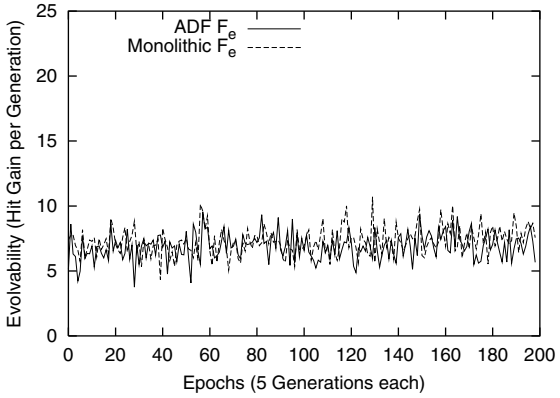


Figure 6: Average evolvabilities per epoch, regressing to $y = A\sin(Bx)$, with A and B varying independently. Compare to Figure 3.

We also performed a control experiment in which all conditions were identical to Experiment 1 except that the function being optimized was $y = A\sin(Bx)$, where A and B varied *independently* at random over precisely the same range. In that case there is no code reuse and thus should be no advantage to factoring the computation, and as Figure 6 shows, the ADF evolvability advantage completely disappeared in this case.

Experiment 1 strongly supports the idea that factoring the repeated and varying portion of the environment into a separate function aided the long-term successful populations by reducing the number of changes necessary to adapt to a new value of A. Though trees arose in ADF populations that did not perform such a factorization and still achieved correct solutions, such non-factored trees were less evolvable than the ones that factored, and so they and their descendents tended eventually to be out-competed.

4.2 EVOLVING CONSTANCY

Experiment 1 disallowed x in the body of the ADF, thus constraining the ADF to only produce constant values. Since that could bias solutions toward the 'more intuitive' factored representation, we also tested architectures without that constancy constraint.

In Experiment 2 the terminal node x was included in the ADF's terminal set, so the only remaining difference be-

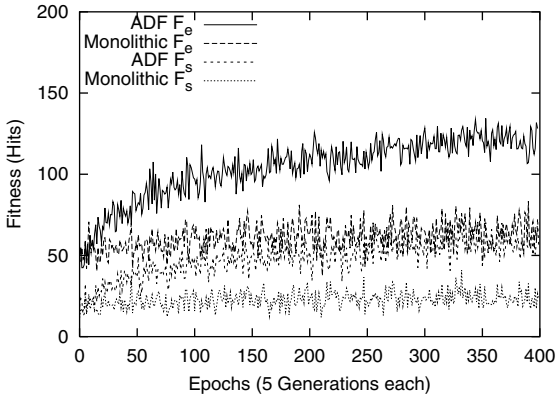


Figure 7: Experiment 2: Average F_s and F_s values when x is allowed in the ADF. The number of epochs is doubled over Experiment 1.

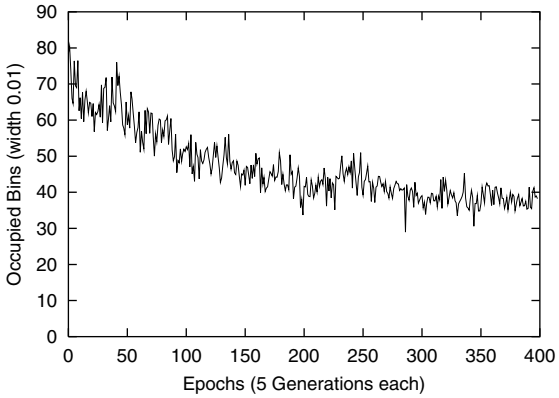


Figure 9: The average number of occupied bins returned by the evolved ADFs over time, given 200 random inputs. Lower numbers (minimum 1) imply 'more constancy'.

tween the RPB and the ADF was that the RPB could call the ADF but not vice-versa. The run length was doubled to 400 epochs over 2000 generations. The parameters were otherwise identical to Experiment 1.

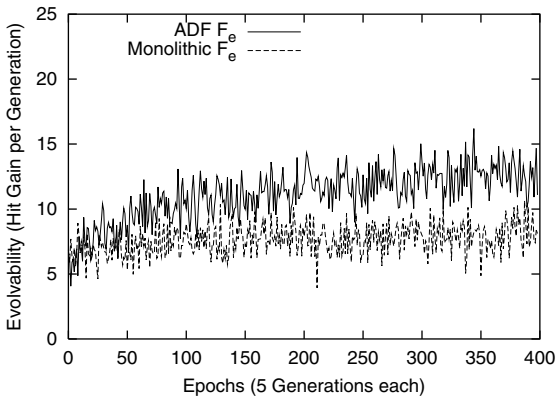


Figure 8: Average evolvabilities over time when x can be used in the ADF. The number of epochs is doubled over the first experiment.

The results can be seen in Figures 7 and 8. Increasing evolvability was still observed, though less pronounced than in Experiment 1. 22% of the runs produced ideal, minimal, correct solutions.

We wondered whether that increasing evolvability corresponded to increasing constancy in the ADF, but determining an ADF's constancy in a satisfying way is somewhat problematic. Simply counting the number of x's in the ADF's code fails because the x nodes may be contained inside of *introns*, non-functional blocks of code such as $0 \times x$, or $x - x$ (Angeline 1994).

Another approach might be to measure the variance of ADF(x) sampled over many values of x, but that tends to be highly sensitive to occasional large-magnitude outliers that skew the averages across runs. The procedure we eventually used is as follows: We called ADF(x) on 200 random points in the range $[-1,1)$ and quantized the return values into bins of size 0.01, so that results were judged identical if they rounded to the same nearest hundredth. We could then simply count the number of distinct occupied bins as a crude measure of ADF constancy. A completely constant function would produce only one occupied bin.

Figure 9 shows that the average number of ADF occupied bins declined significantly over time, suggesting that the ADFs indeed tended towards constancy.

4.3 VARYING EPOCH LENGTHS

In general, there is every reason to expect evolvability effects to depend on the rate of change of the environment. At one extreme, there is no advantage to maintaining a flexible design for future change if no environmental change is ever forthcoming; at the other extreme if there too much change too frequently then no effective adaptation will be possible at all.

In our model, the length of an epoch provides a natural index of the rate of environmental change. In Experiment 3, the epoch length L was varied over the values L = 1, 2, 5, 10, 20, 50, 100, and 200 generations, resulting in runs that ranged from 5 to 1000 epochs. All other parameters were the same as in Experiment 1. In particular, the ADF was not allowed to use x.

The average over 100 runs of the last-epoch F_s and F_e values can be seen in Figure 10 as a function of L. The largest F_s occurs at $L = 2$, while the maximum F_e occurs at $L = 5$. At present we are unsure why the ADF F_e rises from $L = 50$

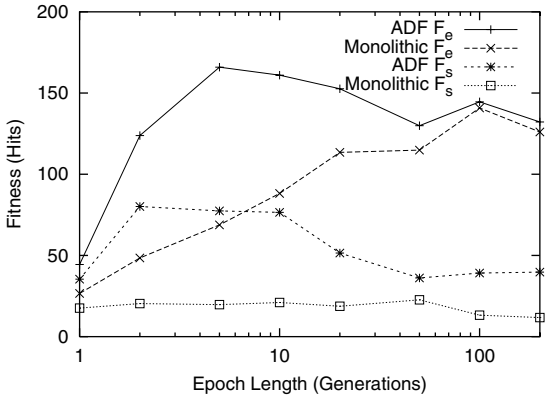


Figure 10: Average starting (F_s) and ending (F_e) fitnesses for the last epoch, for various epoch lengths.

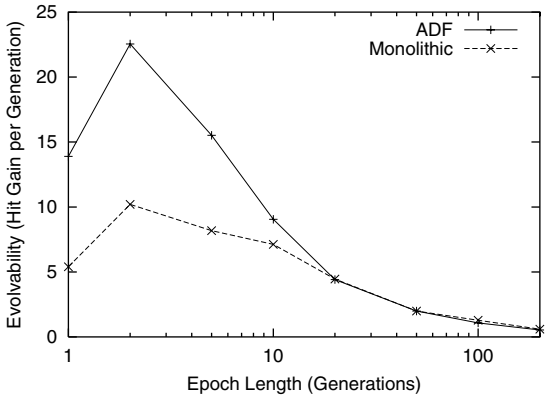


Figure 11: Average evolvability E for the last epoch, for various epoch lengths.

to $L = 100$. One possibility is that $L = 5$ is close to the optimal value when evolvable architectures are the norm, while $L = 100$ is close to optimal when evolvability is not an issue.

As epoch lengths increase, the monolithic F_e increases and F_s declines slowly, in line with expectations that the longer intra-epoch periods allow more evolution but then the inter-epoch changes are more disruptive. In both ADF and monolithic cases, the evolvability peaks at $L = 2$ (Figure 11), suggesting that diminishing returns set in rapidly when measured on a gain-per-generation basis.

Figure 12 shows the number of correct solutions for ADF and monolithic cases, as well as the correct and minimal solutions for the ADF case. The latter value is largest at $L = 5$, and is unaffected by the $L = 100$ resurgence that occurs for F_e and correct solutions, suggesting the $L = 100$ rise may not be due to improved evolvability.

As the epoch lengths increase, the tree sizes in the mono-

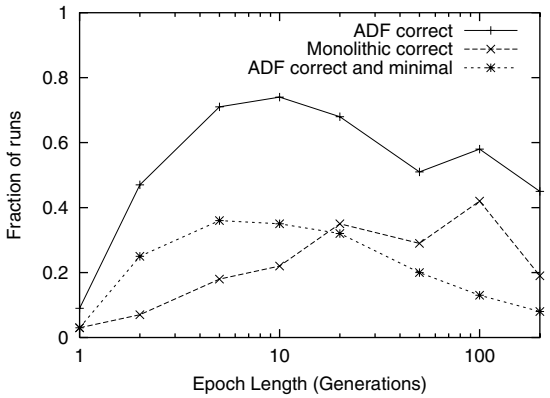


Figure 12: Fraction of ADF and monolithic runs that achieve correct (hits = 200) solutions at the end of the last epoch. Also plotted is the percent of ADF runs that are correct *and* have RPBs with a size of 6.

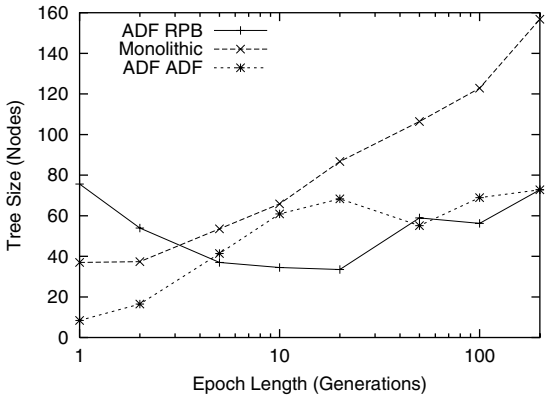


Figure 13: Average tree sizes for the last epoch, for various epoch lengths. Plotted are the size of the RPB in the ADF and monolithic configurations, as well as the ADF size in the ADF configuration.

lithic case increase steadily, but in the ADF case, the RPB average size reaches a minimum at $L = 20$ (Figure 13). The corresponding ADF size seems to vary inversely to that of the RPB; reasons for that effect are presently obscure.

5 CONCLUSION

In this paper we have presented a model, based on genetic programming, which demonstrates the evolution of evolvability when solving a symbolic regression task with a periodically changing fitness function. The successful solutions improved their evolvability by adopting forms that segregated the reused and variable portion of the fitness function (the A parameter), from the unitary and constant portion ($y = \diamond \sin(\diamond x)$). Many intriguing questions are open at this point, from detailed issues of the relative effects of redundancy and variability, to more fundamental goals such as the evolutionary emergence of other software engineering principles, and the scaling up of this research to real world problems.

Well-factored code is not strictly *required* to make a program operate correctly, and bold young programmers often use precisely that argument to resist such basic principles of 'code hygiene'. We have demonstrated how effective code factorings can emerge from an evolutionary process under a variety of appropriate conditions, even though the fitness function guiding the evolution is—like the novice programmer—focused entirely on the external program behavior, and not at all on its internal structure.

Thus, we establish an experimental link between the evolution of evolvability experiments previously published, and the body of knowledge that forms conventional wisdom about good software design. Though the gap between these two fields is still large, this paper represents a step towards bridging that gap.

Acknowledgments

This research was supported in part by DARPA contract F30602-00-2-0584, and in part by NSF contract ANI 9986555.

References

Altenberg, L. (1994), The evolution of evolvability in genetic programming, *in* K. E. Kinnear, Jr., ed., 'Advances in Genetic Programming', MIT Press, pp. 47–74.

Angeline, P. J. (1994), Genetic programming and emergent intelligence, *in* K. E. Kinnear, Jr., ed., 'Advances in Genetic Programming', MIT Press, chapter 4, pp. 75–98.

Blickle, T. & Thiele, L. (1994), Genetic programming and redundancy, *in* J. Hopf, ed., 'Genetic Algorithms Within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)', Saarbrücken, Germany, pp. 33–38.

Dawkins, R. (1989), The evolution of evolvability, *in* C. G. Langton, ed., 'Artificial Life: The Proceedings of an Interdisciplinary Workshop on the Synthesis and Simulation of Living Systems', Vol. 6, Addison-Wesley, Redwood, CA, USA, pp. 201–220.

Dawkins, R. (1996), *Climbing Mount Improbable*, W. W. Norton and Company, New York.

Fogel, D. B., Fogel, L. J. & Atmar, J. W. (1991), Meta-evolutionary programming, *in* R. R. Chen, ed., 'Proceedings of the 25th Asilomar Conference on Signals, Systems, and Computers', Pacific Grove, CA, pp. 540–545.

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Roberts, D. (1999), *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Boston, MA.

Kirschner, M. & Gerhart, J. (1998), 'Evolvability', *Proceedings of the National Academy of Science, USA* **95**, 8420–8427.

Koza, J. (1994), *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, MA.

Nehaniv, C. L. (2000), Evolvability in biology, artifacts, and software systems, *in* 'Artificial Life 7 Workshop Proceedings', pp. 17–21.

Pepper, J. (2000), The evolution of modularity in genome architecture, *in* 'Artificial Life 7 Workshop Proceedings', pp. 9–12.

Punch, B. & Goodman, E. (1995), 'lil-gp1.1 genetic programming system'. *http://garage.cps.msu.edu/software/lil-gp/ lilgp-index.html

Stiemerling, O. & Cremers, A. B. (2000), A paleontological perspective on designing adaptable software, *in* 'Artificial Life 7 Workshop Proceedings', pp. 26–29.

Turney, P. D. (1999), Increasing evolvability considered as a large-scale trend in evolution, *in* A. Wu, ed., 'Proceedings of 1999 Genetic and Evolutionary Computation Conference Workshop Program (GECCO-99 Workshop on Evolvability)', pp. 43–46.