A Search Based Automated Test-Data Generation Framework for Safety-Critical Systems

Nigel Tracey John Clark John McDermid

Department of Computer Science
University of York
Heslington, York.
YO10 5DD, UK
{njt, jac, jam}@cs.york.ac.uk

Keith Mander

Computing Laboratory
University of Kent
Canterbury, Kent.
CT2 7NF, UK
K.C.Mander@ukc.ac.uk

Abstract

This paper presents the results of a three year research program to develop an automated test-data generation framework to support the testing of safety-critical software systems. The generality of the framework comes from the exploitation of domain independent search techniques, allowing new test criteria to be addressed by constructing functions that quantify the suitability of test-data against the test-criteria. The paper presents four applications of the framework — specification falsification testing, structural testing, exception condition testing and worst-case execution time testing. The results of three industrial scale case-studies are also presented to show that the framework offers useful support in the development safety-critical software systems.

1 Introduction

Software testing is an expensive process, typically consuming at least 50% of the total costs involved in developing software [4], while adding nothing to the functionality of the product. It remains, however, the primary method through which confidence in software is achieved. Automation of the testing process is desirable both to reduce development costs and also to improve the quality of (or at least confidence in) the software under test (SUT). While automation of the testing process – the maintenance and execution of tests – is taking hold commercially, the automation of test-data generation has yet to find its way out of academia. Ould has suggested that it is this automation of test-data generation which is vital to advance the state-of-the-art in software testing [31].

This paper presents the results of a three year research project focused on test-data generation for safety-critical software systems. The specific focus of the project was to investigate technique to validate aircraft full authority digital engine controller (FADEC) software. Due to different requirements from different aircrafts and airlines FADEC software undergoes a large amount of change. The costs of reverification associated with this change is extremely large. The goal of this project was to develop automated testing support to reduce this cost, while maintaining or improving quality. The aim was to develop techniques that allowed test-data to be generated automatically for many different testing criteria, providing useful automated support for *real* industrial FADEC testing problems. The techniques developed in this project address testing problems that are applicable to other safety-critical software systems as demonstrated by the nuclear primary protection case-study presented later in this paper.

The framework is based on the application of directed search techniques and, in particular, genetic algorithms[1]. A test criterion can be addressed within the framework by providing a fitness function that provides a quantitive measure of test-data suitability for that test criterion. This paper presents several applications of the framework.

Section 2 outlines why safety-critical software needs to be tested and also introduces the properties of interest during testing. Section 3 provides a brief introduction to genetic algorithm to provide enough context for this work. Section 4 introduces the test-data generation framework that has been developed during this research project and describes the supporting prototype tool-set. Section 5 presents a number of application of the framework along with evaluations carried out using real industrial examples. Section 6 draws some conclusions and presents a summary of the related test-data generation work from the literature.

2 Safety-Critical Systems

A safety-critical system is a system whose failure can lead to injury or loss of life[2]. Safety-critical software is any software that can directly or indirectly contribute to the occurrence of a hazardous state in a safety-critical system[3]. A hazardous state is a condition of a system that, together with other conditions in the environment will inevitably result in an accident [25]. The goal in developing safety-critical systems and software is to ensure that the risk of accidents is acceptable.

2.1 Properties of interest

Verification activities traditionally focus on the functional correctness of software. However, for safety critical systems this is insufficient. Not all software errors cause hazards and not all software that functions according to its specification is safe [25]. Therefore, the goal of verification for safety-critical systems is to verify not only functional correctness, but also additional properties concerned with the safety of the software.

2.1.1 Functional properties

Functional correctness verification is concerned with showing the consistency between the implementation and its functional specification. Most verification techniques focus on this aspect and indeed the majority of industrial verification effort is devoted to functional correctness. Depending on the required integrity, evidence of functional

[1]The framework has also been used to evaluate the performance of different search techniques, genetic algorithms, simulated annealing, steepest descent.

[2]Sometimes the definition of safety-critical is extended to include systems where a failure can lead to loss or damage to property, the environment and economy.

[3]This definition is often divided such that a system that directly contributes to a hazard state is termed safety-critical. Software that can only indirectly contribute to a hazard start is termed safety-related. For the purposes of this paper, this distinction is unimportant as both classes of systems require testing.

correctness ranges from formal proofs of correctness to achieving measures of code coverage during functional testing.

The goal of achieving a specified measure of white-box structural coverage during unit testing is the most prevalent criterion used by industry for the verification of functional properties. Typically, more than 60% of the testing effort is devoted to this type of testing.

2.1.2 Real-time properties

A large proportion of safety-critical systems are required to interact with their environment — monitoring the environment through the use of sensors and controlling hardware in response through the use of actuators. In many cases, the correctness of this interaction relies not only on functional correctness, but also temporal correctness. In such a system, a result that is late may have the potential to cause hazardous states in the system. These types of systems are known as hard real-time systems. While not all hard real-time systems are safety-critical, most safety-critical control systems are hard real-time systems.

A fundamental property of hard real-time systems is the Worst-Case Execution Time (WCET) of the software. The WCET provides a bound on the maximum amount of computing resource required by a given piece of software. Evaluating this property is a prerequisite of higher-level real-time analysis techniques such as schedulability analysis [6].

A typical industrial approach to determine WCET information is to re-use the functional test-data. In practice, this is little better than random testing. Test-data chosen to exercise different aspects of functionality will not necessarily be best suited to exercising different aspects of timing behaviour. Additionally, re-using the functional test-data prevents early determination of WCET as the functional tests are often only available late in the development life-cycle. This exposes the project to great risk. Potential timing problems are discovered late in the life-cycle once a software design has been committed to. Consequently the costs of fixes are extremely expensive.

2.1.3 Safety constraints

As already discussed, verification of the functional correctness of software is not sufficient for safety-critical systems. Verification must also address the possibility of the software contributing to hazardous system states. By tracing hazardous system states into the software during the design, safety-invariants on the software can be established. One technique for achieving this is Software Fault Tree Analysis introduced by Leveson and Harvey [26].

2.1.4 Exception conditions

In languages such as Ada [16], there are pre-defined exceptions associated with run-time violation of the language rules (for example numeric overflow, divide-by-zero and out-of-bounds errors for types and arrays). The compiler is responsible for inserting special run-time checks into the generated object-code to detect these conditions. However, in safety-critical systems, these run-time checks are often disabled to reduce the complexity of the generated object-code, but it is only safe to disable these run-time checks if it can be shown that they will never fail, i.e. the run-time language rules will not be violated. Indeed, even for languages that do not define such run-time rules, such as C and C++, the control algorithms may only be safe in the absence of such errors.

A typical industrial approach to this problem is to provide *special* mathematical operators that are well-defined under at least some of these error conditions, for example a saturating divide routine and a truncating addition routine. The control algorithms are then designed to be safe in the presence of these extended operators. However, this only addresses a subset of the potential run-time errors — divide-by-zero and numeric underflow and overflow. Functional testing is typically relied upon to provide confidence in the absence of out-of-bounds errors for variables and arrays.

2.1.5 Meeting assumptions

When developing software components, assumptions are often implicitly made regarding the calling environment. It is important that these assumptions are met by the system as the behaviour of software may not be well defined outside these assumptions. For example, this can cause particular problems when re-using software. There are several documented cases of software re-use leading to accidents because the assumptions made during the initial development were not satisfied at the point of re-use [25] — re-use of Ariane 4 software on Ariane 5, UK re-use of US air traffic control software and US F-16 software re-use on an Israeli aircraft.

2.2 Why test safety-critical systems?

There are two types of analysis that can be used to verify properties of interest — static and dynamic. Static analysis involves the construction and analysis of an abstract mathematical model of the system. The analysis is performed without executing the *software under test* (SUT), but rather on this abstract model. Analysis of the abstract model enables general conclusions to be made about properties of the SUT and consequently can be said to be more complete than dynamic analysis. Naturally, the question arises ‘if static analysis is more complete and enables general conclusions to be drawn regarding the properties of interest, what is the need for dynamic analysis?’. The first point to note is that static analysis is not analysing the *real* SUT, but rather an abstract model of the SUT. Hence, there is a possibility that the construction of the abstract model hides an undesirable property from the static analysis. In contrast, dynamic analysis (testing) involves executing the actual SUT and monitoring its behaviour. While it is not possible to draw general conclusions from dynamic analysis, it does provide evidence of the successful operation of the software. It also provides evidence of the correct functioning of the compiler and hardware platform. These elements are rarely modelled for static analysis and consequently any errors they introduce go undetected during static analysis. There is also evidence that many problems arise because the requirements are wrong. Testing offers a powerful weapon against requirements errors as it allows the dynamic behaviour of the system to be explored and presented to the end-user in order to validate the requirements.

A survey of the literature (see section 6.1) shows there to be many techniques for automating the generation of test-data. However, these approaches to test-data generation focus on generating test-data in narrow areas. The majority focus on functional properties with most generating test-data for structural testing criteria. As discussed, testing is required to examine many properties, consequently the approaches limited to single testing strategies with no clear path for generalisation offer limited support to the industrial test engineer. In contrast, this project has developed a general extensible framework for test-data generation. This framework offers a clear path for extension to support the testing of new properties of interest. The approach taken exploits the restrictions placed on the design and implementation of safety-critical systems to make this possible:

- Small modules — a safety-critical software system is typically decomposed in a top-down manner into a collection of subsystems. Each subsystem is then further decomposed into a collection of units. The search-based approach to test-data generation taken in this project involves repeated execution of the SUT. Consequently, it is important that the execution time of the SUT does not make it intractable to execute the SUT repeatedly. During the evaluation of this project, the test-data generation framework has been applied both to units and subsystems, however the maximum execution time of the SUT has never exceeded a few tens of milliseconds.

- Restricted data types — safety-critical software is typically restricted to using scalar data-types and composite types built from scalar data-types (such as arrays and records). The quantitive evaluation of test-data performed by the framework in its search for desirable test-data exploits this restriction. It would therefore be difficult to develop effective support for complex dynamic data types.

- Simple design structure — the dynamic search-based nature of the test-data generation framework is only tractable if restricted searches are performed. It would be too computationally expensive to conduct an exhaustive search for test-data as the SUT may typically have an extremely large input domain. Consequently, the complexity of the search problem will affect the ability of the search to locate test-data. The simple design structures used in safety-critical software are imposed to constrain the complexity of software testing and analysis. This in turn limits the complexity of the search problem to allow the test-data generation framework to generate the desired test-data.

3 Genetic Algorithms

Heuristic global optimisation techniques are designed to find good approximations to the optimal solution in large complex search spaces. General purpose optimisation techniques make very few assumptions about the underlying problem they are attempting to solve. It is this property that allows a general test-data generation framework to be developed for solving a number of testing problems. Optimisation techniques are simply directed search methods that aim to find optimal values of a particular fitness function (also known as a cost or objective function). Within the framework, random search, hill-climbing search, simulated annealing [22] and genetic algorithms [13] have been implemented. This paper focuses on the use of genetic algorithms for test-data generation.

Genetic algorithms were developed initially by Holland *et al.* in the 1960s and 1970s [13]. They attempt to model the natural genetic evolutionary process. Selective breeding is used to obtain new sample solutions that have characteristics inherited from each parent solution. Mutation introduces new characteristics into the solutions.

Genetic algorithms work by maintaining a population of sample solutions each of whose *fitness*[4] has been calculated. Successive populations (known as *generations*) are evolved using the genetic operations of *crossover* (selective breeding) and *mutation*. The aim is that through the use of the genetic operations the population will converge towards a global solution.

From an initial population of randomly generated solutions, the fitness of each solution is calculated (details of how the fitness functions are constructed are presented in section 5). Using this information, members of the population are selected to become parents.

Once the parents have been selected they need to be combined to form the offspring. This is achieved using the *crossover operator*. The aim of crossover is to produce offspring that combine the best features from both parents to result in a fitter offspring. A number of the offspring are then *mutated* to introduce diversity into the population.

A new generation is then selected from the offspring and old population. An outline of the genetic algorithm search process is shown in figure 1. More detailed information on using genetic algorithms to address search problems can be found in [34].

```
procedure Genetic_Algorithm is
begin
 INITIALISE (Current_Population);
 CALC_FITNESS (Current_Population);
 loop
    SELECT_PROPECTIVE_PARENTS;
    CROSSOVER (Parents, Offspring)
    MUTATE (Offspring)
    CALC_FITNESS (Offspring);
    SELECT_NEW_POPULATION
      (Parents, Offspring);

    exit when STOP_CRITERION;
 end loop;
end Genetic_Algorithm;
```

Figure 1: Genetic Algorithm

4 Framework for Test-Data Generation

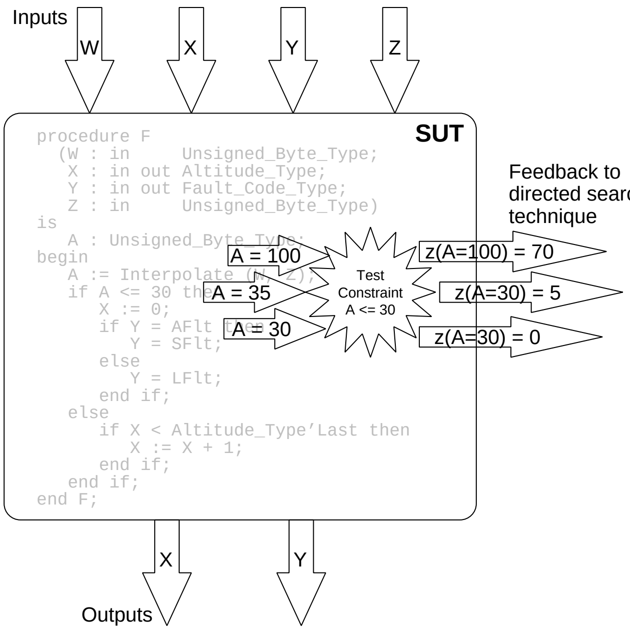
4.1 Introduction

Test-data selection, and consequently generation, is all about locating test-data for a particular test criterion. For many software properties, there is the concept of *good* test-data (test-data that is better for the specified purpose than some other test-data) hence it is possible to provide some measure of the suitability of given test-data against the test criterion. Typically, it is the software tester's responsibility to find the best test-data for the given testing criterion within particular constraints (often time or financial). However, locating suitable test-data can be time-consuming, difficult and hence expensive.

Often, test-data required by a criterion can be expressed as a constraint that must be

[4]Fitness provides a quantitive measure of a solution's suitability for the problem at hand. This is calculated using an fitness function that maps solutions to fitness values

satisfied at a particular point in the SUT's execution[5]. The software tester is required to develop test-data to satisfy the test constraint (or constraints). The test constraint is derived from the specific test criterion. Hence, it might represent the condition to cause the execution of a specific branch in the SUT, be derived directly from the software specification or be targeted towards a specific property of interest in the SUT. Any given test-data will either satisfy the test constraint or not. To be able to apply directed search techniques to automate this generation, the distinction needs to be blurred. This is the role of the fitness function. It needs to return *good* values for test-data that satisfy (or nearly satisfy) the test constraint and *poor* values for test-data that are *far* from meeting the test constraint. At the point (or points) in the SUT's execution where the test constraint (or constraints) is required to be true, the fitness function routine is called to evaluate the suitability of the current test-data[6]. This idea is illustrated in figure 2.



```
procedure F
  (W : in     Unsigned_Byte_Type;
   X : in out Altitude_Type;
   Y : in out Fault_Code_Type;
   Z : in     Unsigned_Byte_Type)
is
   A : Unsigned_Byte_Type;
begin
   A := Interpolate (W, Z);
   if A <= 30 then
      X := 0;
      if Y = AFlt then
         Y = SFlt;
      else
         Y = LFlt;
      end if;
   else
      if X < Altitude_Type'Last then
         X := X + 1;
      end if;
   end if;
end F;
```

Figure 2: Testing as an optimisation problem

An important consequence of the directed search approach being dynamic (i.e actually executing the SUT) is that the fitness function routines have access to all information that is available at run-time. This includes the values of all local variables at the specific point of interest in the SUT's execution. Exactly how the SUT transforms the input test-data values to variable values at the point of the test constraint, and ultimately to output values, does not need to be known. This is highly advantageous as it is not necessary to employ complex, and typically incomplete, analysis in order to determine these relationships. These analysis techniques (e.g. symbolic execution) are often the source of lack of generality in static approaches to test-data generation.

[5]More generally, it can be considered as one or more constraints each of which must be true at specified points in the SUT's execution. This includes prior, post and arbitrary mid-points during the execution.

[6]When test constraints are internal to the SUT, instrumentation is used to insert calls to the fitness function calculation routines.

4.2 Genetic algorithms for test-data generation

Genetic algorithms provide a powerful general purpose search technique. Many aspects of a genetic algorithm implementation are independent of the specific test criteria and can, once implemented, remain fixed, thus helping to reduce the effort required to target new test criteria. Where appropriate, the framework implementation of the problem specific aspects of the search techniques have been taken from the literature. However, it has proved necessary to deviate from some of the existing approaches for the reasons discussed below.

4.2.1 Solutions and solution space

Genetic algorithms are directed search techniques that explore a solution space, attempting to locate solutions that give (near) optimal values of an fitness function. In order to apply genetic algorithms to test-data generation, the solution space representation must be considered.

A solution to a test-data generation problem is an input vector $x = \langle x_1, x_2, \ldots, x_n \rangle$. These individual solutions are selected from D, the input domain for the SUT. Thus the solution space for a test-data generation problem is represented by the input domain for the SUT.

An input vector is a collection of data values derived from the data-types of the underlying programming language used for the SUT. The Ada programming language [2, 16] is commonly used for the development of safety-critical software systems and has therefore been used as the SUT implementation language throughout this work. Ada allows compound data-types in the form of arrays and records. These can simply be broken down into collections of the more basic data-types resulting in an unchanged definition of solutions and the solution space.

4.2.2 Genetic algorithms implementation

Traditionally, genetic algorithms use a binary encoding of the solutions. This stems from the historical development of genetic algorithms. Holland's work focused on the use of binary encodings drawing an analogy between the individual bits and the role of chromosomes in natural evolution. The development of schema theory [13] gave a theoretical justification for the use of binary encodings. Holland used schemas to describe a genetic algorithm's ability to perform an implicitly parallel search of the solution space. Schema theory shows that a binary encoding, when compared to larger alphabet encodings with the same information carrying capacity, contain more schemes and therefore provide a greater ability for parallel search of the solution space.

It is now becoming widely accepted that binary encodings of the solution space are inferior compared to more natural solution space representations [28]. Using a natural encoding scheme can give significant performance improvements [17], removing the overheads of encoding and decoding. However, as a consequence, new implementations of the genetic operators are required to operate on the selected representation [10].

The applications of genetic algorithms to test-data generation presented in the literature use binary encodings [44, 42, 19, 18, 20]. However, for test-data generation, three problems can be caused by using binary encodings — disparity between the original and encoded solution space, information loss and information corruption.

The spatial disparity can cause problems as solutions that may be close to one another in the solution space can end up far apart in the encoded solution space. For example, a simple binary representation of the integer 31 is `0 1 1 1 1 1`. In the original solution space 31 and 32 are close together. However, in the binary encoding of the solution space 32 is represented as `1 0 0 0 0 0`. In this example, all six bits have to change to get from one solution to another. Clearly, mutation and crossover are going to find it difficult to cause the move between these two solutions, in effect causing them to appear far apart in the genetic algorithm's view of the search space. This can lead to a search that has difficulty converging and as a consequence is less efficient. It can also make it more difficult to design fitness functions that provide sufficient guidance to the search process.

Gray code has been suggested as a solution to the problem of the spatial disparity [7] and has been successfully used in test-data generation by Jones [18]. However, the use of Gray code does not help for non-numeric types such as enumeration types and the overhead of encoding/decoding can have a significant impact on the efficiency of the search. Also, Gray code does not eradicate the problem of information loss and corruption.

Information loss arises because the binary string is typically considered as a single entity. Indeed, this is the case for the genetic algorithm test-data generation approaches presented in the literature. However, in test-data generation, particularly for safety-critical software, this can cause problems. This is because information about the parameter boundaries in the binary string is lost causing the crossover operator to result in an effect somewhat similar to mutation. This is illustrated in figure 3.

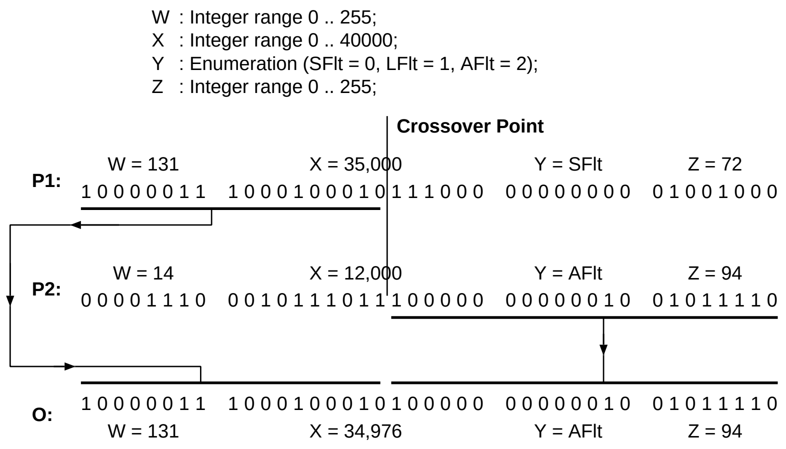


Figure 3: Information loss with binary encoding

The desired effect of crossover is to combine information from both parents in an attempt to produce an offspring with an improved fitness function value. In Figure 3, it can be seen that the effect of applying crossover has been to produce an offspring with the value W from P_1 and Y, Z from P_2. However, an entirely new value of X has been formed by combining part of the binary representation of X from P_1 and part from P_2.

Another possible effect of applying crossover on a binary encoding is information corruption. Typically, a strongly typed language (such as Ada [16]) would be used for the development of safety-critical systems. Variables will often only have valid values within a subset of the possible bit-patterns at the binary level. For example, a type for representing an aircraft's altitude in feet might be defined to range from 0 to $40,000$, while the underlying implementation of the type is likely to use, for example, a 16-bit unsigned value with a range of 0 to $65,535$. However, when generating test-data it is only meaningful to generate values in the range 0 to $40,000$ as values outside of this range would break the strong type-checking rules of the language[7]. Figure 4 shows the information corruption problem that can result by applying a crossover operator. It can be seen that the combination of partial bit-strings from P_1 and P_2 for value X results in a bit pattern that is not valid for the underlying type.

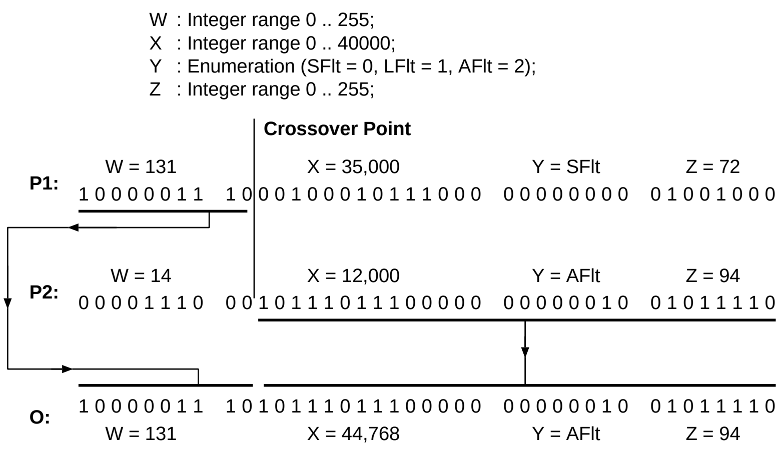


Figure 4: Information corruption with binary encoding

Clearly, a solution to both of these problems is to restrict the crossover points to the parameter boundaries. However, information corruption resulting in invalid bit patterns can still occur following the application of the mutation operator. This problem can be addressed in one of two ways. First, solutions with invalid data can be *punished* by ensuring that the fitness function returns poor values for such solutions. Second, these invalid solutions could be disregarded immediately after generation. However, neither of these approaches overcomes the efficiency problems of the binary encoding approach. In addition to the encoding/decoding overhead, there is an overhead in generating and examining solutions containing invalid data that can never represent the desired test-data.

For these reasons, the genetic algorithm implementation within the framework does not use a binary encoding of solutions, instead it utilises the representation described in section 4.2.1, i.e. the solution space is the input domain of the SUT, D, and a solution is an input vector, $x \in D$. To be able to apply a genetic algorithm to this solution space, it is necessary to implement crossover and mutation operators that can work with this non-binary representation. A range of crossover and mutation operators has been implemented within the framework — their selection and operation is controlled by a configuration file to allow their performance to be evaluated.

Figure 5 illustrates the functioning of the crossover operators implemented in the framework. *Simple* crossover selects a single crossover point. The offspring is then

[7]Jones [19] acknowledges this problem, but points out that a program cannot be considered fully tested unless such values are used during testing. In fact, this is incorrect. The use of a variable that has an invalid value for its underlying type causes an Ada program to become erroneous [16] at which point the code generated by the compiler no longer has to follow the semantic rules of the language.

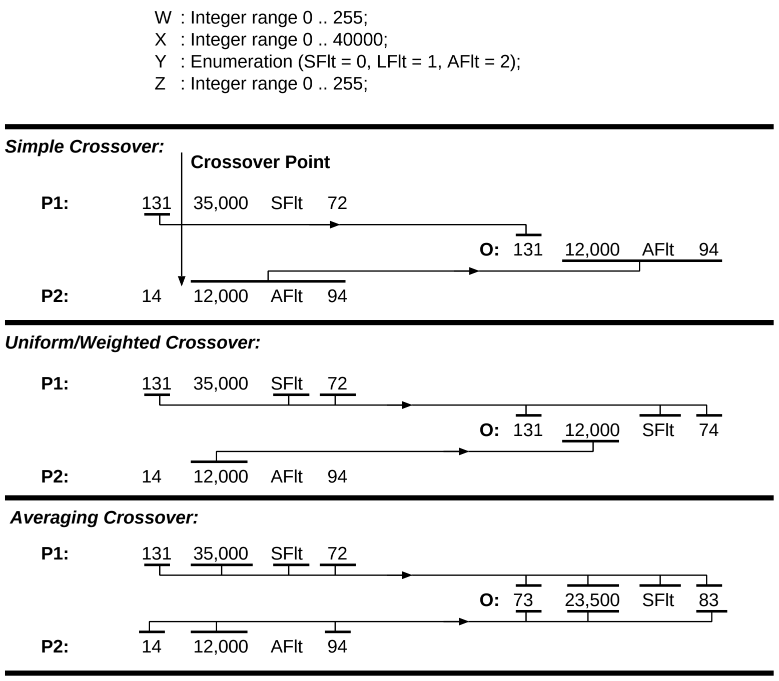


Figure 5: Crossover operators implemented in the framework

formed from values from P_1 up to the crossover point and from P_2 after this point. *Uniform* crossover is a generalisation of the single point crossover operator allowing an arbitrary number of crossover points. For each value, one of the two parents is randomly selected and this parent's value is used in the formation of the offspring. *Weighted* crossover is the same as uniform crossover except a configurable level of bias is given to one of the parents. Typically, this would be used to give a bias towards the parent with the better fitness function value. *Averaging* crossover behaves like uniform crossover for non-numeric data-types. However, for numeric data values the offspring value is formed by calculating the numeric mid-point between the two parent values.

Figure 6 illustrates the mutation operators implemented in the framework. *Simple* mutation sets a parameter value to a randomly generated valid value. *Random* mutation replaces an entire solution with a randomly generated solution. *Neighbourhood* mutation borrows the neighbourhood idea from simulated annealing. Parameter values are set to randomly sampled values within the neighbourhood of their current values.

4.3 Tool-support

Figure 7 shows the tool set that has been developed to support this test-data generation framework. The tool-set consists of an information extraction tool, a collection of fitness function generators, implementations of the search techniques and supporting source code that is independent of the SUT and test criterion.

The information extraction tool takes the SUT and the desired test-criterion and extracts the information required by the fitness function generators, it also produces an instrumented version of the SUT if this is required. The instrumented version of

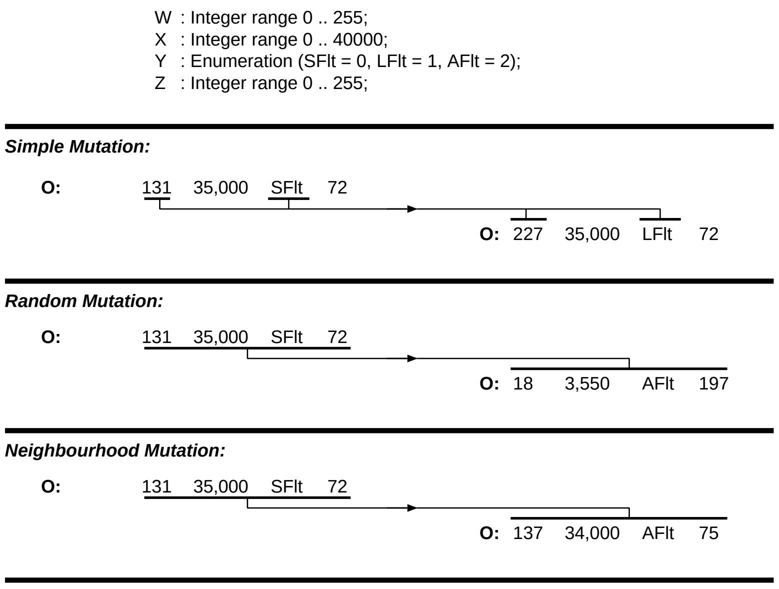


Figure 6: Mutation operators implemented in the framework

the SUT includes additional code to provide information required by specific fitness function modules, such as local variable values at specific points in the SUT. The information extraction tool is approximately $12,000$ lines of Ada code and is based on an extension to the parser and semantic analysis phases of the GNAT Ada compiler[8] [15]. This gives the information extraction tool access to the static semantic information for the SUT. The information extracted by this tool depends on the test criterion specified.

For all test criteria the parameter interface (formal parameter and global variables) of the SUT is extracted. This defines the solution space D and a solution $x \in D$. For white-box and grey-box criteria, information about the control-flow structure is extracted. For black-box criteria, information about the software specification (embedded in special comments in the SUT) is extracted.

Fitness function generators have been developed for a number of test criteria (as discussed later). These vary between $2,000$ and $3,500$ lines of Ada code. The fitness function generators tools take the information extracted from the SUT and generate the Ada source required to measure the effectiveness of test-data for the specified test criterion. By encoding a new test criterion as an fitness function module, the framework can be targeted at a new test criterion.

The generated fitness function, search technique implementation, supporting software and the SUT (or instrumented SUT if required by the specific test criterion) are compiled and linked together to form the test-data generation system. Running this executable initiates the test-data generation process by using the specified search technique to locate test-data to satisfy the specified test criterion.

[8]The parser and semantic analysis phases of the GNAT Ada compiler consist of approximately $240,000$ lines of Ada code.

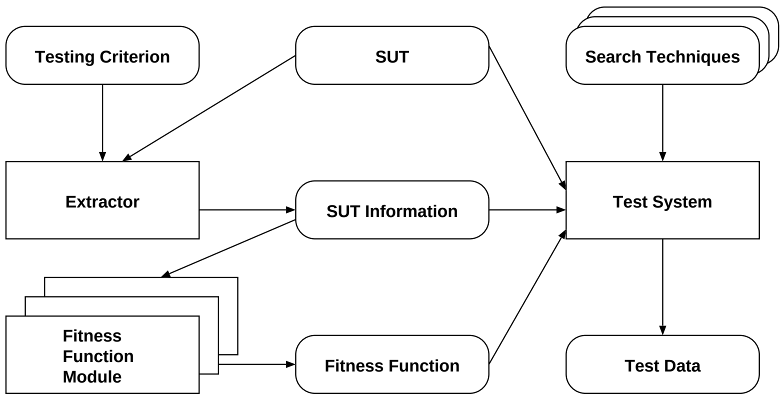


Figure 7: The search based framework

4.4 Search process

Test-data is generated by the framework by employing search techniques to locate optimal values of the fitness function. The search space is the input domain of the SUT. Consequently, an exhaustive search is likely to be intractable. For example, a simple routine to sort 50 32-bit integers with an execution time of 1ms would require a search time of almost 7 years for an exhaustive search. This has important consequences. If the search is not complete (i.e. exhaustive) then the results of test-data generation will also not be complete. Hence, if such a search fails to locate the desired test-data it cannot be concluded that such test-data does not exist. The failure to locate desired test-data may be used to provide *some* confidence that such test-data does not exist, but it is not possible to provide a guarantee. As Sherlock Holmes said *"the absence of evidence is not evidence of absence"*. It is also important that the test-data generated by the framework can be checked for suitability by other means. The generation of test-data exploits stochastic search techniques and may involve the execution of instrumented versions of the SUT. Therefore, it would be virtually impossible to qualify the use of the test-data generation framework in the context of safety-critical software development if the resulting test-data could not be independently checked. However, as the framework produces the test-data, existing trusted testing tools can be used to validate the test-data.

The framework implementation provides a number of search techniques, however only genetic algorithms are discussed here. The abstract process of searching for test-data is described below.

1. Extract required information from SUT.
2. Generate the fitness function implementation for the specified test-criterion.
3. Build test-data generation system
4. Generate candidate test-data using specified search technique.
5. Evaluate the test-data by running the SUT and using fitness function.
6. If the search stopping criteria are satisfied or the desired test-data has been located then stop, otherwise continue from step 4.

Figure 8 shows the test-data generation process using a genetic algorithm search. The search process starts with an initial population of randomly selected test-data. This population is then evaluated for suitability against the specified test criterion. This is achieved by running the SUT with the test-data from the population. The fitness function is used to measure the suitability of the test-data against the specified test-criterion. Once the entire population has been evaluated, the genetic algorithm selects members to become parents. These parents are then combined to form a new generation of test-data. Selected members of this generation are then mutated. This new population is then evaluated by executing the SUT and using the fitness function values. Test-data from the current and new generation are then selected to survive into the next generation of the search. This process continues until the search stopping criteria are satisfied. This can be when the desired test-data has been located or because the search is making no progress in finding the desired test-data.

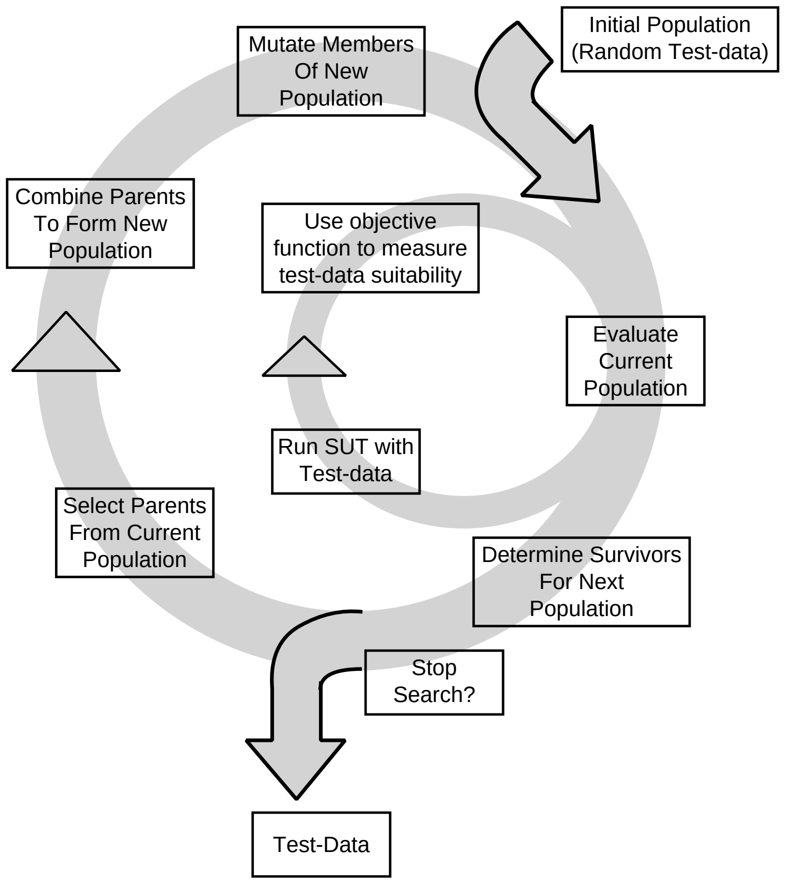

Initial Population (Random Test-data)
Mutate Members Of New Population
Combine Parents To Form New Population
Use objective function to measure test-data suitability
Evaluate Current Population
Run SUT with Test-data
Select Parents From Current Population
Determine Survivors For Next Population
Stop Search?
Test-Data

Figure 8: Genetic algorithm test-data generation process

5 Applications and Evaluations

The effectiveness of this framework centres on its ability to generate good quality test-data for industrially useful software testing criteria. The focus has been the testing of safety-critical software systems. As already stated, a test criterion can be addressed by providing the framework with an fitness function that quantified the suitability of test-data for the criterion. Figure 9 shows the testing criteria for which fitness function modules have been implemented (also see [37, 36, 38, 39, 41]). The figure mirrors the structure of the remainder of this section, where the design of fitness function for several of these applications are discussed in more detail.

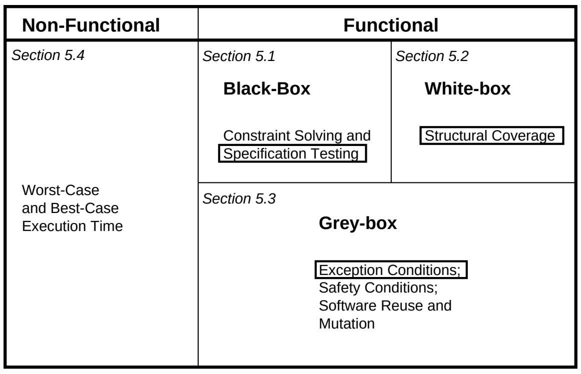


Figure 9: Framework applications

5.1 Black-box

This section describes the construction of an fitness function for the problem of testing software conformance to a specification. For highly safety-critical software systems, it may be the case that formal proofs must be developed. These proofs must show that the software is a refinement of the specification. Producing formal proofs is a complex, time-consuming and expensive process. If the putative properties are simply untrue then attempting a proof (that will inevitably fail) is an expensive method to find errors in the software. To demonstrate that a proof will fail, it is only necessary to find a single test-case that shows the specification is not satisfied. It is exactly this application to which the framework is applied here.

For the purposes of evaluation a notation similar to SPARK-Ada [33, 3] proof contexts has been used to represent the software specification. These proof contexts consist of a pre- and post-condition for the subprogram. These conditions consist of an Ada expression, extended to allow logical implication and equivalence. Figure 10 shows how the formal specification is expressed as part of the subprogram specification for a simple increment subprogram.

The desired test-data for this application are test-data that satisfy the pre-condition constraint, but cause the post-condition to be false. The fitness function needs to return good values for those test-cases which *nearly* meet the criterion and worse values for those which are a long way away from meeting the criterion. For example, consider the constraint $X > 50$. If $X = 49$ this constraint is a lot *nearer* to being true than when

$X = 2$ (however it remains a fact that they are both false, we just consider one to be less false than the other!).

The pre- and post-conditions are made up from relational expressions connected with logical operators. The fitness function is therefore designed to exploit information in the conditions to indicate how *near* particular test-data are to satisfying these conditions. The pre-condition is conjoined with the negated post-condition. To improve the power of the search the pre-condition and negated post-condition are first converted to Disjunctive Normal Form (DNF). For example, the condition $A \rightarrow (B \vee (C \wedge D))$ would become $\neg A \vee B \vee (C \wedge D)$. This causes the search to be finer grained as a solution to any one disjunct (i.e. $\neg A$, B or $(C \wedge D)$) represents a solution to the entire condition. Each pre-condition/negated post-condition disjunct pair is targeted in a separate search attempt. These pairs can be considered as an encoding of one of the possible situations in which the software can fail to implement the specification. The SUT is executed with the currently generated input data and a set of output data obtained. The fitness function is then calculated according to the rules in Table 1. The value K in the table refers to a failure constant that is added to further punish test-data that causes a term to be untrue. Each element is evaluated and contributes to the overall fitness function value. The value is then returned to the search technique to guide its selection of new input data.

Element	Value
Boolean	`if TRUE then` 0 `else` K
$a = b$	`if` $abs(a-b) = 0$ `then` 0 `else` $abs(a-b) + K$
$a \neq b$	`if` $abs(a-b) \neq 0$ `then` 0 `else` K
$a < b$	`if` $a - b < 0$ `then` 0 `else` $(a-b) + K$
$a \leq b$	`if` $a - b \leq 0$ `then` 0 `else` $(a-b) + K$
$a > b$	`if` $b - a < 0$ `then` 0 `else` $(b-a) + K$
$a \geq b$	`if` $b - a \leq 0$ `then` 0 `else` $(b-a) + K$
$a \wedge b$	$z(a) + z(b)$
$a \vee b$	$min(z(a), z(b))$
$a \Rightarrow b$	$z(\neg a \vee b)$ $\equiv min(z(\neg a), z(b))$
$a \Leftrightarrow b$	$z((a \Rightarrow b) \wedge (b \Rightarrow a))$ $\equiv z((a \wedge b) \vee (\neg a \wedge \neg b))$ $\equiv min((z(a) + z(b)), (z(\neg a) + z(\neg b)))$
a xor b	$z((a \wedge \neg b) \vee (\neg a \wedge b))$ $\equiv min((z(a) + z(\neg b)), (z(\neg a) + z(b)))$

Table 1: Fitness Function Calculation

A useful feature of the fitness function as described above is that whenever the pre-condition and negated post-condition are satisfied (i.e. the desired test-data is located) the fitness function evaluates to zero. This gives a highly efficient stopping criterion for the search.

5.1.1 Example

To illustrate how this fitness function provides the required guidance to a search technique, consider the following simple examples. Figure 10 shows the specification of a simple wrap-round increment counter — counting from 0 to 10 and then wrapping back to 0. For this example, we will assume the implementation is incorrect because it wraps at 11 and not 10 (possibly due to using $>$ rather than $\geq$ in the implementation). The goal is to locate test-data that illustrates this lack of conformance to the specification.

```
function Increment (N : in Integer) return Integer;
--# pre   N >= 0 and N <= 10;
--# post (N < 10 -> Increment = N + 1) and
--#        (N = 10 -> Increment = 0);
```

Figure 10: Specification of Wrap-Round Increment Routine

The first step is to convert the pre-condition and the negated post-condition to DNF and form all pairs of pre-condition/post-condition disjuncts. The DNF of the negated post-condition consists of two disjuncts (shown in parenthesis below) hence there are two possible pairings as follows.

$$N \geq 0 \wedge N \leq 10 \wedge (N < 10 \wedge Increment \neq N + 1) \tag{1}$$

$$N \geq 0 \wedge N \leq 10 \wedge (N = 10 \wedge Increment \neq 0) \tag{2}$$

Table 2 shows fitness function values for sample input data the search technique might generate for the second of these disjuncts. It can be seen that as the input data get closer to satisfying the disjunct terms, the fitness function value decreases. Indeed, with input-data of $N = 10$ the fitness function returns zero. This indicates that suitable test-data has been found illustrating the non-conformance of the implementation.

5.1.2 Evaluation

This section presents an evaluation of the framework's ability to generate test-data that highlights implementation errors with respect to a given specification. The purpose of the evaluation is to assess the relative performance of the various search techniques. The framework has been evaluated with software containing known errors. These errors have been inserted into correct implementations using a manual ad hoc approach and a systematic approach. The manual ad hoc insertion of errors (or error seeding) aimed to simulate realistic errors that might be made in practice. In addition, mutation testing has been used to systematically introduce errors into the SUT. Mutation testing is a fault based testing technique designed to show the absence of specific faults, known as *mutants*. Mutants are constructed systematically by introducing simple syntactic changes one at a time into the SUT, producing a set of mutant versions of the SUT. The aim of the syntactic changes is to mimic common coding errors made by programmers (e.g. the use of $>$ in place of $\geq$). The goal of mutation testing is to kill these mutants. A mutant is killed by generating test-data to distinguish the output

Test-data	Disjunct terms	z() contribution
Inp: $N = 2$, Out: $Increment = 3$	$N \geq 0$	0
	$N \leq 10$	0
	$N = 10$	$8 + K$
	$Increment \neq 0$	0
	Final fitness func value	$z() = 8 + K$
Inp: $N = 7$, Out: $Increment = 8$	$N \geq 0$	0
	$N \leq 10$	0
	$N = 10$	$3 + K$
	$Increment \neq 0$	0
	Final fitness func value	$z() = 3 + K$
Inp: $N = 10$, Out: $Increment = 11$	$N \geq 0$	0
	$N \leq 10$	0
	$N = 10$	0
	$Increment \neq 0$	0
	Final fitness func value	$z() = 0$

Table 2: Black-box Fitness Function

state of the mutant SUT from the original SUT[9]. The specification describes the correct output state for the SUT. Consequently, test-data that shows a failure to correctly implement the specification (i.e. the post-condition is not satisfied) will also kill a mutant (as the output states of the original SUT satisfy the post-condition but the mutant does not). The ability of the framework to kill mutants can therefore be used to evaluate the framework's effectiveness at illustrating implementation errors. Some of the mutants will be functionally identical to the original program and hence cannot be killed. These are known as equivalent mutants. A mutation score can be defined as follows:

$$\text{Mutation Score} = \frac{\text{Killed Mutants}}{\text{Number of Mutants} - \text{Equivalent Mutants}} \times 100 \tag{3}$$

The aim is to achieve a mutation score of 100% automatically by targeting the framework at illustrating implementation errors.

The system being used during this evaluation is a safety-critical nuclear primary protection system. Access to the software specification and formally proven implementation was provided by Roll-Royce Plc. The system consists of a six separate channels, each controlling a different aspect of functionality. Each channel is specified and implemented as a separate system. Two of the six channels were made available for the evaluation. The channels are specified formally in VDM-SL [1] — channel 1 consists of 36 pages of specification and channel 2 consists 54 pages. The implementation of these channels was written in Pascal and a formal proof of refinement was developed — channel 1 consists of 2200 lines of code and channel 2 consists of 9700

[9]Technically, this kind of mutation testing is known as *strong mutation* because the observable output of the mutant and original program must differ. Weak mutation [14], in contrast, only requires the intermediate state following the execution of the mutated statement to be different.

lines of code[10]. For the purposes of the evaluation, the Pascal code was translated, using an automated translator, into Ada. The VDM-SL specification was hand-translated into an assertion-based specification consisting of a pre-condition assertion and a post-condition assertion for each of the functions in the implementation. A mutation testing tool was used to generate a number of mutant implementations of each of the functions. The framework was targeted at generating test-data that satisfied the pre-condition, but falsified the post-condition for each of these mutants. Any such test-data highlights the effect of the mutant in causing the implementation to be flawed. Figure 11 shows the mutation report for an example routine used in the evaluation (the variable names used have been obscured to facilitate publication).

The results of the evaluation are shown in tables 3. The table shows the mutation score for both a genetic algorithm search and a simple random search. As can be seen, both random testing and the genetic algorithm perform well. On the two larger functions, the genetic algorithm performed better, showing that 100% of the mutants caused the specification to be broken. In all cases the genetic algorithm was at least as quick as the random search and on the two larger functions it was an order of magnitude faster. The simplicity of this safety-critical code allows a simple random search to work well. However, as the search space and code complexity increases the genetic algorithm based approach offers significant advantages.

	LOC	Errors	Equiv	Random	Genetic Algorithm
1	85	48	0	100%	100%
2	438	22	3	73.7%	100%
3	34	6	0	100%	100%
4	53	20	0	100%	100%
5	591	30	9	95.2%	100%
6	572	60	6	87.0%	100%

Table 3: Specification conformance evaluation (success rate) — LOC: Lines Code; Errors: Number of erroneous versions of the SUT produced; Equiv: Number of erroneous versions that were equivalent to the SUT.

The level of automation in the approach means it can be targeted at finding implementation errors as soon as a software unit and its specification are available. In comparison, the functional testing carried out on the project could only take place once a full release of the software was available. By targeting the automated test-data generation framework at specification testing earlier in the life-cycle, implementation errors can be located earlier. This gives the potential for significant cost savings as the amount of re-work following the discovery of an error is minimised. Also, as the approach is targeted at finding negative test-data it is also possible that errors will be located that would not be found with success-oriented functional testing. For instance, consider the example routine in Figure 11, assuming mutant 6 represents the software as implemented by the developer. Table 4 shows a set of tests that could be used during testing to achieve MC/DC coverage[11] of this incorrect implementation. All of these tests pass,

[10]The two channels together contain approximately $2,000$ lines of executable code with large data-tables and comments filling the remaining $10,000$ lines.

[11]In fact, full MC/DC coverage is not possible for this program. The statement on line 33 is defensive code which in practice is not executable. Also, it is not possible to execute the loop on lines 24–27 zero times.

```
 2 package body Channel is
 4  procedure C_W
 5   (RG : in RG_Type;
 6    RN : in RN_Type;
 7    I : in I_Type;
 8    W : out W_Type)
 9  is
10    I                  : Table_Values;
11    A, B, C, J, K      : Integer;
12  begin
13    I_Vals := I_Tables (RG) (RN);
14    W_Vals := W_Tables (RG) (RN);
16
17    if (I <= I_Vals (I_Vals'First)) then
18      W := W_Vals (W_Vals'First);
19    else
20      if (I >= I_Vals (I_Vals'Last)) then
21        W := W_Vals (W_Vals'Last);
22      else
23        I := Table_Values'Last;
24        while (I_Vals (I) > I) loop
26          I := I - 1;
27        end loop;
28        J := W_Vals (I + 1) - W_Vals (I);
29        K := I - I_Vals (I);
30        A := J * K;
31        B := I_Vals (I + 1) - I_Vals (I);
32        if (B = 0) then
33          C := 0;
34        else
35          C := A / B;
36        end if;
37        W := W_Vals (I) + C;
38      end if;
39    end if;
40  end C_W;
42 end Channel;

-- Mutant 1 : replace <= by >, line 17
-- Mutant 2 : replace <= by <, line 17
-- Mutant 3 : replace >= by <, line 20
-- Mutant 4 : replace >= by >, line 20
-- Mutant 5 : replace > by <=, line 24
-- Mutant 6 : replace > by >=, line 24
-- Mutant 7 : replace - by +, line 26
-- Mutant 8 : replace - 1 by - 2, line 26
-- Mutant 9 : replace + by -, line 28
-- Mutant 10 : replace + 1 by + 2, line 28
-- Mutant 11 : replace - by +, line 28
-- Mutant 12 : replace - by +, line 29
-- Mutant 13 : replace * by /, line 30
-- Mutant 14 : replace + by -, line 31
-- Mutant 15 : replace + 1 by + 2, line 31
-- Mutant 16 : replace - by +, line 31
-- Mutant 17 : replace = by /=, line 32
-- Mutant 18 : replace = 0 by = 1, line 32
-- Mutant 19 : replace := 0 by := 1, line 33
-- Mutant 20 : replace = 0 by = 1, line 33
-- Mutant 21 : replace / by *, line 35
-- Mutant 22 : replace + by -, line 37
```

Figure 11: Example evaluation routine

showing the implementation to be correct with respect to its specification. However, the test-data generation framework generated the test $RG = 4, RN = 0, I = 4046$ that shows the implementation to be *incorrect* because of the boundary error in the loop condition. The targeting of negative test-data has allowed the test-data generation framework to find an error where the success-oriented structural coverage testing does not.

RG	RN	I	Covers
3	5	100	4–18, 40
2	6	5000	4–17, 20, 21, 40
4	1	3694	4–17, 20, 23, 24–32, 35-40
1	7	1993	4–17, 20, 23, $\{24–27\}^2$, 28–32, 35-40
1	7	1600	4–17, 20, 23, $\{24–27\}^*$, 28–32, 35-40

Table 4: MC/DC tests for program in Figure 11

5.2 White-box

The fitness function presented for black-box testing can be used as the basis for addressing white-box test criteria. In white-box testing we are concerned with finding test-data such that the executed path will cover (or execute) the desired statement, branch, LCSAJ, etc. The fitness function needs to return good values for test-data that *nearly* execute the desired statement (or branch, etc.) and poor values for test-data that are a *long way* from executing the desired statement. Branch predicates determine the control-flow and are therefore vital in determining an effective fitness function.

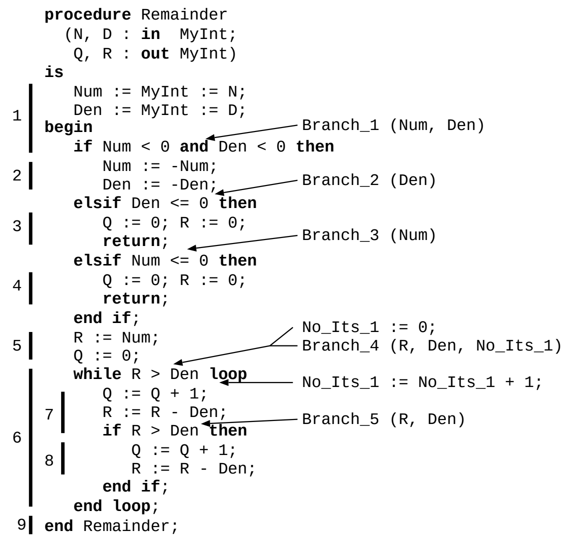
Branch predicates consist of relational expressions connected with logical operators. This allows the same fitness function as for black-box testing criteria to be used. However, in white-box testing, the fitness function requires evaluation with intermediate values of variables from the SUT. Consequently, it is necessary to execute an instrumented version of the SUT. This instrumented version of the SUT is produced by the extractor tool by adding branch evaluation calls into the SUT. These branch evaluation calls replace the branch predicates in the SUT. They are responsible for calculating the fitness function value for each individual branch predicate that is executed. They also return the boolean value of the predicate to allow the instrumented SUT to continue to follow the same control-flow path as the original SUT. The branch evaluation functions work as follows:

- If the target node is only reachable if the branch predicate is true then add $z(\text{branch predicate})$ to the overall fitness function value for the current test-data. If branch predicate must be false then the cost of $z(\neg\ \text{branch predicate})$ is used.
- For loop predicates the desired number of iterations determines whether the loop predicate or $(\neg\ \text{loop predicate})$ is used.
- Within loops, adding the fitness function value of branch predicates is deferred until exit from the loop. At this point the minimum cost evaluated for that branch predicate is added to the overall cost. This prevents punishment of taking an undesirable branch until exit from a loop, as the desirable branch may be taken on subsequent iterations.

The fitness function gives a quantitative measure of the suitability of the generated test-data for the purpose of executing the specified node in (or path through) the SUT.

5.2.1 Example

To illustrate how this fitness function provides the required guidance to a search technique consider the following simple example. Figure 12 shows a simple program to calculate the remainder and quotient given a numerator and denominator. The figure also illustrates the instrumentation that is performed by the extractor tool to obtain a version of the SUT that calculates the value of the fitness function during its execution. The nodes of a control-flow graph for the software are shown on the left of the figure.



```
      procedure Remainder
        (N, D : in  MyInt;
         Q, R : out MyInt)
      is
         Num := MyInt := N;
  1      Den := MyInt := D;
      begin                              Branch_1 (Num, Den)
         if Num < 0 and Den < 0 then
  2         Num := -Num;
            Den := -Den;                 Branch_2 (Den)
         elsif Den <= 0 then
  3         Q := 0; R := 0;
            return;                      Branch_3 (Num)
         elsif Num <= 0 then
  4         Q := 0; R := 0;
            return;
         end if;                         No_Its_1 := 0;
  5      R := Num;                       Branch_4 (R, Den, No_Its_1)
         Q := 0;
         while R > Den loop              No_Its_1 := No_Its_1 + 1;
            Q := Q + 1;
  7         R := R - Den;                Branch_5 (R, Den)
  6         if R > Den then
  8            Q := Q + 1;
               R := R - Den;
            end if;
         end loop;
  9   end Remainder;
```

Figure 12: Example SUT

For the purpose of this example we assume the desired test-data are required to execute the path $\langle 1, 2, 5, 6_{(2+)}, 7, 8, \ldots, 9\rangle$, i.e. execute nodes 1, 2, 5 and then loop 6 two or more times including executing nodes 7 and 8 and then any path leading to node 9. Table 5 shows how the fitness function evaluates for different input data the search technique might generate. Again, as the input data get closer to satisfying the disjunct terms, the fitness function value decreases, ultimately reaching zero once appropriate test-data has been located.

While white-box testing is useful in its own right, it has not been a focus for the application of the framework. We have preferred to concentrate on testing techniques that target errors in the implementation. In addition, while it is possible to use the framework to generate white-box test-data, it is only possible to generate the input-data. The expected output-data must still be determined from the specification requiring a significant amount of effort.

5.2.2 Evaluation

The evaluation has been performed using a number of subsystems from the implementation of a civil aircraft engine controller. This sample represents 7% of the executable

Test-data	Branch Eval Calls	z() contribution
N = 5,	Branch_1	5 + K
D = -10	Execution stopped and returned to search technique	
	Final fitness func value	$z() = 2 + K$
N = -5,	Branch_1	0
D = -3	Branch_4	0
	Branch_5	1 + K
	Branch_4	1 + K
	Branch_5 contributes on exit from loop	1 + K
	Execution stopped and returned to search technique	
	Final fitness func value	$z() = 2 + K$
N = -15,	Branch_1	0
D = -2	Branch_4 Iteration 1	0
	Branch_5	0
	Branch_4 Iteration 2	0
	Branch_5	0
	Branch_4 Iteration 3	0
	Branch_5	0
	Branch_4 Iteration 4	0
	Branch_5	1 + K
	Branch_4 Iteration 5	0
	Branch_5 contributes on exit from loop	0
	Final fitness func value	$z() = 0$

Table 5: White-box Fitness Function

lines of code from the engine controller implementation (approximately 7,000 lines of code from the 100,000 line FADEC software).

For this evaluation, the goal has been to generate test-data to achieve 100% boolean coverage for the SUT. Table 6 shows the results of applying a random search and a genetic algorithm search to generate structural test-data for the engine controller sample routines.

From these results, it can be seen that random search perform well. Random search was allowed to generate 50,000 tests in its search for suitable test-data. For the simple routines, it located test-data extremely efficiently and successfully. However, for the more complex routines, random search was both less successful and significantly less efficient. However, the genetic algorithm significantly more efficient. For routines 1 to 9, full coverage was achieved (taking into account the infeasible conditions in routines 8 and 9). For routine 10, the 89% coverage achieved by the genetic algorithm failed to generate test-data for 2 further conditions that were feasible. Increasing the population sizes by 25% and increasing the length of time the search is allowed to continue with no progress enabled test-data to be generated for these two test-cases. On average, this increased the search times by just under 1 minute, resulting in the longest search time still only being 11 minutes.

Subsystem	GA Hill	Random
1.	100%	100%
2.	100%	100%
3.	100%	100%
4.	100%	100%
5.	100%	90%
6.	100%	42%
7.	100%	42%
8.	89%	66%
9.	86%	47%
10.	89%	82%

Table 6: Evaluation of test-data generation for structural testing

5.3 Grey-box

Grey-box testing is a combination of black-box and white-box testing. In practice, many testing problems fall into this class. Here tests are derived from a specification of the desired behaviour but with reference to the implementation details. The particular testing problem we address in this section is that of testing the exception conditions (other problems such as re-use testing and safety-invariant testing have been addressed in [39, 40]).

The exception handling code of a system is, in general, the least documented, tested and understood part, since exceptions are expected to occur only rarely [25]. The importance of testing exception conditions was illustrated in a recent incident, where the Ariane 5 launch vehicle was lost due to an unhandled exception destroying $400 million of scientific payload [27]. Safety-critical systems present special problems. Typically, it is easier to prove the absence of any exceptions than to prove that the exception handling is safe. Again, before carrying out a proof, confidence in the property's truth is desirable. To show that such a proof would fail, it is only necessary to generate a single counter-example. That is test-data that cause an exception to be raised.

In the work to date, the Ada language [16] model of exceptions has been used. To address this problem within the framework an fitness function is required. As might be expected, this fitness function is a combination of the black- and white-box fitness functions. The white-box fitness function is used to guide the search towards test-data that execute the desired statement. The black-box fitness function is used to cause the run-time check associated with the exception to be false.

As in white-box testing, an instrumented version of the SUT is produced. In addition to the branch evaluation calls, constraint evaluation calls are also added. These constraint evaluation calls encode the condition required to cause a run-time check to fail and hence raise an exception. They function as follows.

- If the exception is required to be raised then the $z(\neg\text{run time check})$ is added to the overall cost for the current test-data.

- If the exception is required not to be raised then $z(\text{run time check})$ is added to the overall cost for the current test-data.

- Again, as for branch evaluation calls, if the current test-data causes the exception to be raised when it is not desired (or vice-versa) then execution of the SUT is terminated and the fitness function value returned to the search procedure.

- As for branch evaluation calls, adding the fitness function value of exception conditions is deferred until exit from the loop.

This fitness function provides two fold guidance, guiding the search toward test-data that both executes the statement of interest and violates the associate run-time check required to raise the exception.

5.3.1 Example

To illustrate how this fitness function provides the required guidance to a search technique consider the following simple example. Figure 13 shows a simple function that is capable of raising exceptions.

```
function F (X, Y : Small_Int) return Integer is
 Z : Integer;
begin
 if X < 0 then
    raise Invalid_Data;
 end if;
 Z := X + Y;
 if Z > 1 and Z <= 5 then
    return Integer'Last;
 else
    return ((X ** 4) / ((Z - 1) * (Z - 5)));
 end if;
end F;
```

Figure 13: Example SUT

For the purposes of this example, assume the desired test-data are that which will cause the divide-by-zero exception. Table 7 shows how the fitness function, $z()$, evaluates.

Test-data	Branch and Constraint Calls	z() contribution
X = 2,	Branch_1 (X $\geq$ 0)	0
Y = -5	Exception	$32 + K$
	Execution stopped and returned to search technique	
	Final fitness func value	$z() = 32 + K$
X = 5,	Branch_1 (X $\geq$ 0)	0
Y = -3	Exception	$3 + K$
	Execution stopped and returned to search technique	
	Final fitness func value	$z() = 3 + K$
X = 5,	Branch_1 (X $\geq$ 0)	0
Y = -4	Exception	0
	Final fitness func value	$z() = 0$

Table 7: Grey-box Fitness Function

5.4 Evaluation

The evaluation has been performed using a number of subsystems from the implementation of a civil aircraft engine controller. The evaluation examines the integration of test-data generation and static techniques for determining exception freeness. In the final production version of the engine controller software, the run-time checks are disabled. It is therefore important that there is no possibility of a run-time exception. The implementation of the engine controller software is written in SPARK-Ada. Consequently, only `Constraint_Error` exceptions need to be considered further.

The subsystems were selected so as to cover the diverse functionality of the engine controller in the case-study. Each of software units within each subsystem was tested in turn.

- Subsystem 1 — Data input and validation routines for continuous data types used within the engine controller implementation.
- Subsystem 2 — Calculates parameters used as the basis for control of the engine from validated sensor information.
- Subsystem 3 — Data input and validation routines for discrete data types used within the engine controller implementation.
- Subsystem 4 — Control functionality for the fuel metering valve.
- Subsystem 5 — Control functionality for the variable stator vanes.
- Subsystem 6 — Calculates parameters used as the basis for control of the engine from validated sensor information and distributes to were they are used.
- Subsystem 7 — Maintenance activities.

This slice of the engine controller implementation is approximately 30,000 lines of code (or approximately 30% of the total implementation). The subsystems from the engine controller were first passed through the SPARK Examiner to extract the verification conditions for exception freeness. These verification conditions were then passed through the SPADE automatic simplifier. This discharges many of the verification conditions. For the engine controller implementation, 89.2% of the verification conditions were discharged automatically by the simplifier. This was further improved by providing the simplifier with information about the compiler-dependent types such as the ranges of `Integer`s and `Float`s. With the aid of this information, a further 2.4% of the verification conditions were discharged. The simplifier performs basic syntactic simplification. It was possible to discharge another 6.1% of the verification conditions by exploiting simple semantic information. For example, the simplifier was not able to prove the following verification condition:

```
H1: vara >= 0 .
H2: vara <= 6 / 5.
H3: varb >= 0 .
H4: varb <= 16 .
       ->
C1: vara * varb >= -214783647 / 8192 .
C2: vara * varb <= 214783647 / 8192 .
```

The results of extracting exception freeness verification conditions and automated simplification are summarised in Table 8.

Subsystem	VCs	Discharged using simplification	Discharged using simple semantic information	Remaining VCs
1	141	101	26	14
2	61	61	0	0
3	232	223	7	2
4	274	239	26	9
5	136	98	26	12
6	96	63	25	8
7	1008	999	8	1
Total	1948	1784 91.6%	118 6.1%	46 2.4%

Table 8: Extraction and simplification of verification conditions

As a result of the simplification and exploitation of simple semantic information, less than 3% of the verification conditions remain unproven. To show exception freeness of the software it is necessary to discharge these remaining proof obligations. To do this, proofs need to be constructed. However, manual proof attempts can be effort-intensive and therefore extremely expensive, especially if the property being proven is untrue. In these cases, a proof attempt will inevitably fail. To build confidence in the correctness of the property being proven, the test-data generation framework was targeted to search for a counter-example to the proof. Each check annotation associated with an unproven verification condition was used as the target for test-data generation.

Table 9 shows the results of applying the framework to generate test-data that violates the check annotations. (All verification conditions were automatically discharged for subsystem 2. It therefore does not appear in the table).

Subsystem	Number of VCs	GA	Random
1	14	14	1
3	2	2	0
4	9	1	0
5	12	10	10
6	8	8	8
7	1	0	0

Table 9: Test-Data generation for exception conditions — showing number of counter-examples located by test-data generation and number of samples.

Subsystem 1 — the unproven verification conditions for this system stem from three subroutines. Two of these subroutines carried out basically the same functionality, one at initialisation time and one during normal engine operation. These routines contribute six unproven verification conditions each. Test-data generation successfully located test-data showing these verification conditions could be violated and exceptions raised at run-time. The genetic algorithm search located test-data for all of the verification conditions. The random search search did not locate any test-data. In the third routine, there were two unproven verification conditions. The genetic algorithm search techniques was 100% successful in locating test-data to show that the associated run-time exceptions could be raised. The random search successfully located test-data to show that one of the verification conditions could be violated.

Subsystem 3 — only one routine in this subsystem contained unproven verification conditions. The random search failed to successfully locate test-data to show that run-time exceptions could be raised. However, the genetic algorithm was successful in locating test-data.

Subsystem 4 — the unproven verification conditions for this system stem from two subroutines. The first of these subroutines contains a single unproven verification condition. Test-data generation was successful with the genetic algorithm search but not with the random search. The second subroutine contains eight unproven verification conditions. The test-data generation attempts for these eight verification conditions yielded no test data. An examination of the implementation shows that data-flow through the system is such that these verification conditions can never be violated. While this does not constitute a proof of exception freeness, an informal justification was constructed.

Subsystem 5 — the unproven verification conditions for this system stem from two subroutines. The first of these subroutines contains ten unproven verification conditions. Both search techniques were 100% successful in showing that run-time exceptions could be raised. The second subroutine contains two unproven verification conditions. The test-data generation attempts for these conditions yielded no test-data. Again, an informal justification was constructed to show that no test-data was located because the run-time checks could never fail.

Subsystem 6 — only one routine in this subsystem contained unproven verification conditions. Both search techniques were successful in generating test-data to show that the eight unproven verification conditions could be violated.

Subsystem 7 — one routine in this subsystem contained a single unproven verification condition. The test-data generation attempts for these conditions yielded no test-data. Again, an informal justification was constructed to show that no test-data was located because the run-time checks could never fail.

As already stated, for the final engine controller system the run-time checks are turned off. This means that exceptions would not be raised, but rather that data values would become invalid. This could have serious safety implications for the system as the engine control-laws may not be stable with invalid data. It is therefore of concern that the test-data generation system located test-data that raises run-time exceptions. A detailed investigation into these situations showed that violation of the run-time rules (and hence potentially invalid data) was not possible in the current system. The use of protected arithmetic operators which are well-defined in the presence of divide-by-zero, overflow and underflow prevented a number of these cases. However, in these cases the resulting test-data is still interesting because the arithmetic operators return a mathematically incorrect result. In general, it is important to know the situations when this can happen.

The physical value ranges of sensor readings prevented a number of exception conditions occurring in practice. Also, the overflow of counters, another potential cause of exceptions, was mitigated against by global configuration data. Figure 15 shows the remaining two verification conditions for the smooth signal subprogram in Figure 14 (the irrelevant hypotheses have been removed) that illustrate these two types of potential exceptions. Test-data was generated for each, illustrating that an exception could be raised. For example $Count$ input value of 100 when either $(CurrentVal - GoodVal) > SmoothThresh$ or $(GoodVal - CurrentVal) > SmoothThresh$ is true will cause an exception. In practice however, the hardware sensors that provide values for $currentval$ and $goodval$ ensure that $currentval - goodval$ is always in range and global configuration data ensures $Count \leq 10$.

```
type RealType is delta 0.0001
  range -250000.0 .. 250000.0;

type CounterType is range 0 .. 100;

procedure SmoothSignal
 (CurrentVal        : in        RealType;
  SmoothThresh      : in        RealType;
  GoodVal           : in  out   RealType;
  OutputVal         :     out   RealType;
  Count             : in  out   CounterType;
  CountThresh       : in        CounterType)
is
   Tmp1, Tmp2 : RealType;
begin
   Tmp1 := CurrentVal - GoodVal;
   Tmp2 := GoodVal - CurrentVal;
   if Tmp1 > SmoothThresh or else
      Tmp2 > SmoothThresh
   then
      Count := Count + 1;
      if Count < CountThresh then
         OutputVal := GoodVal;
      else
         OutputVal := CurrentVal;
         GoodVal := CurrentVal;
         Count := 0;
      end if;
   else
      OutputVal := CurrentVal;
      GoodVal := CurrentVal;
      Count := 0;
   end if;
end SmoothSignal;
```

Figure 14: Smooth Signal Subprogram

The use of fixed global data stored as part of the engine configuration prevents many exceptions from actually occurring. In all cases, this global data was passed in by the calling environment and contained values such that the counter could never overflow. This is the case in the smooth signal example where the global configuration ensures the counter is reset after at most 9 iterations (i.e. $SmoothSignal$ is never called with $CountThresh$ above 9). Other expressions depended on data interpolated

```
H1:    currentval >= - 250000 .
H2:    currentval <= 250000 .
H3:    goodval >= - 250000 .
H4:    goodval <= 250000 .
       ->
C1:    currentval - goodval >= - 250000 .
C2:    currentval - goodval <= 250000 .
```

```
H1:    count >= 0 .
H2:    count <= 100 .
       ->
C1:    count <= 99 .
```

Figure 15: Verification Conditions for Smooth Signal

from static data-tables. These tables were such that overflow in the expressions could not occur. However, the test-data generated is still useful as the basis of a code-review to ensure that the global configuration does indeed prevent such data occurring at run-time. For those verification conditions where the test-data generation was unsuccessful, proofs were attempted. In all cases these were successful in discharging the verification conditions.

It is the lack of inter-procedural data-flow information that causes test-data to be generated that the system itself could never generate in practice. For example, as discussed above the $SmoothSignal$ is never called with a $CountThresh$ greater than 9. However, during test-data generation, when the $SmoothSignal$ subroutine is considered in isolation, this fact is lost. One possible approach to address this problem is to use the SPARK-Ada pre-condition annotations. These would capture the necessary information so that the test-data generation could be targeted to locating test-data to satisfy the pre-condition assertion but raise the run-time exception. Indeed, the supply of such information allows even more of the verification conditions to be proved automatically by simplifier. However, the construction of such annotations can be very expensive and for many industrial safety-critical systems they are simply not available (as is the case with the aircraft engine controller code used in the evaluation). Even with these annotations, large amounts of proof effort can be wasted on unsuccessful proofs [35, 21] and consequently the automatic testing approach to gain confidence is still useful.

The longest search time during the evaluation was 23 minutes, hence it can be seen that confidence in the exception freeness of the system (or a counter-example) can be gained very quickly and automatically. This significantly reduces the risks and wasted efforts involved in a proof attempt. Overall, for this slice of the engine controller implementation less than two man-weeks of effort were involved in extracting the verification conditions, simplifying them and then generating test-data. The only remaining task to guarantee exception freeness is to develop formal proofs for the remaining eleven verification conditions.

5.5 Non-Functional

The problem of WCET (worst-case execution time) testing is to find test-data that causes execution of the longest path through the SUT. To allow this to be addressed within the framework, a measure of actual dynamic execution time is used as the fitness function. This removes the stopping criteria present in the other fitness functions, however the search technique has to be designed to stop after exhausting a specified amount of time or when the search is making no progress, as there is no “absolute” stopping criterion.

While testing can never give an absolute guarantee of WCET, it does give a lower bound on the WCET. This information can be combined with static analysis techniques (which give an upper bound) to form a confidence interval. The WCET can be guaranteed to fall within this interval. Testing and/or analysis can continue until this interval is sufficiently small. Figure 16 illustrates this idea. Static analysis techniques can be used to obtain a guaranteed upper bound on the actual WCET. Dynamic testing techniques can be used to obtain (and progressively improve) a guaranteed lower bound on the actual WCET. The resulting interval can then be used to determine whether sufficient static and dynamic analysis has been carried out.

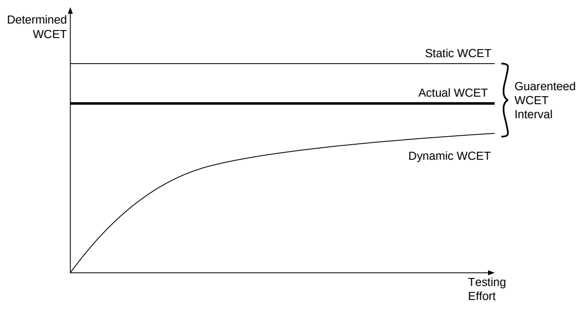


Figure 16: Static and dynamic methods for determining WCET

5.5.1 Evaluation

To evaluate the effectiveness of the framework when applied to WCET testing it has been targeted at finding extreme execution times for the aircraft engine controller code discussed above. The controller code is structured as a cyclic executive with top-level routines from each of the major sub-systems being called repeatedly in a loop. The evaluation involved targeting the framework at four of these top-level routines. The test-data generation system was executed on the target hardware to allow accurate measures of the execution time to be obtained. Figure 10 shows the evaluation results and compares these figures with those obtained using a combined testing and static analysis approach as used in the FADEC development project. This combined approached involved dynamically executing all paths in the software. The longest dynamic path is then fed into a static analysis tool that adds up the worst case instruction times for all instructions on this path.

The intervals between the obtained dynamic and static WCET in these results represents the optimism present in the framework’s dynamic approach and the pessimism in the static analysis model of the processor. The dynamic WCET figures, as already

Top Routine	Dynamic WCET (Cycles)	Static WCET (Cycle)	Interval
Top 1	599	692	15.5%
Top 2	189	225	19.0%
Top 3	7866	10265	30.5%
Top 4	1797	2218	23.4%

Table 10: Engine Controller WCET Results

stated, can only give a lower bound on the execution time. However, this technique can be used earlier in the development life-cycle due to the complete automation and the fact that it does not require all of the code or tests to have been constructed. This technique can be applied as soon as there is code that is able to run, the test-cases used to drive the combined static and dynamic approach are not needed. These early WCET figures obtained can be used to provide confidence that there will not be any serious timing problems. This reduces the risk of finding timing problems right at the end of the development life-cycle when any problems will be extremely expensive to fix.

6 Related Work and Conclusions

6.1 Related Work

Many techniques for automating the generation of test-data have been developed and reported in the literature. Test-data generation techniques can be broadly classified into random, static and dynamic methods. Random test-data generators, as their name implies, simply randomly sample the input domain of the SUT. While this is easy to automate, it is problematic [4] — it can be expensive to calculate the expected outputs for the large amount of test-data generated and its uniform sampling means it can fail to locate the desired test-data for particular criteria. Static techniques do not require the SUT to be executed, but work on an analysis of the SUT. In contrast, dynamic techniques involve the repeated execution of the SUT during a directed search for test-data that meet the desired criterion.

Figure 17 shows the major automated test-data generation approaches presented in the literature [5, 8, 9, 11, 20, 24, 23, 29, 30, 43, 45]. The arrows in the figure indicate the progression of ideas and methods through the various approaches.

Static techniques typically use symbolic execution to obtain constraints on input variables for a particular testing criterion. Symbolic execution works by traversing a control flow graph of the SUT and building up symbolic representations of the internal variables in terms of the input variables. Branches within the code introduce constraints on the variables. Solutions to these constraints represent the desired test-data. There are limitations with this approach. It is difficult to use symbolic execution to analyse recursion, arrays when indices depend on input data and some loop structures using symbolic execution.

The dynamic approach to test-data generation was first suggested in 1976 [29]. This work proposed constructing a “straight-line” version of the SUT and the application of function minimisation techniques. Branches in the SUT are replaced by constraints to create the straight-line version. A function is created to map these constraints on to a real-value. Small values indicate that the constraint is close to being satisfied. Function

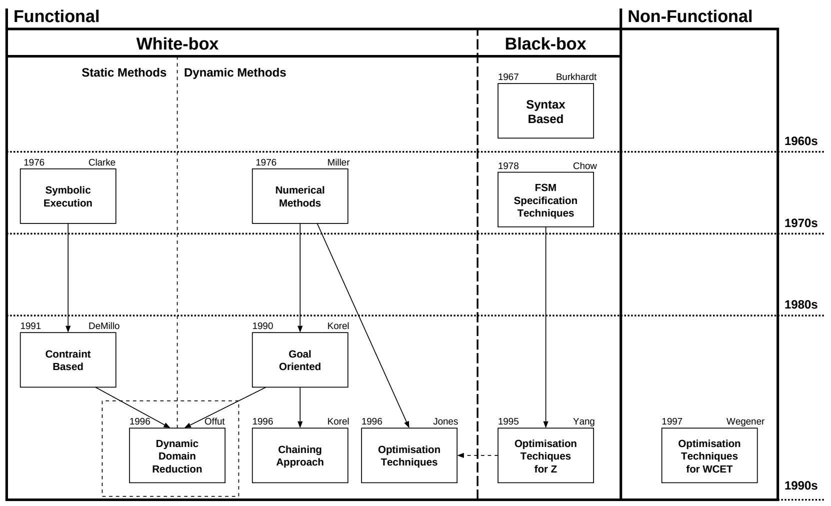


Figure 17: Some Approaches to Automating Test-Case Data Generation

minimisation techniques are used to minimise the values returned from these functions and hence locate test-data. It wasn't until 1990, when Korel [23, 24, 12] built on this, removing the need for the straight-line programs and generating test-data for more data-types. Korel's approach uses a local search technique, which for complex search spaces may get *stuck* in only locally optimal solutions, thus failing to find the desired test-data. As a way forward Korel suggested the use of heuristic global optimisation techniques. Several approaches have explored the use of such search techniques for test-data generation [42, 19, 20, 32].

All of these previous approaches focus on generating test-data in a narrow area. Nearly all focus on functional properties, with most selecting test-data using structural testing criteria. The work presented in this paper is the result of a research programme focused on developing a generalised framework for test-data generation. The aim was to provide a framework capable of automatically generating test-data for both black- and white-box functional criteria and also non-functional properties. The particular focus has been the provision of useful support for the testing problems that are faced when developing high-integrity or safety-critical systems. The generality of the framework comes from the use of directed search techniques and in particular global optimisation techniques.

Despite the suggested use of heuristic approaches to structural test-data generation back in the 1970s [29], work in this area has only recently begun to appear. This section provides a detailed review of the approaches that use global optimisation techniques for structural test-data generation. The test criterion addressed in these approaches is to generate a test-data set that satisfies a specified structural coverage test adequacy criterion.

Xanthakis *et al.* present the first application of heuristic optimisation techniques for test-data generation in the literature (so far as we are aware) [44]. Their approach used random testing to generate test-data to achieve as much coverage as possible of the SUT and then employed a genetic algorithm to fill any gaps. Path traversal conditions for the uncovered parts of the SUT are extracted statically and a simple genetic algorithm implementation is used to find test-data to satisfy these path traversal conditions. The system also exploited variable dependence information extracted from the SUT to guide the manipulation of the test-data during the search.

Building on the work of Xanthakis, Watkins [42] proposed using genetic algorithms again as a mechanism for automatic test-data generation for structural testing. The process works by constructing a control-flow graph for the SUT and extracting all possible paths through the program. A simple genetic algorithm is then applied with the aim of achieving path coverage. This avoids much of the static analysis required in the approach presented by Xanthakis. The fitness function used by Watkins is the inverse of the number of times that a particular path has previously been tested. Thus a path that has been visited often is assigned a lower fitness value to encourage execution of alternative paths. Watkins evaluated the techniques against random test-data generation using the number of unique tests generated before achieving full coverage, as the evaluation criterion. Two versions of a triangle classification program are used in the evaluation (with and without a right-angled triangle classification check). The results show genetic algorithms to require an order of magnitude fewer tests to achieve path coverage of the simple triangle classifier and 50% fewer tests to cover the more complex triangle classifier when compared with random testing. However, it is unclear whether the fitness function would truly provide enough guidance to the genetic algorithm for any non-trivial program. The path taken through the SUT is governed by the branch predicates. Hence, these branch predicates contain a lot of information that could be exploited in the search for test-data. The simple fitness function used in this work disregards all this information. The evaluation of the work only shows test-data generation for two simple programs — these contain no loops, function calls, or arrays. For such simple programs, static approaches using symbolic execution and constraint solving would be highly effective. There is no detailed presentation of the genetic algorithm implementation in Watkins' work. No discussion of encoding scheme, genetic operators or search parameters is presented. Consequently, it is not possible to draw any conclusions about the decisions taken.

Yang [45] presents an approach to generating test-data directly from a Z specification using genetic algorithms. From the declaration part of the Z, the type and domain of each variable are derived. The expressions in the Z are considered as a "flow of execution" through the specification. Conjunction forms sequences and disjunction branches. Using this information, a tree of routes (or paths) describing the possible execution flow through the specification is constructed. The genetic algorithm is then used to search for test data that satisfies each route through the expressions. The fitness is calculated in one of two ways - Hamming distance[12] and comparison. For expressions $f \Re g$, where $\Re$ is a relational operator, the fitness is calculated as follows. For the equality operator, the Hamming distance between f and g is used, with high values indicating a low fitness. For inequality operators the Hamming distance between f and $op(g)$ is used, where op is either a successor or predecessor depending on the relational operator. For example, for $f > g$ the fitness will be measured using the Hamming distance between f and $successor(g)$, this gives the search a bias towards the boundary values. Again, high Hamming distance values indicate a low fitness.

The comparison method exploits information contained in the Z expressions. The calculation returns a measure of the relative *closeness to being true* for the expression. This is scaled to values between 0.0 and 100.0. If the expression is actually true then

[12]The Hamming distance between two values, $d(u, v)$ is equal to the number of bit positions in which the bits of u and v differ.

a fitness value of 100.0 is always returned. This feature provides a highly efficient stopping criterion for the search process, i.e. the search can be terminated whenever a fitness of 100.0 is located.

The results published in Yang's work were limited to two small Z specifications for the triangle classification problem and a quadratic equation solver. Only a small subset of Z is used in these specifications, with all predicates being simple relational expressions. The work gives no indication of how test-data might be generated for larger Z specifications, especially if more complex predicate expressions are used.

Jones *et al.* have published a series of papers [19, 18, 20] extending Yang's work and allowing genetic algorithms to be applied to structural testing of software. In particular, branch coverage of Ada programs is considered. In this work, the information contained in the branch predicates is exploited in the calculation of test-data fitness. The approach involves executing an instrumented version of the SUT. The instrumentation is added in order to dynamically calculate the fitness value during program execution for the given test-data. The instrumentation inserts function calls to three types of functions.

`CHECKING_BRANCH` functions are added at the start of each basic-block in the SUT. These functions register when each basic-block has been executed.

`LOOKING_BRANCH` functions are added following each conditional branch in the SUT. These functions exploit the information contained in the branch predicate and calculate the fitness value of the current test-data. They function as follows:

- If the current test-data has executed the desired branch, the fitness is set to a high value.
- If a sibling node was required to be executed (i.e. the current test-data has taken an undesired branch) the fitness is related to the branch predicate. Two different fitness functions were evaluated in this work. The first calculated the fitness according to the Hamming distance. For example, if the branch predicate is $X \leq 0$, the fitness would be set according to the Hamming distance between X and succ(0). The second used the reciprocal of the numerical difference between the operands of the relational expression. For example, again using the branch predicate of $X \leq 0$, the fitness would be calculated as the reciprocal of the distance between X and succ(0). The aim is to guide the search to test-data where the sibling node will be executed, i.e. where $X > 0$. The $succ(0)$ is used to bias the test-data generation to be as close as possible to the sub-domain boundary.
- If neither of the above is relevant, the fitness is set to a low value.

`LOOKING_BRANCH_LOOP` functions are added following loop constructs. The fitness in this case is related to the difference between the actual and desired number of iterations of the loop. Additional loop counter variables are added to the SUT to monitor the loop iterations.

This fitness function gives significantly more guidance than that proposed by Watkins. Given the simple program shown in figure 18, assume that the false branch has already been executed once and the search is attempting to locate test-data for the true branch.

Tabel 11 shows the fitness values for various test-data using Watkins' fitness function and the reciprocal of the numerical difference fitness function used by Jones.

This clearly shows the additional guidance provided by Jones' fitness function. As the value of X gets *closer* to causing the branch predicate to be true, the fitness

```
function F (X, Y : Integer) is
begin
 if X > 0 then
    return Y / X;
 else
    return Y;
 end if;
end F;
```

Figure 18: Simple program

X	Y	Watkins	Jones
-100	1	1	0.0099
-10	1	1	0.0909
0	1	1	1
1	1	∞	∞

Table 11: Comparision of Watkins' and Jones' fitness functions

value increases. This helps to guide the search to such test-data, whereas with the simpler fitness function proposed by Watkins, the fitness value remains unchanged until the branch predicate evaluates to true. Consequently, the fitness function provides no guidance on how to get to such test-data.

Jones' work also considers a number of implementation aspects of the genetic algorithms themselves. These were not discussed in detail by Watkins or Yang. Gray code, sign and magnitude and twos complement binary encodings of the test-data are evaluated. The results show Gray code to give better guidance, especially when the test-data is dominated by integer variables. This is due to the fact that two values close in the integer domain can have very different binary representations in twos complement and sign and magnitude representations making it difficult for the genetic algorithm to move between them. In contrast, successive integer values only differ by the negation of a single bit when using a Gray code representation.

An evaluation is presented in Jones' work using a collection of small programs — a triangle classifier, a remainder calculator, a linear search, a binary search, a quick sort and a quadratic equation solver. Branch coverage is achieved for all of these programs with significantly fewer tests than required by purely random testing (up to two orders of magnitude).

Although the work of Jones *et. al* advances that of both Xanthakis and Watkins by exploiting information in the branch predicates, there are still limitations and weaknesses. These are as follows:

1. The fitness function fails to recognise the successful outcome of conditional branches prior to an undesired branch. For example, consider the path $P = \langle i, j, k, l \rangle$. Test-data that follows a sub-path $\langle i, j, k \rangle$ before taking an undesired branch is *better* than test-data that follows a sub-path $\langle i \rangle$ before taking an undesired branch. However, this is not captured by the fitness function which only exploits the information available at branch predicates that fail.

2. The calculation of the fitness function for loop iterations is only based on the desired and actual number of iterations. More guidance could be gained by incorporating knowledge of the loop-predicate itself, just as using information from the branch predicates improves the level of guidance.

3. The fitness function as presented only works with simple branch predicates of the form $X \Re Y$, where $\Re$ is a relational operator. More complex expressions have to be broken down into nested sequences of condition branch statements.

4. Binary encoding of the data-types can lead to invalid bit patterns being produced by the crossover and mutation operators. For example, an integer with a legal range from 0 to 100 might be represented in 8 bits and hence crossover and mutation can easily cause bit-patterns that are outside of the valid range.

5. In addition to the invalid bit patterns, there is also the problem of information loss when applying the genetic operators to the binary strings.

In a recent paper by Pargas *et al.* [32] an approach is presented that addresses the first two of these problems. Again, the work targets the generation of test-data for structural testing criteria. A control dependence graph for the SUT is utilised in the calculation of test-data fitness. In a control-dependence graph nodes represent basic-blocks and edges represent control dependencies between basic-blocks. A node Y is control dependent on a node X (where X contains a conditional branch) if the reachability of node Y is dependent on the conditional branch in X. Figure 19 shows an example program with its control-flow graph and control dependence graph.

```
       procedure Example is
          i, j, k : Integer
       begin
1.        Read (i, j, k)
2.        if (i < j) then
3.           if (j < k) then
4.              i := k;
             else
5.              k := i;
             end if;
          end if;

6.        Process (i, j, k);
       end Example;
```

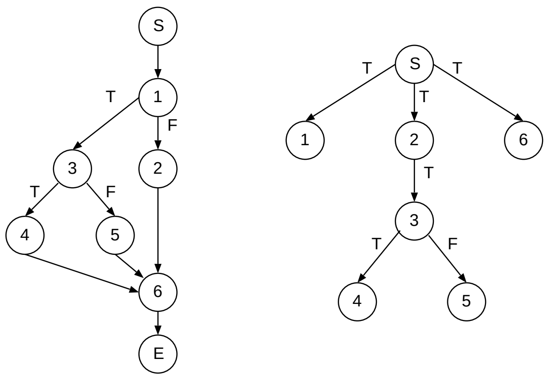


Figure 19: Example program with associated control-flow and control-dependence graphs

The fitness of test-data is evaluated by comparing the statements executed using the test-data to the desired path through the control-dependence graph. For example, consider the problem of locating test-data to execute statement 5 in the example program. The control-dependence graph indicates that the path $\langle S(T), 2(T), 3(F)\rangle$ needs to be executed. Test-data is assigned a fitness value according to how much of this path it executes. For example, test-data that executes the path $\langle S(T), 2(T), 3(T)\rangle$ will be assigned a higher fitness than test-data that executes the path $\langle S(T), 2(F)\rangle$.

This approach addresses the problems 1, 2 and 3 above. Problems 1 and 2 are overcome by exploiting the information contained in the control-dependence graph and by relating fitness to the number of successful branch executions. Problem 3 is addressed by not explicitly considering the structure of the branch predicates. However, as with Watkins work, this failure to exploit the information contained in branch predicates results in insufficient guidance for the search technique to locate the desired test-data for all but the simplest of programs. Problems 4 and 5 are not addressed in Pargas' work and indeed the genetic algorithm implementation presented continues to suffer from these problems.

6.2 Conclusions

Many of the approaches for automated software test-data generation presented in the literature are inflexible or have limited capacity. In constrast, optimisation techniques offer a flexible and efficient approach to solving complex problems. To allow the optimisation based framework to generate test-data for a specific testing criterion it is necessary only to devise a suitable fitness function. This paper has presented the results of three industrial scale case-studies — specification failure testing, exception condition testing and WCET testing.

The results presented in this paper are extremely encouraging. They show that the search-based framework for test-data generation has been able to automatically generate useful test-data for real industrial testing problems using real industrial case-studies. Through the complete automation the aim is to allow this targetted testing for errors to be applied to build confidence in the desired properties of the software. It is hoped that this will ultimately reduce the costs associated with traditional testing and proof techniques by reducing the number of errors they find.

An important fact is that the framework tools provided to support this automated test-data generation need not be of high-integrity even when testing safety critical code. They can be viewed as simply generating test-data that can be checked by other means, i.e. use of a suitable test-harness to check that the generated test-data does in fact cause the desired exception. This is important as the algorithms are stochastic and it is extremely difficult to reason about their efficacy for application to arbitrary code. Also many of the test criteria require the execution of an instrumented version of the SUT. The validity of testing the instrumented code is problematic for safety-critical systems. Again, the generated test-data can be checked by trusted means to avoid this problem.

6.3 Further Work

Optimisation techniques will never be able to guarantee their results[13]. However, it may be possible to devise software metrics which can give guidance in a number of areas – to suggest which optimisation techniques will give the best results; to suggest suitable parameter values for the optimisation techniques; and also to give an indication as to the likely quality of the result. There remains a large number of test-data selection criteria worthy of consideration. For each criterion it is only necessary to devise an fitness function which gives the search process sufficient guidance. This allows virtually any testing criteria to be incorporated into the same generalised framework, although only time will tell if the techniques we have described here are always so effective.

[13]For example, if optimisation fails to find test-data illustrating a specification failure that does no imply that the software must be correct, it simply means that the search failed to find any such failures.

7 Acknowledgements

This work was funded by grant GR/L4872 from the Engineering and Physical Sciences Research Council (EPSRC) in the UK as part of the Software Engineering for Business Process Change programme. It builds on work started under the Realising Our Potential Award grant GR/K63702 also from the EPSRC.

The authors would like to thank Rolls-Royce Plc. for their support and provision of access to material enabling case-studies to be carried out on real industrial systems.

References

[1] D. J. Andrews (ed.). *Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base Language*. ISO/IEC 13817-1, December 1996.

[2] ANSI/MIL-STD 1815A. *Reference manual for the Ada programming language*, 1983.

[3] J. Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.

[4] B. Beizer. *Software Testing Techniques*. Thomson Computer Press, 2nd edition, 1990.

[5] W. Burkhardt. Generating programs from syntax. *Computing*, 2(1):83–94, 1967.

[6] A. Burns and A. Wellings. *Real-Time Systems and Their Programming Languages*. International Computer Science Series. Addison Wesley, 1990.

[7] R. A. Caruana and J. D. Schaffer. Representation and hidden bias: Gray vs. binary coding for genetic algorithms. In *Proceedings of the Fifth Internationl Conference on Machine Learning*. Morgan Kaufmann, 1988.

[8] T. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4:178–187, May 1978.

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

[10] L. D. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reihold, 1991.

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

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

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

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

[15] A. C. T. Inc. The gnat ada-95 compiler, 1997. http://www.gnat.com/.

[16] ISO/IEC 8652:1995. *Ada 95 : Language Reference Manual*, 1995.

[17] C. Z. Janikow and Z. Michalewiz. An experimental comparison of binary and floating point representations in genetic algorithms. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1991.

[18] B. Jones, H. Sthamer, and D. Eyres. The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of 3rd International Conference on Software Quality Management*, volume 2, pages 435–444, 1995.

[19] B. Jones, H. Sthamer, and D. Eyres. Generating test-data for Ada procedures using genetic algorithms. In *Genetic Algorithms in Engineering Systems: Innovations and Applications*, pages 65–70. IEEE, September 1995.

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

[21] S. King, J. Hammond, R. Chapman, and A. Pryor. The value of verification: Positive experience of industrial proof. In *Formal Methods 1999 Technical Symposium*, 1999.

[22] S. Kirkpatrick, J. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

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

[24] B. Korel. Automated test data generation for programs with procedures. In *International Symposium on Software Testing and Analysis*, pages 209–215. ACM/SIGSOFT, 1996.

[25] N. G. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, 1995.

[26] N. G. Leveson and P. R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, SE-9(5):569–579, 1983.

[27] J. L. Lions. Ariane 5: Flight 501 failure report. Technical report, ESA/CNES, July 1996.

[28] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1996.

[29] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, September 1976.

[30] A. J. Offutt and J. Pan. The dynamic domain reduction procedure for test data generation.
http://www.isse.gmu.edu/faculty/ofut/rsrch/atdg.html, 1996.

[31] M. Ould. Testing - a challenge to method and tool developers. *Software Engineering Journal*, 6(2):59–64, March 1991.

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

[33] Praxis Critical Systems. *Spark-Ada Documentation 2.0*, 1995.

[34] C. R. Reeves. *Modern Heuristic Search Methods*, chapter 1 – Modern Heuristic Techniques, pages 1–25. Wiley, 1996.

[35] P. C. S. Rod Chapman. Private communication, 1997.

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

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

[38] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE, October 1998.

[39] N. Tracey, J. Clark, J. McDermid, and K. Mander. Integrating safety analysis with automatic test-data generation for software safety verification. In *Proceedings of the 17th International Conference on System Safety*, pages 128–137. System Safety Society, August 1999.

[40] N. Tracey, A. Stephenson, J. Clark, and J. McDermid. A safet change oriented process for safety-critical systems. In *In the Proceedings of Software Change and Evolution Workshop. International Conference on Software Engineering*. IEEE, May 1999.

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

[42] A. L. Watkins. The automatic generation of test data using genetic algorithms. *Proceedings of the 4th Software Quality Conference*, 2:300–309, 1995.

[43] J. Wegener, H. H. Sthamer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

[44] S. Xanthakis, C. Ellis, C. Skourlas, A. L. Gall, S. Katsikas, and K. Karapoulios. Application des algorithmes genetiques au test des logiciels. In *Proceedings of 5th International Conference on Software Engineering*, pages 625–638, 1992.

[45] X. Yang. The automatic generation of software test data from Z specifications. Technical report, Department of Computer Studies, University of Glamorgan, 1995.