Reformulating Software Engineering as a Search Problem *

John Clarke,
The University of York
Heslington, York,
YO10 5DD, UK.
Tel: +44 (0)1904 432722
Fax: +44 (0)1904 432767
jac@cs.york.ac.uk

José Javier Dolado,
Facultad de Informática,
University of the Basque Country,
20.009, Spain,
Tel: +34 943 018053
Fax: +34 943 219306
dolado@si.ehu.es

Mark Harman,
Robert Hierons,
Brunel University,
Uxbridge, Middlesex,
UB8 3PH,UK.
Tel: +44 (0)1895 274 000
Fax: +44 (0)1895 251 686
Mark.Harman@brunel.ac.uk
Rob.Hierons@brunel.ac.uk

Bryan Jones,
University of Glamorgan,
Pontypridd,
CF37 1DL, UK.
Tel: +44 (0)1443 482730
Fax: +44 (0)1443 482715
bfjones@glam.ac.uk

Mary Lumkin,
Systems Integration Research,
British Telecom,
Adastral Park, Ipswich,
UK. IP5 3RE
Tel. +44 (0)1473 644191
Fax. +44 (0)1473 648809
Mary.Lumkin@bt.com

Brian Mitchell,
Spiros Mancoridis,
Department of Mathematics
& Computer Science,
Drexel University,
Philadelphia, PA, USA, 19104.
Tel: +1 (215)895-2668
Fax: +1 (215)895-1582
bmitchel@mcs.drexel.edu
smancori@mcs.drexel.edu

Kearton Rees,
Systems Integration Research,
British Telecom,
Adastral Park, Ipswich,
UK. IP5 3RE
Tel. +44 (0)1473 644191
Fax. +44 (0)1473 648809
Kearton.Rees@bt.com

Marc Roper,
Strathclyde University,
Livingstone Tower,
26 Richmond Street,
Glasgow G1 1XH, UK.
Tel: +44 (0)141 548 2956
Fax: +44 (0)141 552 5330
marc@cs.strath.ac.uk

Martin Shepperd,
Bournemouth University,
Talbot Campus,
Poole, BH12 5BB, UK.
Tel: +44 (0)1202 595034
Fax: +44 (0)1202 595314
mshepper@bournemouth.ac.uk

Abstract

Metaheuristic techniques such as genetic algorithms, simulated annealing and tabu search have found wide application in most areas of engineering. These techniques have also been applied in business, financial and economic modelling. Metaheuristics have been applied to three areas of software engineering: test data generation, module clustering and cost/effort prediction, yet there remain many software engineering problems which have yet to be tackled using metaheuristics. It is surprising that metaheuristics have not been more widely applied to software engineering; many problems in software engineering are characterised by precisely the features which make metaheuristic search applicable.

In this paper it is argued that the features which make metaheuristics applicable for engineering and business applications outside software engineering, also suggest that there is great potential for the exploitation of metaheuristics within software engineering.

The paper briefly reviews the principal metaheuristic search techniques and surveys existing work on the application of metaheuristics to the three software engineering areas of test data generation, module clustering and cost/effort prediction. It also shows how metaheuristic search techniques can be applied to three additional areas of software engineering: maintenance/evolution, system integration and requirements scheduling. The software engineering problem areas considered thus span the range of the software development process, from initial planning, cost estimation and requirements analysis, through to integration,

*This work is part-funded by EPSRC SEMINAL (Software Engineering using Metaheuristic INovative ALgorithms) network (GR/M78083), http://www.discbrunel.org.uk/seminal .

maintenance and evolution of legacy systems. The aim is to justify the claim that many problems in software engineering can be re-formulated as search problems, to which metaheuristic techniques can be applied.

The goal of this paper is to stimulate greater interest in metaheuristic search as a tool of optimization of software engineering problems and to encourage the investigation and exploitation of these technologies in finding near optimal solutions to the complex constraint-based scenarios which arise so frequently in software engineering.

1 Introduction

Many of the problems associated with complexity management of the software development process have proved resistant to conventional analytic solutions. The software development process and its products tend to be characterised by a large number of competing and inter-related constraints. Some of these are clearly specified, while others are notoriously vague and poorly defined. Changes to one parameter can often have a large impact upon several related areas, making the balancing of concerns very difficult.

If there were only a single solution to a typical set of software engineering constraints, then software engineering would probably not be regarded as 'engineering' at all. The authors take as their premise the claim that software engineers face problems which consist, not in finding *the* solution, but rather, in engineering an *acceptable* or *near optimal solution* from a large number of alternatives. Often it is not clear how to achieve an optimal solution, but it is more clear how to evaluate and compare candidates. For instance, it may be hard to know how to achieve a design with low coupling and high cohesion, yet relatively easy to decide whether one design is more closely coupled than another.

Metaheuristic search techniques are a set of generic algorithms which are concerned with searching for optimal or near optimal solutions to a problem within a large multi-modal search space. These search spaces typically arise when a number of competing constraints have to be balanced against one another to arrive at a solution which is 'good enough'. Typically, for these problems, it is infeasible to apply a precise analytic algorithm which produces the 'best' solution to the problem, yet it is possible to determine which is the better of two candidate solutions. Metaheuristic Algorithms [16, 22, 23, 26, 33, 44, 57] have been applied successfully to a number of engineering problems ranging from load balancing in the process industries (pressing of sugar pulp), through electromagnetic system design, to aircraft control and aerodynamics [68].

It is surprising that technologies such as metaheuristic search have not penetrated the software engineering research community and are not widely applied when compared with the more traditional engineering disciplines. This paper argues that software engineering problems can often be thought of as search problems and thus metaheuristic search can be applied to a whole range of software engineering problems such as requirements gathering, systems integration and evolution.

The aim of the paper is to provide a new perspective on software engineering problems, in which they are reformulated as search problems to which Metaheuristic Algorithms can be applied. This new perspective should be regarded as complementary and supplementary to existing approaches, rather than being a replacement. The paper argues that metaheuristic search techniques simply provide another technology with which many software engineering problems can be attacked. In order to justify this claim, the paper surveys existing successful results in the application of metaheuristic search techniques to automated software test data generation, module clustering and cost/effort prediction and provides a reformulation of three other software engineering problem areas as metaheuristic search problems. Together, the six areas considered span the software development process from scheduling and requirements, cost and effort estimation, through system integration to source transformation for maintenance and re-engineering.

The hope is that the paper will stimulate greater interest in metaheuristic search technologies within the software engineering community, leading to further research into the application of these techniques to solving the complex sets of constraints which make software development and evolution such a demanding process.

The rest of this paper is organised as follows. Section 2 provides a brief overview of the field of Metaheuristic Algorithms. Section 3 surveys previous work on the application of Metaheuristic Algorithms to software testing, module clustering and cost estimation. Section 4 shows how software engineering can be thought of as a search problem to which Metaheuristic Algorithms can readily be applied. Section 5 concludes with directions for further work.

2 Metaheuristic Search Techniques

2.1 Optimisation

Optimisation seeks to find the set of parameters for which an objective function has a maximum or minimum value. The objective function may have a number of local extreme points in addition to the global extremum. Metaheuristic algorithms are a set of techniques that have been successfully applied to solving optimisation problems in the presence of many local extrema, with many parameters and in the presence of conflicting constraints. They have been used to find acceptable approximations to the solution of many NP complete problems.

Examples of meta heuristic algorithms are local search-based techniques such as hill climbing, simulated annealing and tabu search, and evolutionary techniques such as genetic algorithms. These are described briefly in the following two sections.

The key ingredients of any search-based optimisation problem are:

- The choice of the representation of the problem;
- The definition of the fitness function.

2.2 Local Search

This section overviews three local search metaheuristic techniques: hill climbing, simulated annealing and tabu search.

2.2.1 Hill Climbing

In hill climbing, the search proceeds from a randomly chosen point by considering the neighbours of the point. Once a fitter neighbour is found this becomes the 'current point' in the search space and the process is repeated. If there is no fitter neighbour, then the search terminates and a maxima has been found (by definition). The approach is called hill climbing, because when the fitness function is thought of as a landscape, with peaks representing points of higher fitness, the hill climbing algorithm selects a hill near to the randomly chosen start point and simply moves the current point to the top of this hill (climbing the hill).

Clearly, the problem with the hill climbing approach is that the hill located by the algorithm may be a local maxima, and may be far poorer, in terms of fitness, than the global maxima in the search space. However, hill climbing is a simple technique which is easy to implement and has been shown to use useful and robust in the software engineering applications of modularization [24, 40] and cost-estimation [32] (see also Sections 3.2 and 3.4).

There are many variations of the hill climbing theme. For example, should the algorithm select the first neighbour it finds with better fitness than the current individual (first ascent hill climbing) or should it consider all of the neighbours and select that with the best fitness improvement (steepest ascent hill climbing)? Whichever variation is chosen, the important characteristic of hill climbing is the property that it will select a local peak, which may be a local optimum and once this local peak is found the algorithm terminates; fast, simple, but prone to sub-optimality. Simulated annealing and tabu search are local search techniques which also search a local neighbourhood within the search space, but each has a different approach to overcoming this tendency to become trapped by a local maxima.

The key ingredient of the hill climbing algorithm is:

The definition of a neighbourhood on the configuration space.

2.2.2 Simulated Annealing

Simulated annealing [38] is a method of local searching, in contrast to genetic algorithms which sample the whole domain and improve the solution by recombination in some form. In simulated annealing, a value, x_1, is chosen for the solution, x, and the cost (or objective) function, E, is minimised. Cost functions define the relative undesirability of particular solutions. In general larger values of the function are associated with larger costs. Similarly, since we wish to satisfy constraints, values of the function's dependent variable that blatantly break constraints will attract a greater cost than those that satisfy all constraints, or come close to doing so.

```
Initialise x to x_0 and T to T_0

loop — Cooling
  loop — Local search
    Derive a neighbour, x′, of x
    ΔE := E(x′) − E(x)
    if ΔE < 0
    then x := x′
    else derive random number r ∈ [0, 1]
      if r < e^(−ΔE/T)
      then x := x′
      end if
    end if
  end loop — Local search
  exit when the goal is reached or a pre-defined stopping condition is satisfied
  T := C(T)
end loop — Cooling
```

Figure 1: A Generic Simulated Annealing Algorithm

When minimizing, the objective function is usually referred to as a cost function. When maximizing it is usually referred to as a fitness function.

The simulated annealing algorithm then considers a neighbouring value of x_1 and evaluates its cost function; what constitutes a neighbouring value of x_1 may not be immediately obvious and a neighbour, $x_1\prime$, must be defined in an appropriate way. If the cost function for $x_1\prime$ is reduced, the search moves to $x_1\prime$ and the process is repeated. However, if the cost function increases, the move to $x_1\prime$ is not necessarily rejected; there is a small probability, p, that the search will move to $x_1\prime$ and continue. The probability, p, is a function of the change in cost function, ΔE, and a parameter, T:

$$p = e^{-\frac{\Delta E}{T}}.$$

This probability is of a similar form to the Maxwell-Boltzmann distribution law for the statistical distribution of molecular energies in a classical gas, where ΔE and T are related to energy and temperature respectively. When the change in cost function is negative (i.e. an improvement), the probability is set to one and the move is always accepted. When ΔE is positive (i.e. unfavourable), the move is accepted with the probability given in the equation above; the probability depends strongly on the values of the ΔE and T. At the start of the simulated annealing, T is high and the probability of accepting a very unfavourable move is correspondingly high. During the search, T is slowly reduced by some function, C, called the 'cooling' function because of its counterpart in the physical world. The effect of 'cooling' on the simulation of annealing is that the probability of following an unfavourable move is reduced. In practice, the temperature is decreased in stages, and at each stage the temperature is kept constant until thermal quasi-equilibrium is reached. The set of parameters that determine the temperature decrement (i.e. initial temperature, stop criterion, temperature decrement between successive stages, number of transitions for each temperature value) is called the cooling schedule. The cooling schedule is critical to the success of the optimisation.

The algorithm is described in Figure 1.

The key ingredients of the simulated annealing algorithm are:

1. the definition of a neighbourhood on the configuration space;
2. the specification of the cooling schedule.

The term simulated annealing arises from the use of the Maxwell-Boltzmann probability function along with a procedure of cooling; metallurgical annealing involves heating a metal or alloy to a high temperature and

```
generate initial solution
loop
  Identify neighbourhood set
  Identify tabu set
  Identify aspirant set
  Choose the best move
  exit when goal is satisfied or the stopping condition is reached
end loop
```

Figure 2: A Generic Tabu Search Algorithm

cooling slowly to room temperature with the aim of removing internal stresses and thereby improving ductility and reducing brittleness. The technique originates from the theory of statistical mechanics and is based upon the analogy between the annealing of solids and solving optimisation problems. It is not helpful to pursue the analogy with the physical world to any greater extent.

The effectiveness of the search depends upon the initial temperature (is it high enough to randomise the search sufficiently?) and the subsequent cooling (is it slow enough to ensure that equilibrium is achieved at each temperature?).

2.2.3 Tabu Search

Tabu search [22] is an iterative procedure for solving discrete combinatorial optimisation problems. The space of all possible solutions is searched in a sequence of moves from one possible solution to the best available alternative. In order to prevent being stuck at a sub-optimal solution and to avoid drifting away from the global optimum, some moves are classified as forbidden or tabu (taboo). The list of tabu moves is formed using both short-term and long-term memory of previous unpromising moves. A move may also be regarded as 'unpromising' simply because it was recent. On occasions, a tabu move may be allowed. This is an aspiration criterion, whereby the tabu move might lead to the best solution obtained so far.

The search space, or neighbourhood, comprises a set of moves that lead to another solution when applied to the current solution. Tabu search starts at a possibly random point and determines a sequence of moves during which a tabu set is established; some members of the tabu set may be classified as a member of an aspirant set. The criteria for classifying aspiring moves are specific to the application. Moves may be tabu if they could lead to a solution that has already been considered (recency or short- term condition) or has been repeated many times before (frequency or long-term condition). The next move is the best neighbouring solution which is either not tabu or is an aspirant.

A generic tabu search algorithm is summarised in Figure 2.

The key ingredients for setting up a tabu search are to:

1. Define the neighbourhood of a solution;

2. Define an aspiring move.

2.3 Evolutionary Search Using Genetic Algorithms

Genetic algorithms (GAs) [6, 26] search for optimal solutions by sampling the search space at random and creating a set of candidate solutions called a 'population'. These candidates are combined and mutated to evolve into a new generation of solutions which may or may not be fitter. Recombination is fundamental to the GA and provides a mechanism for mixing genetic material within the population. Mutation is vital in introducing new genetic material thereby preventing the search from stagnating. The next population of solutions is chosen from the parent and offspring generations in accordance with a survival strategy that normally favours fit individuals but nevertheless does not preclude the survival of the less fit. In this way, a diverse pool of genetic material is preserved for the purpose of breeding yet fitter individuals. This terminology strongly suggests that GAs are based on the principles of Darwinian evolution. This analogy should not be taken too literally since Darwinian

Set generation number, $m := 0$
Choose the initial population of candidate solutions, $P(0)$
Evaluate the fitness for each individual of $P(0)$, $F(P_i(0))$
loop
  Recombine: $P(m) := R(P(m))$
  Mutate : $P(m) := M(P(m))$
  Evaluate: $F(P(m))$
  Select: $P(m+1) := S(P(m))$
  $m := m + 1$
  exit when goal or stopping condition is satisfied
end loop;

Figure 3: A Generic Evolutionary Algorithm

evolution is assumed blind and improvements happen by chance rather than by aspiring to a goal; improved individuals are those with a better chance of survival and thus they are able to pass their genes to their offspring.

Chance plays an important role in GAs, though their success in locating an optimum[1] strongly depends on a judicious choice of a fitness function. The fitness function must be designed carefully to reflect the nature of the optimum and to direct the search along promising pathways.

GAs operate on a population of individuals (often called chromosomes) each of which has an assigned fitness. The population size should be sufficiently great to allow a substantial pool of genetic material, but not be so large that the search degenerates into a random search. Those individuals that either undergo recombination or survive are chosen with a probability which depends on fitness in some way. There are many different selection mechanisms: in the so-called roulette wheel, the population's cumulative fitness is normalised to give a set of probabilities for each individual; in the N-tournament, N individuals are selected at random from a population and the fittest chosen.

A generic evolutionary algorithm is presented in Figure 3.

The key ingredients of the genetic algorithm are

1. The specification of probabilities of crossover and mutation;

2. The choice of recombination, mutation, and selection algorithms.

An advantage of GA is that the solution domain is sampled, though if the evaluation of the fitness function is computationally expensive, this advantage may rapidly become a disadvantage. In spite of sampling, there is no guarantee that the global optimum will be found and the search may rapidly stagnate at a sub-optimum unless steps are taken to preserve the diversity of genetic material.

2.4 Evolutionary Search Using Genetic Programming

In Genetic Programming [33], the goal is to produce a program which solves a particular problem. The fitness function is defined in terms of how close the program comes to solving the problem. The operators for mutation and mating are defined in terms of the program's abstract syntax tree. Because these operators are applied to trees rather than sequences, their definition is typically less straight forward than those applied to GAs.

Genetic programming can be used to find functions which describe (and predict) data. In this way the genetic program is an algorithmic fit to a set of data. The fit need not be perfect, in order to be useful. Indeed, many of the applications of genetic programming described by Koza [33], require not a perfect fully 'correct' solution, but merely one which is 'good enough'. Genetic programming can be used to find fits to software engineering data, such as project estimation data, as will be described in Section 3.4.

[1] or sub-optimum; in this section, the term optimum is taken to subsume sub-optimum.

2.5 The Use of Metaheuristic Algorithms to Solve Problems in Software Engineering

The strength of metaheuristic algorithms is that they seek iteratively an optimum solution within a landscape that may be extremely complicated and even discontinuous. The key to applying metaheuristic algorithms successfully is, first, to formulate the problem in hand as a search/optimisation problem. For instance, as will be seen in Section 3, the automated generation of test data can be formulated as a search within the input domain guided by a fitness function which captures an appropriate test adequacy criterion. Having formulated the problem as a search problem, it is necessary to define the essential ingredients relevant to the chosen metaheuristic.

All metaheuristic techniques require a suitable representation for the candidate solution (or 'individual') together with a fitness (or cost) function. The fitness function defines a 'fitness landscape' which gives the search a sense of direction; the search will be hindered by landscapes which have too shallow gradients and by landscapes with a few sharply-defined peaks or craters. The definition of suitable fitness functions for software engineering problems is therefore a major area of work required to under-pin the exploitation of metaheuristics in software engineering.

All search techniques rely upon a concept of a move from one individual to another. This can be either in the form of a 'move' to a 'near neighbour' or a 'mutation' which may produce a jump to a very different individual (i.e. not a 'near' neighbour). The available mutation and/or move operations that can be applied to an individual are typically delimited by the choice of representation.

The definition of representation, fitness function and mutation/move operations should be considered first, as these are a pre-requisite for the application of any of the metaheuristic search techniques outlined in this paper. Having defined these three key ingredients it becomes possible to apply hill climbing, simulated annealing and other local search techniques. In order to apply genetic algorithms it will be necessary to define a crossover operator.

A near neighbour may be defined as a predecessor or successor for an ordinal type and to be arbitrarily close for a floating type. Normally, the search space will be multi-dimensional and a decision must be made about how many values to change and in what way.

A crossover operator takes two individuals (the parents) and produces a new individual (the offspring) which is related to both parents, by sharing some of the information of each. In the literature, this process is regarded as an analog of sexual reproduction in living individuals (though typically, no gender distinction is made.)

In order to apply metaheuristics to software engineering problems the following steps should therefore be considered:

1. Ask: Is this a suitable problem?
   That is, "is the search space sufficiently large to make exhaustive search impractical?"
2. Define a representation for the possible solutions
3. Define the fitness function
4. Select an appropriate metaheuristic technique based on the definability of the 'key' ingredients' listed in section 2
5. Start will the application of simple hill-climbing. If the results are encouraging (i.e. better than random search) consider other local search and genetic approaches.

The motivation for the final piece of advice in this prescription for the application of metaheuristics is the observation [24, 32, 39, 40] that hill climbing may produce results which are sufficiently good to make the application of more sophisticated techniques an unnecessary additional effort. Of course, there are also landscapes in software engineering, such as the test data input landscape [60], where genetic approaches outperform hill-climbing. However, even for these landscapes, hill climbing represents a good starting point. If the results for the simple hill climbing approach are not better than those for a random search then this may indicate a lack of understanding of the problem area or that the representations used are perhaps inadequate. It could also be a reflection of the fact that the problem area simply is not suited to a search-based approach.

2.6 Problem Characteristics

This section describes some of the characteristics of the problem space which would lead to it being suitable for attack using a search-based solution. The section also contains examples of source engineering activity areas where search-based techniques are unlikely to prove successful, as a way of bounding the problem.

Likely criteria for success include

1. Large solution space.
   Building any artifact can be thought of as a search problem. The space of possible solutions has to be sufficiently large that enumeration of candidate solutions is impossibly expensive.

2. No known efficient and complete solution.
   Clearly there is little to be gained from applying a new solution to a problem which is already solved. However, search-based approaches may yield additional insight and may help to 'fill in gaps' where existing solutions are only partial.

3. The existence of suitable candidate fitness functions.
   Fields of software engineering activity which have a large number of readily accepted metrics are example of this.

4. Cheap generation of candidate solutions.
   A large solution space makes search-based solutions inevitable, but in order to apply them, the candidate solutions will need to be valuable in reasonable time. Typical search-based algorithms require many executions of the fitness function, so speed of execution of the fitness function is crucial.

Areas of software engineering activity which are unlikely to profit from search-based solutions include:

- Most forms of requirements elicitation;

- Aspects of human interaction;

- Situations where it is unclear what is required.

3 Existing Applications of Metaheuristics to Software Engineering

3.1 Testing

For the problem of generating test data for software, the individuals over whom optimization occurs are the test cases and these can be represented in ways that allow mutation, crossover and the notion of a neighbourhood. Crucially, testing normally aims to achieve certain measurable objectives. In fact, many test generation techniques are based around some notion of the coverage of the code or specification. This coverage can be measured and incorporated into an objective function.

Metaheuristics have been applied to the following types of testing

1. Structural testing;

2. Specification based testing;

3. Testing to determine the worst case execution time.

Each of these types of testing, and the application of metaheuristics to them, shall now be discussed.

3.1.1 Structural Testing

Structural, or white-box, test techniques determine the adequacy of a test set by considering the structure of the code. Normally such techniques consider certain constructs and measure the proportion of these constructs, in the source code, that are executed during testing. This proportion is called the *coverage*. Given a construct being considered, it is normal to insist on full coverage: all such (feasible) constructs are executed during testing.

A number of forms of coverage, including the following, are based on the control flow graph:

```
if (a>b+c || b>a+c || c>a+b) out = 'not a triangle';
else if (a==b && b==c) out = 'equilateral';
else if (a==b || b==c || a==c) out = 'isosceles';
else out = 'scalene';
```

Figure 4: A fragment of code for the triangle problem

1. *statement coverage*: The proportion of the reachable program statements that are covered during testing;
2. *branch coverage*: The proportion of the feasible program branches (edges of the control flow graph that leave a node that has more than one possible next node) that are covered during testing;
3. *path coverage*: The proportion of the feasible paths, through the control flow graph, that are covered during testing.

When considering a particular notion of coverage, random testing may provide a high level of coverage. There will often, however, be constructs that are unlikely to be executed by random testing and thus test generation may be broken down into two phases: use random testing to cover most constructs and then use some other technique to derive tests to cover the remaining constructs. A number of authors have considered the use of metaheuristics in the second phase [28, 29, 43, 54].

Suppose a construct has not been executed during testing. Then the tester may choose some path that contains this construct and try to derive test cases that follow this path. Consider the fragment of code (written in a `C` style notation), from a program produced to solve the triangle problem, shown in Figure 4. In order to test the `then` branch of the final `if` statement it is sufficient to follow the `else` branches of the first two `if` statements followed by this `then` branch.

Naturally a path that has been chosen might be infeasible and thus if a test generation process fails to execute the path for a significant length of time the tester might choose an alternative path. An objective function may be defined by, given a test input, measuring how close it gets to taking this path (up to the end of the construct being considered). This might be how many predicates it has in common with the intended path [43]. The fitness may be measured by instrumenting the code and the test input may be represented by a string of values or a bit string.

Suppose, again, that the tester wishes to test the final `then` branch of the code in Figure 4. If a test case takes the first `else` branch and then the second `then` branch, under the notion of fitness given above, this test case is given fitness 1.

Given a particular criterion, it may be possible to further refine the objective function. Jones et al. [28, 29], who consider the use of genetic algorithms for branch testing, give a fitness function for a test that reaches the initial vertex of the branch. The fitness function depends upon the values of the state when it reaches this vertex. Any test that fails to reach the vertex is given a low fitness. Assuming the test reaches the initial vertex of the branch being considered, the calculation of the fitness considers two cases: in the first case the test does not go down the correct branch and in the second it does go down the correct branch.

If the test fails to go down the correct branch, the fitness depends upon how close the state variables were, at that point, to executing the correct branch. The closer a test is to executing the correct branch, the higher its fitness. Consider, the `then` branch of the second `if` statement from Figure 4. If the input reaches the initial node of this (and thus follows the `else` branch of the first `if`) but follows the `else` branch at this point, the fitness might be inversely proportional to how close `a` is to `b` and how close `b` is to `c`. The fitness might then be

$$\frac{1}{(1+|a-b|+|b-c|)}.$$

If the test goes down the correct branch the fitness depends upon how close the state is to the boundary of the branch: the closer to the boundary the higher the fitness. Consider the `else` branch of the second `if` statement. If this branch is taken as the result of executing a test case, then again, the fitness might be inversely proportional to $|a-b|+|b-c|$. This second case is motivated by the observation that tests around boundaries are often effective at detecting faults and thus the fitness should drive the values towards the boundary of the branch. Again the fitness may be determined by instrumenting the code.

The fitness function described by [28, 29] could, potentially, be combined with that described in [43]. Given a branch, some path to this branch, might be chosen. If a test does not reach the initial vertex of the branch, then the fitness depends on how close the test is to executing the path. If the test reaches the initial vertex of the branch, then the total fitness depends upon that described in [28, 29]. The fitness described in [28, 29] might be called a *local fitness*, since it only considers the initial node of the branch. Then the total fitness, for a test that reaches the initial node of the branch, might be the local fitness plus the number of predicates on the path to the branch.

3.1.2 Specification-Based Testing

Tracey et al. [54] consider the problem of testing from a formal specification that is in the form of a set of (disjoint) pre/post conditions. For such a system, a test case leads to a failure if, for one of these pre/post pairs, it satisfies the precondition but fails the post condition.

For each pre/post condition pair (P, Q), the predicate $C(P, Q) = Q \vee \neg P$ is formed. Then a fault is detected if a test execution makes $C(P, Q)$ false. In order to try to drive test cases towards detecting faults, a fitness function, that depends on how close the test case is to making $C(P, Q)$ false, is used. $C(P, Q)$ is rewritten to disjunctive normal form and the conjuncts from this are considered in turn. Thus, each conjunct is tested with the intention to try to make it take on the value false. Naturally, the fitness must depend upon more than just whether the value is true or false. Instead, assuming the conjunct is true for the value being considered, each term in the conjunct contributes to the final fitness value which is the sum of these values (the problem is to minimize this fitness). For certain classes of predicate, such as $e_1 > e_2$ for expressions e_1 and e_2, a fitness function that gives a range of values may be defined. For example, with $e_1 > e_2$, the fitness is 0 if the predicate is false and otherwise the fitness is $e_1 - e_2$.

The authors use simulated annealing to try to find faults. Interestingly, the authors find a similar approach in which assertions are placed in the code with the intention of trying to make them false, can be used to test exception conditions [54].

3.1.3 Worst Case Execution Time

Real time control systems are often designed to respond to a stimulus within some fixed time interval. A failure to respond sufficiently within this interval may lead to a critical condition. Thus, it is often important to determine the worst case execution time of a function or method. The problem of determining the worst case execution time is often, however, highly complex as it may depend crucially on properties of the hardware and the compiler.

Genetic algorithms have been used in an attempt to find the maximum and minimum execution times [61, 62]. Given an input, represented by a chromosome, the fitness of this input depends upon the execution time: when trying to find maximal times the fitness is proportional to the execution time and when trying to find minimal times the fitness is inversely proportional to the execution time. Simulated Annealing has been used in a similar manner [55]. This approach not only consistently outperformed random testing but also found a new minimum execution time for one module [62].

3.2 Module Clustering

According to Shaw and Garlan [51], the software architecture of a system consists of a description of the system elements, the interactions between them, the patterns that guide their construction, and constraints on their relationships. For smaller systems the elements and relations may be modeled using source code entities such as procedures, classes, method invocation, inheritance, and so on. However, for larger systems, the desired entities may be abstract (high-level), and modeled using architectural artifacts such as subsystem components and relations.

Subsystems provide developers with structural information about the numerous software components, their interfaces, and their interconnections. Subsystems generally consist of a collection of collaborating source code resources that implement a feature or provide a service to the rest of the system. Typical resources found in subsystems include modules, classes, and possibly other subsystems. Subsystems facilitate program understanding by treating sets of related source code resources as high-level software abstractions. Subsystems can also be organized hierarchically, allowing developers to study the organization of a system at various levels of detail by navigating through the hierarchy.

The entities and relations needed to represent software architectures are not found in the source code. Thus, without external documentation, techniques capable of deriving a reasonable approximation of the software architecture from source code are needed. Research into the software clustering problem has proposed several approaches to deal with this challenge by defining techniques that partition the structure of a software system into subsystems (clusters). Most of these techniques determine clusters (subsystems) using either source code component similarity [42], sets of heuristic rules [48], concept analysis and clustering metrics [3, 27, 35, 56], or information available from the system implementation such as module, directory, and/or package names [4].

Although the above mentioned software clustering techniques have been shown to produce good results on certain types of systems, promising results have been obtained on a variety of different systems by applying heuristic-search techniques to the software clustering problem [39]. Mitchell and Mancoridis have studied hill-climbing, simulated annealing and genetic algorithm searches, and have implemented their software clustering algorithms as a suite of integrated tools that can be downloaded over the Internet [49]. Other researchers have examined how to modularize software systems using genetic algorithms [24, 36].

3.3 Software Clustering as a Search Problem

The starting point for applying search-based techniques to the software clustering problem is to formalize the representation of the structure of the system to be clustered. Based on this representation, a fitness function is required to measure the relative quality of the system modularization once its structure is decomposed into subsystems (clusters). Finally, an algorithm is required to traverse the space of candidate solutions, using the fitness function to locate a good solution from the enormous set of all possible solutions. Like most metaheuristic search problems, the result produced by the clustering algorithm need not be the optimal solution in a theoretical sense. Rather, it is a solution that is perceived by several people to be 'good enough'.

3.3.1 Representation

One of the goals of most software clustering algorithms is to be neutral of programming language syntax. To accomplish this goal, source code analysis tools can be used to transform the structure of the system's code into a language-independent directed graph. Graph G is formally defined as $G = (M, R)$, 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 (e.g. module u uses an exported function in module v). Also, G can have weighted edges to establish the 'strength' of the relation between a pair of modules.

The primary goal of software clustering algorithms is to propose subsystems (clusters) that expose abstractions of the software structure. Each partition of the software structure graph G is defined as a set of non-overlapping clusters that cover all of the graph's nodes. The goal is to partition the graph so that the clusters represent meaningful subsystems. Finding a 'good' partition involves navigating through the search space of all possible partitions of G in a systematic way.

It should also be noted that there are many ways to define G. For example, a *Module Dependency Graph* places an edge between a pair of modules if one module uses resources provided by the other module. Other graphs can be created by considering dynamic program behaviour (e.g. dynamic loading, object creation, runtime method invocation), inheritance relationships, and so on. Clustering alternative graph representations of the same software systems provides users with additional insight into the overall system architecture, which is helpful for activities such as program understanding and software maintenance.

3.3.2 Fitness Functions

When designing a fitness function for software clustering applications, a fundamental decision must be made about what constitutes a good partition of the software structure graph. One of the most popular approaches implemented by researchers is to define a fitness function that maximizes the cohesion of the individual clusters, while at the same time, minimizing the coupling between all of the clusters.

As an example, consider the fitness function implemented in the Bunch [37] clustering tool. The authors refer to their fitness function as *Modularization Quality* (MQ). *MQ* for a software structure graph G partitioned into k clusters is calculated by summing the *Cluster Factor* (*CF*) for each cluster of the partitioned graph. 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). The weight of the external edges is split in half in order to apply an equal penalty to both clusters that are connected by an external edge. Internal edges of a cluster are referred to as intra-edges. The number of intra-edges for module i is denoted by μ_i. The edges between two distinct clusters i and j are the inter-edges. The number of inter-edges between a module i and a module j is denoted as $\varepsilon_{i,j}$. If edge weights are not provided by G, it is assumed that each edge has a weight of 1. Also, note that $\varepsilon_{i,j} = 0$ and $\varepsilon_{j,i} = 0$ when $i = j$. The formal definition of the *MQ* calculation is defined as follows:

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

Notice how *MQ* increases as the cohesiveness of the individual clusters increases and the coupling between all of the clusters decreases. Thus the goal of the search algorithms implemented in Bunch is to maximize *MQ*. An alternative objective function that integrates the concept of cluster granularity into the fitness evaluation has been investigated by Harman et al [24]. Lutz [36] describes a fitness function for a GA that measures complexity based on coupling and cohesion principles.

3.3.3 Software Clustering Search Algorithms

One way to find the best partition of the software structure graph is to perform an exhaustive search through all of the valid partitions, and select the one with the best fitness value. However, this approach is often impossible because the number of ways the system can be partitioned into subsystems grows exponentially with respect to the number of its nodes (modules) [37].[2] Because discovering the optimal grouping of modules into clusters is only feasible for small software systems (e.g. fewer than 15 modules), heuristic search algorithms are required to locate acceptable results quickly.

Mitchell and Mancoridis have designed and implemented the following software clustering search algorithms into the Bunch tool:

1. **Hill-Climbing Search Algorithm.** Bunch's hill-climbing clustering algorithms [37] start by generating a random partition of the software structure graph G. Modules from this partition are then rearranged systematically by examining neighbouring partitions in an attempt to find an 'improved' partition (a partition with a better fitness). If a better partition is found, the process iterates, using the improved partition as the basis for finding even better partitions. The hill-climbing search algorithm eventually converges when no improved partitions of G can be found.

   During each iteration, several options are available for controlling the behaviour of the hill-climbing algorithm:

   (a) The neighbouring process uses the first partition that it discovers with a larger fitness function (*MQ*) as the basis for the next iteration.

   (b) The neighbouring process examines all neighbouring partitions during the current iteration and selects the partition with the largest *MQ* as the basis for the next iteration.

   (c) The neighbouring process ensures that it examines a minimum number of neighbouring 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 neighbouring process continues, and uses the next partition that it discovers with a larger *MQ* as the basis for the next iteration.

[2] It should also be noted that the general problem of graph partitioning (of which software clustering is a special case) is NP-hard.

2. **Hill-Climbing using Simulated Annealing.** Bunch's hill-climbing algorithm includes the ability to accept, with some probability, a partition with a worse fitness function value as the new solution of the current iteration. The probability of accepting a non-improving partition is reduced exponentially as the clustering process continues by invoking a user-defined *cooling function*. Results associated with the use of Bunch's simulated annealing feature have shown an improvement in performance without sacrificing any quality in the clustering results [40].

3. **Genetic Algorithm (GA).** The Bunch GA [19] uses traditional operators such as selection, crossover, and mutation to determine a 'good' partition of the software structure graph. Although GAs have been shown to produce good results in many search problems, the quality of the clustering results produced by Bunch's hill-climbing algorithm are typically better than the Bunch GA. Mitchell and Mancoridis think that further work on Bunch's encoding and crossover techniques is necessary.

   Harman et al. [24] implemented two GAs and a hill climber to cluster software systems using a fitness function that is also based on maximizing the cohesiveness of clusters and minimizing inter-cluster coupling. The fitness function used by Harman et al. also includes a concept of 'target granularity', the idea is that the tool should aim to produce clusterings with a desired number of modules. The fitness function achieves this by enforcing a quadratic punishment as the clustering produced deviates from the target granularity. The two GAs used explored cross-over techniques designed to promote the formation of good building blocks, by preserving at least one module in each child. Although this produced improved results over a GA with no such cross-over technique, Harman et al. found that hill climbing far out performed both variations of genetic algorithms, confirming results by the Bunch team [39].

   Lutz [36] has also worked on hierarchical clustering using search-based techniques. His work was concerned with the problem of decomposition of software into hierarchies at different levels of abstraction, like the work of Macoridis et al., but unlike that by Harman et al., which was concerned with only a single level of abstraction (the implementation level). Lutz also focuses upon the clustering and hierarchies in designs rather than code.

   The approach adopted by Lutz differs strongly from the approach adopted in the work of Macoridis et al. and that of Harman et al. with regard to the choice of fitness function. The fitness function used by Lutz is based upon an information-theoretic formulation inspired by Shannon [50]. 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.

Search-based software modularisation is now an established approach to software clustering, with promising results in terms of quality and efficiency. Mitchell's Ph.D. dissertation [39] reports promising results, both in terms of quality and performance, when applying search algorithms to the software clustering problem. For example, the Bunch hill-climbing algorithm produces acceptable clustering results for systems such as Sun's swing class library in about 30 seconds, and the linux operating system in approximately 90 seconds.

3.4 Cost Estimation

Initial approaches for applying metaheuristic methods to the problem of cost estimation have considered 'system identification' or 'symbolic regression' approach [17, 18]. This work has used Genetic Programming (GP) to learn predictive functions from training sets. The idea is similar to that studied in the works of McKay and Willis, in which GP is used to discover the function that relates the data points of the independent to the dependent variables. Dolado [17, 18] has shown that GP is able to discover nonlinear as well as linear systems. In the cost estimation problem the data relates the size of the application to the effort (usually measured in person-months). Size is usually measured in function points or in lines of code.

The operands of a solution may be constants or variables. The operators include the following:

$$+, -, *, /, power, sqrt, square, log, exp.$$

This variety of operators will allow approximation of almost any function likely to be required by this model. The population of equations to be evolved is composed of a set of well-formed equations to which the usual genetic operators mutation and crossover are applied. Crossover is applied to the trees by exchanging subtrees between equations. Mutation consists in changing the value of the nodes of the trees, by replacing a node with other operator or with a new numeric value.

The fitness function used in the evaluation of the equations is the *mean squared error*,

$$mse = \frac{1}{n-2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2.$$

This is a usual measure for assessing regression models. Other measures, such as the correlation coefficient, could not be considered since it is not clear which one should be recommended. The next generation is formed by selecting the equations according to the 'fitness proportionate selection method'. With this method an individual of the population is selected or rejected according to a probability proportional to its fitness.

Dolado [17] reports that the number of individuals in each generation were low, around 50 equations, and the number of generations that provided the best results was surprisingly very low, from 3 to 5 generations in most cases. These parameters have not demanded special computing power, as the convergence to the solution was achieved so quickly. This behaviour can be attributed to the few data points available in each data set, as well as the low number of variables.

Dolado [17, 18] also reports results regarding the application of GP to cost estimation. First of all, the GP obtains equations that approximate the data at least as well as the classical regression method. In some cases GP is able to obtain better cost functions than classical regression. However, the improvement in the predictions is not dramatic and GP can not overcome the intrinsic limitations of the data points. GP has helped to conclude that the best curve that approximates several data sets, publicly available, is a linear function between size and effort [18]. The results of classical regression can be misleading due to the fact that only a small set of the space of functions can be analysed.

The main benefit that GP provides is confidence in the results. GP explores the solutions guided only by the fitness of the equations and does not constrain the form of the solution. Therefore the final set of equations that are found by GP have the best predictive values. By running the algorithm many times, an ample set of good approximations is obtained.

Two methods for size estimation were compared in [17], the component-based method and the MarkII function point method. The results of GP were compared to those obtained by Artificial Neural Networks (ANN) and classical regression. It was observed that ANNs performed slightly better than GP, from the predictive point of view. However, the drawback of the ANNs in such prediction systems is that they do not provide a symbolic representation of the equation, rendering the method less useful for the software project manager. With the ANN approach it is hard to work out the reason behind the automated system's choice of predicted value, leading to uncertainty as to the confidence that can be placed in such an estimate.

Linear regression (or curve estimation) presented the worst values. GP essentially offers the best of both approaches since it provides symbolic equations as the result of its evolutionary process and, also, takes advantage of the computational power of techniques. Another advantage of the GP approach is that it is a non-parametric method since it does not make any assumption about the distribution of the data, deriving the equations according only to fitted values.

A similar approach to software project cost estimation is proposed by Burgess and Lefley [13], who use GP to evolve functions which provide good fits to the trends found in software project cost data. Burgess and Lefley present result comparing GP-based effort estimation against approaches based upon stepwise regression and neural nets. They also show how to highlight statistically significant predictions.

Aguilar et al. [1] have also worked on the problem of search-based software project cost estimation. In their work, the use of the evolutionary algorithms (EA) is directed towards the extraction of management rules for estimation and decision in software projects. The key elements of their approach are:

1. the construction of a dynamic model of the software development process
2. the simulation of the model with different ranges in the parameters for obtaining a database
3. the application of the evolutionary algorithms to the database generated for obtaining a set of management rules.

The database contains values obtained by the dynamic simulation of the software process model. The database can be as big as the manager wants, only bearing in mind that the simulation for every combination of ranges of parameters takes time and that the performance of the EA depends on the accuracy of the rules founded.

The representation (coding) of the search space is a set of individuals, each one taking the form:

$$[l_1, u_1, ., l_m, l_m, c]$$

where l_i, u_i are the values representing an interval for the attribute i and c is the class to which the tuple belongs for the classification. The algorithm is a sequential covering EA, and the fitness function tries to discriminate between correct and incorrect classifications of the examples. This is carried out by maximizing the function

$$f(i) = 2(N) - CE(i) + G(i) + coverage(i)$$

where N is the number of examples being processed, $CE(i)$ is the number of examples belonging to the rule whose class is different to that of the rule, $G(i)$ is the number of examples correctly classified by the rule and the *coverage* is the proportion of the search space covered by the rule.

The application of this EA to a numerical database classifies the data into a series of management rules. The benefit of this approach is that the task of finding the good decision rules within the database generated is left to the EA. Aguilar et al. [1] report that the EA performs better than the classical decision tree-based algorithms, like C4.5.

Kirsopp and Shepperd [32] use case based reasoning to estimate unknown project attributes, based upon partial information about the project. The approach is to locate similar projects (based upon the known data for the new project) and to use the data from these similar previous projects to estimate the unknown attributes of the new project. Kirsopp and Shepperd have many project attributes, some of which are better predictors than other.

Determining the best set of attributes to use as the basis for a prediction, is a feature subset selection problem, to which search-based approaches present reasonable solutions [63]. Interestingly, Kirsopp and Shepperd [32], (like Mancoridis and Mitchell [19, 39] and Harman et al. [24]), found that hill climbing out-performed more elaborate search techniques for this problem. These results, taken together, give rise to optimism that progress can be made in Search-Based Software Engineering, using relatively simple search-based techniques.

4 Software Engineering as a Search Problem: Pushing the boundaries

This section considers several aspects of software engineering for which metaheuristics have not been applied, but which, it will be shown, are promising candidates for the application of these techniques.

An important precursor to the application of metaheuristic algorithms to software engineering is the ability to characterise software engineering problems as search problems. The notable successes with test data generation are partly due to the fact that this problem has a large, natural search component and an easily definable fitness function. Such characteristics are not so readily identifiable in other areas of software engineering.

This section considers other problem areas in software engineering, arguing that it *is* possible to view these problems as search problems, in such a way that metaheuristic search techniques can be applied. In order to do so, it will be shown that there are natural definitions of the 'key ingredients' of metaheuristic search techniques identified in Section 2.

The rest of this section considers three areas of software engineering: requirements, systems integration and software maintenance and evolution. It will shown how each can be formulated as a search problem, defining appropriate fitness functions, mutation operations and representations for individuals and, where applicable, defining near neighbours and cross-over operations.

4.1 Requirements Phasing

The need to phase or schedule requirements typically occurs in projects which have a relatively short development time, a high level of user involvement, and often operate in a dynamic environment. However, the process is more widespread than this and occurs whenever there is a need for rapid and iterative product development.

Essentially the development of a system consists of an iterative cycle of "negotiate a bit, build a bit". This negotiation takes the form of selecting from a set of potential requirements, which are then implemented prior to the system being reviewed by the customers. A requirement is typically an aspect of system functionality and so this set is not going to be trivially small. The problem of interest is how to select the set of requirements that is going to constitute the next iteration of the system. This is not too problematic in those cases where there is a single customer. However, when there are a number of customers with differing needs it is important that all views are taken into account when this choice is made. It may well be impossible for all customers to be represented at a meeting and it is important that the system does not only reflect the views of those customers who happen to be the most vocal.

This was recognised by Boehm and his co-workers some time ago and is the motivation behind the development of the spiral model into the WinWin spiral model [12]. The original spiral model incorporated a phase which required the project manager to establish the objectives, constraints and alternatives for the next level of development. Determining the origin of these objectives, constraints and alternatives was found to be difficult and so three additional activities were added on to the front of the spiral model to give the WinWin spiral model. These three activities involved: identifying the system's key stakeholders (developers, customers, users etc.), identifying the stakeholders' win conditions (basically a desired outcome), and negotiating win-win (mutually agreeable) reconciliations of these win conditions. The negotiation process is supported by the WinWin groupware tool which visualises links between stakeholders' win conditions, issues, options and negotiated agreements, enables negotiations, and records changes, but does not attempt to resolve any of the issues automatically.

Karlsson and Ryan [31] use the Analytic Hierarchy Process to determine cost-benefit tradeoffs for a set of requirements. The relative value of requirements is determined by customers and users, and the relative cost is determined by the developers. These results are then combined and the system stakeholders use the resulting cost-value diagram as a vehicle for determining which requirements will be implemented. The selection of a subset of requirements from a number of alternatives can of course be viewed as a classic Multi Criteria Decision Making problem and one approach is to attempt to resolve the problem using the well-established techniques from this discipline (see [7] for example).

To see how metaheuristic approaches can address this problem, consider first of all the relatively straightforward case which ignores the criteria such as cost, functionality, and development time, that different customers might wish to see optimised in different ways (this more complicated case will be dealt with later), and assume that a customer's view is amalgamated into a single opinion. Assume also that all requirements are independent. At each iteration of the system build there is:

- A set of requirements, R, which have yet to be implemented:

$$R = \{r_1, ..., r_n\}$$

- A development cost associated with each requirement.
- A resource (a limit on how much the customers wish to spend).
- A set of customers who might have different opinions of what is required in the next phase of the system.

A choice has to be made of which set of requirements is going to be incorporated into the next phase of development. Each customer obviously wishes to select the set which is of greatest value to them. The problem is how to select the set of requirements that is going to be acceptable to as many customers as possible.

A customer's opinion is expressed as a priority or weighting, p_x, associated with each requirement. Suppose there are k customers and n potential requirements. A customer-priority matrix, CP can be defined, such that $p_i^j \in CP$ represents the priority associated with customer i for requirement j $(1 \leq i \leq k, 1 \leq j \leq n)$. The values used for each p_i^j will depend upon the nature of the application. If all users are equal, then each could simply rank the requirements, with the most desirable being ranked n, through to the least desirable ranked 1. Alternatively, the users could each be allocated a number of 'votes', with multiple votes per requirement permitted. This would allow users to record the relative magnitude of their desire to see a certain requirement implemented. It would also allow the developer to allocate a greater number of 'votes' to users considered more important.

Using either of these strategies it becomes possible to 'score' a choice of requirements according to overall user preferences. The association of an absolute value (using a voting mechanism or some method of quantifying this value in terms of business benefit) will be preferable as these can meaningfully be summed to give the total value of a requirement. Using such an 'absolute value' system, the value of the j^{th} requirement is defined as

$$tp_j = \sum_{i=1}^{k} p_i^j$$

Using a ranking system, the values associated with each requirement merely place the elements on an ordinal scale, making many forms of value combination meaningless according to measurement theory [52]. For ranked priority values a more complex way of combining individual user priorities will have to be found.

Having scored the requirements in some way, the problem is now to find the optimal set of requirements to build. However, this problem is an instance of the 0-1 knapsack problem [30], which is known to be NP-hard [21]. This makes the implied search problem appropriate for a metaheuristic-based algorithm. An individual solution to the problem can be represented by a bit vector, where each bit denotes the presence or absence of a requirement. Mutation can be represented by bit twiddling, crossover by simple selection of a crossover point, and neighbour proximity can be represented by hamming distance. This encoding would allow all the forms of metaheuristic search to be applied to the problem of choosing a set of requirements.

Once a solution is found an additional problem arises - how to explain this to the customer. The interpretation of solutions is nearly always a problem when applying metaheuristic techniques, but the importance varies according to the problem domain. For example, the area of test data generation discussed earlier, a customer (in this case, a person using the system to generate test data) is probably going to be content with the fact that data has been automatically generated to cover some percentage of program paths. They are unlikely to want to know *why* a particular value was generated, just pleased that the system has done some of their work for them. The attitude of the customer is likely to be different in the case of requirements phasing. They will probably want to know why a certain set was chosen and what happened to the requirements that they ranked highly. This matter is important, particularly while the customer lacks confidence in the system. One approach is to save more of the solutions and provide the customer with a list of the alternative solution combinations that just missed being selected. Another is to try and explain the thinking behind the fitness function. Whatever approach is taken it is important that all customers feel they have a chance to question and enhance the system for everyone's mutual benefit.

Unfortunately, the assumption that requirements are independent and that customer preferences can be expressed as a simple opinion is unrealistic. There will be dependencies between requirements that will impact upon the decision of which set of requirements to select. Requirements which depend upon others which are not already in the system are going to incur a greater cost. It is also unrealistic to assume that cost and requirement functionality are the only important factors. Customers are likely to be concerned about other constraints such as development time, and the developers themselves might have a strong desire to build the system in a way that smoothes out release dates. So associated with each requirement will be not one attribute, but a vector of attributes representing the cost, development time, associated dependencies, etc.

If the primary focus of each customer is still on the functionality then these additional attributes pose no real problem except as constraints that have to be included within the fitness function (in much the same way as cost is a constraint in the simple case). A difficulty arises when different groups of customers decide priorities according to different criteria. For example, one set of customers might prioritise requirements on the basis of functionality, another group might focus on development time on the basis that they want some sort of system as soon as possible, whilst a third group might want the cheapest working system imaginable and hence prioritise according to cost. This forms a classic Multi Criteria Decision Making problem. One approach is to try and adapt the fitness function to accommodate this, but the danger is that in trying to find a solution the main focus of each of the distinct customer groups becomes lost. Another approach is to tackle it in two phases, firstly by considering each group independently and extracting a subset of requirements which meets their criteria using fitness functions which reflect the priorities, and then combining these three subsets in a second pass of the algorithm which uses a more balanced fitness function (or one which reflects the relative group sizes).

These additional constraints also change the nature of the problem – it is no longer a matter of selecting from an ordered list of requirements. The choice of which requirement to incorporate has to take the vector of costs, times, dependencies etc. into account. Constraints on requirements make it possible to generate 'invalid individuals' – those for which the set of requirements denoted are not closed under the dependence relation, for example. When this happens, there are two possible strategies available:

1. ‘Repair’ the individual [41]. That is, include in the requirement set r the transitively dependent requirements of those in r. This has the advantage that it reduces the size of the search space to only those individual that represent valid solutions (those containing all relevant dependent requirements). However, it may bias the search towards inclusion of requirement upon which many others depend.

2. ‘Punish’ the individual with a very low fitness score. That is, for individuals which do not contain dependent requirements, reduce the fitness score to make these less likely to be selected. This has the disadvantage that the search space is larger than that for a ‘repair-based’ solution, but does not have the potential bias that repair might introduce.

Of course, it will also be possible to generate individuals which require a cost greater than that allowed by the customer. These can be allocated a very low fitness score, making it unlikely (though not impossible) for them to be selected. The coefficient used to determine the unfitness level for cost-breaking solutions, can be adjusted to take account of the ‘softness’ of the cost ceiling. This would allow for solutions to be selected which were near the cost boundary and noticeably more attractive in terms of customer requirement.

This flexibility of the metaheuristic search approach is one of the benefits of the methods which the authors believe makes it ideally suited to many software engineering problems.

4.2 Systems Integration

Systems integration consists of taking individual components and combining them, usually in an incremental fashion, to eventually create the entire system. The importance of adopting an incremental approach, and the probability that not all components are going to be available at the same time, means that many stubs and drivers (dummy components plugged in place of the real component to simulate the interaction) have to be written. The testing that takes place at this stage is important as it is often the first opportunity to determine if two (or more) components interact correctly with each other. Any component which fails at this stage is likely to be sent back to the developers for re-work. This re-work introduces two problems - firstly a delay in the systems integration process (the more important the component, the greater the delay), and secondly the need to re-test when the corrected component is re-integrated. All this leads, in turn, to the systems integration process becoming very chaotic, when ideally it ought to operate like a well-organised assembly plant.

Viewing the integration problem as a search problem provides an opportunity to achieve this assembly plant type of process. Each component in a system may make use of, and/or be used by, some other component. If a heavily used component is integrated late in the process (having been stubbed prior to its introduction) and then reveals several faults in the components with which it interacts, these interacting components may have to be re-worked, and then re-integrated, resulting in significant time delays and much repeated testing. On the other hand, if a component which uses many other components is integrated early, then the components it uses will have to be replaced by stubs. This can be wasteful in terms of resources and causes problems as interaction faults are only visible when the real components are introduced.

This leads to the situation where, ideally, the heavily used components are integrated first and the heavily dependent components are integrated last. Obviously some components are both used and dependent and so a cost is associated with each use or dependency. This cost, along with information about which components have been integrated so far, can then be used to determine the ideal order of integration. This in turn can be fed back to the developers to enable the production of components in the order which is most likely to lead to the cheapest (least overall time due to fewest re-works, minimal stubbing etc.) system integration.

An order of integration of the components can be represented by a permutation of the set of components. Since the individual is a permutation, it is not sensible for crossover to proceed by simply taking parts of two individuals and conjoining them: the result of this is unlikely to be a valid individual. Instead, it is necessary to apply a crossover technique such as order crossover or cycle crossover. These are guaranteed to return a valid permutation. Under order crossover, a random cut point is chosen and the left substring of the first individual is kept. The individual is completed by taking the elements left out in the order they occurred in the second parent. In contrast, cycle crossover merges the two permutations (see, for example, [47] for more details).

The mutation operators available to this representation of the systems integration problem are essentially the swap and shuffle operators:

`swap` This operator interchanges two genes.

`shuffle` This operator produces a gene permutation.

For the application to integration ordering the effect of a swap has a relatively low impact upon the overall fitness. This is because the cost function depends upon whether one activity comes before or after another and swap only changes two such values. For a large sequence of components to be integrated this will have a smaller effect than a similar swap on a transformation sequence. As will be seen in the following section, transformation sequences are more akin to simple programs from a 'linear programming language', where a single change to a 'programming language statement' can have a dramatic effect on the meaning of the overall 'program'.

For the systems integration question, there is also a clear notion of a neighbourhood. That is, two permutations p and p' are neighbours if and only if p can be obtained from p' by swapping two adjacent event genes. The problem now is to find an appropriate objective function to drive the optimisation process. Naturally, given an order for the events, this objective function should just provide the estimated cost associated with this order. This may be determined by, for each ordered pair (c_1, c_2) of components, determining the costs associated with integrating c_1 before c_2. The total cost, of an order of integration, may then be determined.

As has been demonstrated, it is possible to define all the key ingredients for metaheuristic algorithms for the application to systems integration problems. It is therefore possible to claim that all three metaheuristic techniques considered in this paper can be applied to this software engineering problem.

4.3 Maintenance and Re-engineering using Program Transformation

Program transformation has a long history dating back to the work of Darlington and Burstall on pure functional languages [15]. Using program transformation, a sequence of simple transformation steps is applied to a program to translate it from one form into another. All approaches to transformation share the common principle that they alter the program's syntax without affecting its semantics.

Transformation is familiar from its use in compiler technology, where, for some time, it has been used to support optimisation [2]. However, transformation is also applied at higher levels of abstraction, where the goal is to restructure the program in some way [5], to partially evaluate it [11, 14], or to support activities typically associated with the later phases of the software development life cycle, such as maintenance [8, 25, 59] and re-engineering [9, 34]. There are many systems which support transformation for these activities, sich as FermaT [58] and KIDS [53].

For these problems the number of transformation steps may be very large (the maintainers' assistant [59] for example, supports several hundred transformations). At each step many of these will be applicable to the program under transformation. Effective program transformation strategies are hard to define [10], because the space of solutions is exponentially large in the number of basic transformation steps and the typical transformation sequence length.

Transformation for these applications can thus be viewed as a search problem in the space of admissible transformation sequences, to which genetic algorithms, simulated annealing and tabu search can be applied. Such techniques have been shown to be effective for the specific transformation-based problem of automatic parallelisation [45, 46, 66, 67], so there is reason to hope that they will be applicable to more general software engineering problems.

Fitness functions on programs are nothing more than code-level software metrics [20, 52]. A metaheuristic approach to program transformation therefore provides a mechanism to combine program transformation rules with software metrics. The metric is used to guide the search, while the possible solutions that can be found are bounded by the set of transformations chosen.

The existing work on transformation using evolutionary techniques has focussed upon automated parallelization. Three approaches have been considered. The rest of this section considers each of these and how it might be built upon to allow evolutionary techniques to be applied to software maintenance and re-engineering through transformation.

4.3.1 Program as Individual: Local Search Approach

With the program to be transformed as the individual, the search space consists of all semantically equivalent programs which can be reached by a sequence of transformation steps from the original. The representation of the program itself (as an Abstract Syntax Tree or Control Flow Graph) is unimportant, so long as the transformation steps can be applied to the program to yield a new individual.

In the 'Program as Individual' representation, the mutation operators are the transformation steps. This approach is adopted by Williams [65]. He compares the approach to one which uses the transformation sequence as an individual (described in Section 4.3.3) and reports that the 'program as individual' approach yields superior results.

This approach is essentially a local search using hill-climbing; as the program is mutated (using transformation) and better programs are retained. Williams' success with the approach indicates that the landscape for program transformation (at least in the case of auto-parallelisation) may be smooth enough for hill climbing to produce reasonable results.

It is not clear whether these superior results reflected the type of program Williams was considering, nor whether they were achieved because the domain under consideration was auto-parallelisation, rather than more general re-engineering transformation. More work will be required to see if these results are replicated when evolutionary transformation is applied to re-engineering.

4.3.2 Program as Individual: Traditional Genetic Programming Approach

Using 'Program as Individual' it is hard to define crossover, because two programs p_1 and p_2, with identical behaviour would have to be combined to produce a program p' with identical behaviour to each of p_1 and p_2. Such a combination is far from obvious in most cases. However, there is hope that advances in genetic programming [33] may be of assistance here.

Ryan [45] describes an approach, called Pargen I, for auto-parallelisation, in which the individual is a program to be transformed, represented as an Abstract Syntax Tree, to which genetic programming mutation and crossover operators are applied. This approach can inherently lead to a program which is not faithful to the semantics of the original. In the language of program transformation, this would result in a non-meaning preserving transformation.

In an attempt to overcome this possibility, Ryan uses a set of test cases which he incorporates into the fitness function. The evolutionary algorithm therefore has a multi-objective fitness function. The fitness function rewards solutions for both correctness (according to the test cases) and efficiency. Of course, the problem here is that the algorithm may render programs which are fast but incorrect. Worse, even where the algorithm yields answers which (according to the fitness function) are 'correct' they are only correct with respect to the test cases and so correctness is no longer guaranteed. For re-engineering applications, correctness is vital, and so the Pargen I approach is unlikely to form a good basis for further development.

4.3.3 Transformation Sequence as Individual: Genetic Algorithm Approach

Taking a step up from the program as the individual, the sequence of transformations applied to the program could be considered to be the individual. In this approach, the fitness function involves applying the sequence of transformations to the original program to produce a transformed program p' to which the metric to be optimised is applied. This approach is adopted by Ryan for auto-parallelisation using the Pargen II system.

For more general re-engineering transformation, using this approach, a number of mutation operations are possible: `replacement`, `shift left/right`, `rotate left/right` and `swap` are four obvious choices:

`replacement` This operator takes a sequence of transformations and replaces one or more transformation steps in the sequence to produce a mutated sequence;

`shift left/right` This operator takes a sequence of transformations, removes one transformation step, and shifts the remaining transformation steps to the left or right, adding a new transformation step at the beginning or end of the sequence;

`rotate left/right` This operator does not replace any transformation steps, but moves all to the left or right, rotating the last/first transformation step to the first/last position respectively;

`swap` This operator interchanges two transformation steps in a transformation sequence.

Of these candidates, `swap` and `rotate` do not introduce any new transformation steps. In most situations these two mutation operators will therefore be insufficient on their own, as new information will need to be introduced to ensure that there do not exist some areas of the search space which can never be explored.

Replacement is likely to be effective in the early stages of a search, where the algorithm will need to be free to consider widely different individuals. The shift and rotate operators will tend to be less disruptive to the effect of the transformation sequence and may be more suited to later stages of the search.

Using the transformation sequence as the individual makes it possible to define crossover relatively easily. Two sequences of transformations can be combined to change information, using single point, multiple point and uniform crossover. In fit individuals, subsequences of transformations are likely to represent simple transformation tactics, which individually make some progress towards the goal embodied in the fitness function. It would be sensible to choose an approach to crossover which did not unnecessarily disrupt these transformation tacticals and so uniform crossover is unlikely to be successful.

Changing a single transformation step can radically alter the way in which a sequence of transformations behaves. This makes it hard to define a suitable concept of near neighbour for this representation. For the 'transformation sequence as individual' approach to the representation problem, it is thus possible to define fitness, mutation and crossover operators but hard to define the concept of near-neighbour. Such a representation therefore suggests a genetic algorithm as a means of tackling the search problem.

4.3.4 Transformation Program as Individual: Adapted Genetic Programming Approach

An alternative approach, based upon GP, but not upon a test-based fitness function, would be to construct a transformation *program* [64] as the individual and to measure its fitness against a suite of 'training' programs. The fitness would be based upon the improvement an individual transformation program was able to produce across this training suite.

The standard syntax-tree based operators of genetic programming can be applied to mutate transformation programs and to crossover two transformation programs. The fitness function would reward a transformation program which performed well across the range of programs in the training suite. The primitive operations of the transformation language would be the simple transformation steps, while the constructs of the language would be sequencing, selection and iteration. The transformation language would also have to include predicates over program structures.

This may allow higher-level program transformation tactics to emerge. It would also be a way of examining both the training suite and the set of transformation primitives available. The behaviours with different training suites would give insights into the similarities the programs in the suite present for transformation. The behaviours with different sets of transformation primitives would yield insight into the effectiveness and applicability of the chosen sets of transformations for the particular task in hand.

The goal here would not be to produce a *perfect* transformation algorithm for transforming the suite of programs (or programs which resemble members of the suite). This would be a very quixotic aim. Rather, the aim would be to gain insight into the transformation tactic applicable, the sets of transformations which are useful for a given metric and training suite and the kinds of program which are amenable to a chosen transformation goal.

5 Conclusion

Metaheuristic Algorithms have found wide application in engineering problems. The problems for which these techniques have been most suitable have consisted of situations where complex, competing and inter-related constraints are placed upon the development process and its products. In such a situation, the implicit tradeoffs between criteria form points in a fitness landscape, defined by a cost function which assesses the fitness of a particular configuration of constraints.

Software engineering provides many examples of situations where competing constraints have to be balanced against one-another. It therefore appears to be an attractive and ripe area for the application of metaheuristic search techniques. Such techniques are applicable when good solutions are easy to recognise but hard to generate, because of the competing constraints, the size of the search space and the complexity of the cost function.

These techniques have already been applied very successfully to the problem of finding test data to achieve test-data adequacy criteria, to module clustering and to cost estimation. This paper has considered several other areas of software engineering for which metaheuristic search will be likely to find successful application. These areas span the range of the software development process, from initial planning and requirements analysis, through to maintenance and the evolution of legacy systems.

References

[1] Aguilar-Ruiz, J., Ramos, I., Riquelme, J. C., and Toro, M. An evolutionary approach to estimating software development projects. *Information and Software Technology 43*, 14 (Dec. 2001), 875–882.

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

[3] Anquetil, N. A comparison of graphs of concept for reverse engineering. In *Proc. Intl. Workshop on Program Comprehension* (June 2000).

[4] Anquetil, N., and Lethbridge, T. Recovering software architecture from the names of source files. In *Proc. Working Conf. on Reverse Engineering* (Oct. 1999).

[5] Ashcroft, E. A., and Manna, Z. The translation of `goto` programs into `while` programs. In *Proceedings of IFIP Congress 71* (1972), C. V. Freiman, J. E. Griffith, and J. L. Rosenfeld, Eds., vol. 1, North-Holland, pp. 250–255.

[6] Bäck, T. *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, 1996.

[7] Belton, V., and Stewart, T. J. *Multiple Criteria Decision Analysis: An Integrated Approach.* Kluwer Academic Publishers, 2001.

[8] Bennett, K. Automated support for software maintenance. *Journal Information and Software Technology 33*, 1 (1991), 74–85.

[9] 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.

[10] Bennett, K. H. Do program transformations help reverse engineering? In *IEEE International Conference on Software Maintenance (ICSM'98)* (Bethesda, Maryland, USA, Nov. 1998), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 247–254.

[11] Bjørner, D., Ershov, A. P., and Jones, N. D. *Partial evaluation and mixed computation.* North–Holland, 1987.

[12] Boehm, B., Egyed, A., Kwan, J., Port, D., Shah, A., and Madachy, R. Using the winwin spiral model: A case study. *IEEE Computer 31*, 7 ("July" 1998), 33–44.

[13] Burgess, C. J., and Lefley, M. Can genetic programming improve software effort estimation? A comparative evaluation. *Information and Software Technology 43*, 14 (Dec. 2001), 863–873.

[14] Consel, C., Hornof, L., Marlet, R., Muller, G., Thibault, S., and Volanschi, E.-N. Partial evaluation for software engineering. *ACM Computing Surveys 30*, 3es (Sept. 1998). Article 20.

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

[16] De Jong, K. On using genetic algorithms to search program spaces. In *Genetic Algorithms and their Applications: Proceedings of the second international conference on Genetic Algorithms* (MIT, Cambridge, MA, USA, 28-31 July 1987), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, pp. 210–216.

[17] Dolado, J. J. A validation of the component-based method for software size estimation. *IEEE Transactions on Software Engineering 26*, 10 (2000), 1006–1021.

[18] Dolado, J. J. On the problem of the software cost function. *Information and Software Technology 43* (2001), 61–72.

[19] 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).

[20] Fenton, N. E. *Software Metrics: A Rigorous Approach.* Chapman and Hall, 1990.

[21] Garey, M. R., and Johnson, D. S. *Computers and Intractability: A guide to the theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[22] Glover, F. Tabu search: A tutorial. *Interfaces 20* (1990), 74–94.

[23] Goldberg, D. E. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[24] Harman, M., Hierons, R., and Proctor, M. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference* (New York, 9-13 July 2002), Morgan Kaufmann Publishers, pp. 1351–1358.

[25] Harman, M., Hu, L., Hierons, R. M., Zhang, X., Munro, M., Dolado, J. J., Otero, M. C., and Wegener, J. A post-placement side-effect removal algorithm. In *IEEE International Conference on Software Maintenance (ICSM 2002)* (Montreal, Canada, Oct. 2002), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 2–11.

[26] Holland, J. H. *Adaption in Natural and Artificial Systems.* MIT Press, Ann Arbor, 1975.

[27] Hutchens, D., and Basili, V. System structure analysis: clustering with data bindings. *IEEE Transactions on Software Engineering SE-11*, 8 (1985), 749–757.

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

[29] Jones, B. F., Eyres, D. E., and Sthamer, H. H. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal 41*, 2 (1998), 98–107.

[30] Jung, H.-W. Optimizing value and cost in requirements analysis. *IEEE Software 15* (1998), 74–78.

[31] Karlsson, J., and Ryan, K. A cost-value approach for prioritizing requirements. *IEEE Software 14*, 5 (September/October 1997), 67–74.

[32] Kirsopp, C., Shepperd, M., and Hart, J. Search heuristics, case-based reasoning and software project effort prediction. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference* (New York, 9-13 July 2002), Morgan Kaufmann Publishers, pp. 1367–1374.

[33] Koza, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.

[34] Lakhotia, A., and Deprez, J.-C. Restructuring programs by tucking statements into functions. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier, 1998, pp. 677–689.

[35] 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.

[36] Lutz, R. Evolving Good Hierarchial Decompositions of Complex Systems. *Journal of Systems Architecture 47* (2001), 613–634.

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

[38] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E. Equation of state calculations by fast computing machines. *Journal of Chemical Physics 21* (1953), 1087–1092.

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

[40] Mitchell, B. S., and Mancoridis, S. Using heuristic search techniques to extract design abstractions from source code. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference* (New York, 9-13 July 2002), Morgan Kaufmann Publishers, pp. 1375–1382.

[41] Mitchell, M. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[42] Müller, H., Orgun, M., Tilley, S., and Uhl, J. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice 5* (1993), 181–204.

[43] 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.

[44] Reeves, C. R., Ed. *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Scientific Press, Oxford, UK., 1992.

[45] Ryan, C. *Automatic re-engineering of software using genetic programming.* Kluwer Academic Publishers, 2000.

[46] 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.

[47] Sait, S. M., and Youssef, H. *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems.* IEEE Computer Society, 1999.

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

[49] The Drexel University Software Engineering Research Group (SERG). http://serg.mcs.drexel.edu.

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

[51] Shaw, M., and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[52] Shepperd, M. J. *Foundations of software measurement.* Prentice Hall, 1995.

[53] Smith, D. R. KIDS: a semiautomatic program development system. *IEEE Transactions on Software Engineering 16*, 9 (Sept. 1990), 1024–1043.

[54] Tracey, N., Clark, J., and Mander, K. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis* (March 1998), ACM/SIGSOFT, pp. 73–81.

[55] 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.

[56] 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.

[57] van Laarhoven, P. J. M., and Aarts, E. H. L. *Simulated Annealing: Theory and Practice.* Kluwer Academic Publishers, Dordrecht, the Netherlands, 1987.

[58] Ward, M. Assembler to C migration using the FermaT transformation system. In *IEEE International Conference on Software Maintenance (ICSM'99)* (Oxford, UK, Aug. 1999), IEEE Computer Society Press, Los Alamitos, California, USA.

[59] 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.

[60] 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.

[61] Wegener, J., Grimm, K., Grochtmann, M., Sthamer, H., and Jones, B. F. Systematic testing of real-time systems. In *4th International Conference on Software Testing Analysis and Review (EuroSTAR 96)* (1996).

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

[63] Whitley, D., Beveridge, J. R., Guerra-Salcedo, C., and Graves, C. Messy genetic algorithms for subset feature selection. In *Proceedings of the 7th International Conference on Genetic Algorithms* (San Francisco, July 19–23 1997), T. Bäck, Ed., Morgan Kaufmann, pp. 568–575.

[64] Wile, D. Program developments: formal explanations of implementations. *Communications of the ACM 26*, 11 (Nov. 1983), 902–911.

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

[66] Williams, K. P., and Williams, S. A. Genetic compilers: A new technique for automatic parallelisation. In 2^{nd} *European School of Parallel Programming Environments (ESPPE'96)* (L'Alpe d'Hoez, France, 1996), pp. 27–30.

[67] Williams, S. A., and Fagg, G. E. Experience of developing codes for distributed and parallel architectures. In *PPECC Workshop: 'Distributed v Parallel: Convergance or Divergence?'* (Mar. 1995), Parallel & Distributed Group DRAL, pp. 115–118.

[68] Winter, G., Periaux, J., Galan, M., and Cuesta, P. *Genetic Algorithms in Engineering and Computer Science.* Wiley, 1995.