

Integrating Automated Testing with Exception Freeness Proofs for Safety Critical Systems

Nigel Tracey John Clark Keith Mander John McDermid

Department of Computer Science.

University of York,

Heslington, York.

YO10 5DD,

England.

+44 1904 432749

{njt, jac, mander, jam}@cs.york.ac.uk

Abstract

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. This paper presents a technique for automatically generating test-data to test exceptions. The approach is based on the application of a dynamic global optimisation based search for the required test-data. The authors' work has focused on test-data generation for safety-critical systems. Such systems must be free from anomalous and uncontrolled behaviour. Typically, it is easier to prove the absence of any exceptions than it is to prove that the exception handling is safe. A process for integrating automated testing with exception freeness proofs is presented as a way forward for tackling the special needs of safety critical systems. An evaluation shows the application of the technique to a commercial aircraft engine controller system as part of a proof of exception freeness.

1 Introduction

A failure occurs when software is prevented from performing its intended action. A subclass of failures, which are known as exceptions [11], may be due to erroneous inputs, hardware faults or logical errors in the software code. Since by definition exceptions are expected to occur rarely, the exception handling code of a system is, in general, the least documented, tested and understood part. In a recent incident, the Ariane 5 launch vehicle was lost due to an mis-handled exception destroying \$400 million of scientific payload [19].

Safety-critical systems present special problems. Typically, it is easier to prove the absence of any exception conditions than it is to prove that the exception handling is safe. Our major concern is the development of software for safety critical systems. This type of development is often carried out using a 'safe' subset of a language. This allows the application of static analysis and the potential generation of proofs to show adequate system safety. We have therefore focused on the SPARK-Ada language [2, 22]. SPARK-Ada is a widely used Ada subset for the implementation of safety-critical systems.

This paper presents an approach for automatically generating test-data to support a proof of exception freeness. The aim is to reduce the effort and costs associated with unsuccessful proof attempts due to software bugs. Section 2 briefly surveys approaches for automatic test-data generation from the literature. Section 3 introduces the test-data generation problem with regards to testing exceptions. Section 4 then discusses the specific requirements of safety critical systems. The technique presented is a further application of the authors' automated test-data generation framework. This framework is based on the application of heuristic global optimisation techniques. 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 which they are attempting to solve. It is this property that allows a general test-data generation framework to be developed for solving a num-

ber of testing problems. Optimisation techniques are simply directed search methods that aim to find optimal values of a particular objective function (also known as a cost or fitness function). Section 5 introduces a optimisation technique (genetic algorithms) and then describes in detail how the framework has been applied to the testing of exceptions. Section 6 presents an evaluation of the test-data generation approach. This evaluation shows the application of the automatic test-data generation technique as part of a proof of exception freeness. This has been carried out using the software for a civil aircraft engine controller with the help of Rolls-Royce Plc.

2 Automatic test-data generation

Automated test-data generators can be divided into three classes – random, static and dynamic. Random test-data generation is easy to automate, but problematic [3]. First, it produces a statistically insignificant sample of the possible paths through the software under test (SUT). Second, it may be expensive to generate the expected output-data for the large amount of input data produced. Finally, given that exceptions occur only rarely, the input domain which causes an exception is likely to be small. Random test-data generators may never hit on this small area of the input domain.

Static approaches to test-data generation generally use symbolic execution. Many test-data generation approaches presented in the literature use symbolic execution to obtain structural test-data [5, 6, 14, 23]. 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, for the desired path. Branches within the code introduce constraints on the variables. Solutions to these constraints represents the desired test-data. A number of problems exist with this approach. Using symbolic execution it is difficult to analyse recursion, array indices which depend on input data and some loop structures. Also, the problem of solving arbitrary constraints is known to be undecidable.

Dynamic test-data generation involves execution of the SUT and a directed search for test-data that meets the desired criterion. The dynamic approach was first suggested in 1976 by Miller [20]. The work of Korel *et al.* built on this using locally directed search techniques [7, 17, 18]. This is further expanded by Gallagher *et al.* [9]. Local search techniques only work effectively for linear continuous functions. Consequently, these techniques are likely to become stuck at a local optimum and fail to locate the required global optimum [13]. The use of global optimisation

techniques for dynamic test-data generation has been investigated more recently [13, 30] in an attempt to overcome this limitation.

The authors' have built on this work to develop an extensible test-data generation framework based on global optimisation techniques. It has already been used to generate test-data for a wide variety of testing problems — constraint-solving [4], specification-based functional testing [25], worst-case execution time testing [26], structural testing [27] and safety-related testing [28]. This paper discusses an extension allowing its application to test-data generation for exceptions. The following section introduces the specific problems involved in testing exceptions and their handlers.

3 The test-data generation problem

Predefined exceptions are raised when the language rules are violated at run-time and in response to hardware errors. Test-data alone cannot test the raising of exceptions in response to hardware errors. For these hardware errors integration with a fault injection technique [29] is required. The focus, in this paper, is the generation of test-data which violates run-time language rules. For Ada there are a number of predefined exceptions, these are as follows [1]:

`Constraint_Error` – data going out of range.

`Program_Error` – control-structure violation.

`Storage_Error` – running out of storage space.

`Tasking_Error` – task communications failure

SPARK-Ada allows testing for `Constraint_Error` exceptions to be the focus of the work to date. The SPARK-Ada tool-set [22] mitigates against other predefined exceptions through language restrictions or static analysis. `Tasking_Error` cannot occur as Ada tasking is not part of SPARK-Ada. `Storage_Error` is also unlikely to occur as dynamic memory allocation is not being used and therefore storage requirements can be calculated statically. The situations where `Program_Error` exceptions can occur are detected by the SPARK Examiner static analysis tool.

`Constraint_Error` exceptions occur when a value goes out of range; examples include type bounds, array bounds, arithmetic underflow/overflow and attempts to divide by zero. The Ada compiler is responsible for inserting checks into the object-code to check for `Constraint_Errors` at all points where such errors may occur. Figure 1 shows how such compiler checks might be inserted. This example just illustrates the idea of the compiler generated checks. The Ada code

containing the checks is not semantically correct. Obviously, the code as presented has the potential to raise an exception as the `if` condition is evaluated. The compiler inserted checks must be coded so this cannot happen. The test-data generation problem is to find, for each point where a `Constraint_Error` is possible, data that causes the run-time violation.

```

subtype Idx is Integer range 1 .. 1000;
A : array (Idx) of Integer;
B,C : Idx;
D : Integer;

-- Original Code
A (D) := B;
D := (C * C) + B;

-- Code with compiler checks
if D not in Idx or B not in Integer then -- BB 1
  raise Constraint_Error;                -- BB 2
end if;
A (D) := B;                               -- BB 3
if (C * C) + B not in Integer then      -- BB 3
  raise Constraint_Error;                -- BB 4
end if;
D := (C * C) + B;                          -- BB 5

```

Figure 1: Predefined Exceptions Checks

4 Freedom from exceptions

Many systems, including aviation, commerce, and medical systems, depend upon and require the correct functioning of software to perform their desired task. With such critical systems the probability of software failure must be reduced to acceptable levels.

An important aspect in developing high-integrity and safety-critical systems is that of certification. Certification typically includes the process of independently verifying conformance to a standard. Examples of relevant standards for safety-critical software include defence standard 00-55 [21], civil aviation standard DO178B [8] and generic standard IEC-61508 [12]. Typically such standards will require full test coverage of the object code (for example the requirement for full modified-condition decision coverage in DO178B). This task is made very much more difficult when the compiler inserts run-time checks. This is illustrated in figure 1, whereas the original program appeared to have a single basic block¹ with compiler checks there are actually five. However, if it can be proved that

¹A basic block is a sequence of instructions such that if any one of them is executed then all are executed

such checks never fail (i.e. exceptions can never occur) then they can be omitted. Indeed it is usually easier to reason that a program is free from exceptions than to reason about the correctness of exception handling that might be used in full Ada [2].

SPARK-Ada and its associated tools [2, 22] (SPARK examiner, SPADE automatic simplifier and proof checker) allow such proofs of exception freeness to be performed. Using the SPARK Examiner, appropriate check annotations can be generated that represent the necessary run-time checks.

As well as generating these check annotations the SPARK examiner can generate verification conditions for each of the checks. If these conditions can be discharged then that proves that the run-time exception associated with the annotation cannot occur. For example, consider the code `A(D) := B;` from figure 1. The SPARK examiner automatically inserts the check annotation associated with the required run-time checks, `--# check B in Integer`. From this the verification conditions shown in figure 2 are generated.

```

H1:    true .
H2:    b >= index__first .
H3:    b <= index__last .
      ->
C1:    b >= integer__first .
C2:    b <= integer__last .

```

Figure 2: Verification Conditions

This shows three hypotheses, which can be assumed to be true, and two conclusions. To prove that the corresponding exception cannot occur a proof is required to show that the conclusions logically follow from the hypotheses. In this example it is simple to see that this is so.

In many cases the SPADE automatic simplifier will take the verification conditions associated with run-time checks and simplify them to true (or possibly false). If the simplifier cannot discharge (prove) the conditions, a guided proof is required. Such a proof, whether manual or semi-automatic, can be very effort intensive requiring highly skilled engineers. Before any such effort is invested a good deal of confidence in the successful outcome is desirable to reduce the risk. If the putative properties being proved are simply untrue then attempting a proof (which will inevitably fail) is an extraordinarily expensive method to find errors in the software.

To achieve the desired confidence one approach is

aggressively to test for exception conditions. This is equivalent to finding a counter-example for the proof. This allows the test-data generation for exception conditions to be integrated into a process to show exception freeness for safety-critical software and so reduce the costs involved in proving exception freeness. The process for integration is discussed in detail in section 6. Typically where exception freeness is not proved, special arithmetic operators are used to protect against overflow, underflow and division-by-zero usually by saturating the result. This in turn can require additional complexity in the control code as it must remain stable even when results are saturated. By proving exceptions cannot occur, the additional complexity and the need for special operators is removed. A reduction in the costs involved in the proof step make it more likely to be used on *real* commercial projects.

5 Optimisation based test-data generation

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 which 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 objective function (also known as a cost or fitness function). In this work both simulated annealing [16] and genetic algorithms [10] have been used as the optimisation technique. Both have performed effectively, although this paper highlights the use of genetic algorithms.

Genetic algorithms were developed initially by Holland *et al.* in the 1960s and 1970s [10]. 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 and mutation introduces new characteristics into the solutions. A population of sample solutions is maintained each of whose *fitness* (or cost) 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 coverge towards a global solution.

For a dynamic search to succeed, it needs to be given some guidance. This guidance is given in the form of a cost-function which relates a program input to a measure of how *good* it is, with respect to the

objective of executing a given part of the SUT. The amount of guidance given by the cost-function is one of the key elements in determining the effectiveness of the test-data generation. The other key factor is, of course, the search technique itself.

To generate test-data to raise an exception, the cost-function needs to indicate whether the desired exception has been raised. However particular test-data will either satisfy this criterion or not, this alone provides insufficient guidance to the search. The cost-function needs to return small values for test-data that *nearly* raises the exception and larger values for test-data that is *far* from raising the exception. The guidance is effectively twofold. First, the search must be guided to test-data that executes the statement that may cause an exception. Second, test-data is required that violates the run-time language rules at this point. Since branch predicates determine the path followed they are therefore vital in determining an effective cost-function.

Branch predicates consist of relational expressions connected with logical operators. The cost-function is designed such that it will evaluate to zero if the branch predicate evaluates to the desired condition and will be positive otherwise. This is important as it gives a highly efficient stopping criterion for the search process, i.e. the search stops once the cost-function is zero. The cost-function is calculated as shown in table 1 below. In the table, K represents a failure constant which is added to the cost value to further punish incorrect test-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 $(b - a) + K$
$a \vee b$	$\min(\text{cost}(a), \text{cost}(b))$
$a \wedge b$	$\text{cost}(a) + \text{cost}(b)$
$\neg a$	Negation is propagated over a

Table 1: Cost-Function Calculation

In order to evaluate the cost-function it is necessary to execute an instrumented version of the SUT. The instrumented SUT contains procedure calls which monitor the execution path taken and the evaluation of the branch predicates. There are two types of procedure call that need to be added by an instrumenter – branch evaluation calls and exception monitoring calls. The branch evaluation calls replace the branch predicates in the SUT. These functions (`Branch_1` in figure 3(b)) are responsible for guiding the search to the desired statement. They function as follows.

- If the target node (basic-block containing the exception to be raised) is only reachable if the branch predicate is true, then add the cost of (*branch predicate*) to the overall cost for the current test-data.
- If the target node is only reachable if the branch predicate is false, then add the cost of $\neg(\textit{branch predicate})$ to the overall cost for the current test-data.
- If the current test-data causes an undesired branch to be taken (i.e. the true branch when the target node is on the false branch) then terminate the execution of the SUT and return the cost to the search procedure. This improves the performance of generating the desired test-data.
- Within loops, adding the cost 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.

In this mode any branch predicate outcomes that do not directly cause or prevent the exception being raised add nothing to the cost. Any test-data which causes the desired exception, by any path, is acceptable. However, it is also possible for the user to specify the complete path to the exception that is desired. The search technique is then further constrained to find test-data which both follows the desired path and raises the exception.

The exception monitoring calls simulate the expansion of the run-time checks inserted by the compiler. These instrumentation calls (`Excep_1` to `Excep_7` in figure 3(b)) encode the condition required to cause a run-time exception, they function as follows.

- If the exception is required to be raised then the cost of the condition required to cause the excep-

tion is added to the overall cost for the current test-data.

- If the exception is required not to be raised then the cost of the \neg (condition to cause the exception) 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 to be terminated and the cost returned to the search procedure.
- Again, as for branch evaluation calls, adding the cost of exception conditions is deferred until exit from the loop.

It is important to note that these exception monitoring calls may actually raise exceptions themselves. For example, in the evaluation of $(C \times C) + B$ not in Integer from figure 1 a `Constraint_Error` might be raised. The code is written so that if this occurs then a zero cost is returned as the current test-data causes the desired exception.

The entire process is supported by a prototype tool-set which supports the testing of Ada programs. This tool-set extracts the required information from the SUT and then generates a custom implementation of the optimisation system. It is the execution of this system which dynamically executes the SUT, calculates the cost values and ultimately generates the required test-data.

5.1 Example

To demonstrate how this works in practice, consider the problem of generating test-data which causes a divide-by-zero error (i.e. raises a `Constraint_Error`) in the program in figure 3(a). The following illustrates how the cost-function works. For the example the population size is set at five, `Small_Int` integers are assumed to be in the range -5 to 5 and K (the punishment constant) is set to 10.

The following shows the initial random population of solutions and their associated fitness values calculated using the cost function.

No.	X	Y	Cost	Description
1	-3	4	3 + K	Fails at <code>Branch_1</code>
2	2	-5	32 + K	Fails at <code>Excep_6</code>
3	-5	-1	5 + K	Fails at <code>Branch_1</code>
4	1	-5	45 + K	Fails at <code>Excep_6</code>
5	3	2	0 + K	Fails at <code>Branch_2</code>
Ave. Cost			27	

```

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;

```

(a) Original Program

```

function F (X, Y : Small_Int) return Integer is
  Z : Integer;
begin
  if Branch_1 (X) then
    raise Invalid_Data;
  end if;
  Excep_1 (X, Y); -- (X + Y) in Integer;
  Z := X + Y;
  if Branch_2 (Z) then
    return Integer'Last;
  else
    Excep_2 (X); -- (X ** 4) in Integer;
    Excep_3 (Z); -- (Z - 1) in Integer;
    Excep_4 (Z); -- (Z - 5) in Integer;
    Excep_5 (Z); -- (Z - 1) * (Z - 5) in Integer;
    Excep_6 (Z); -- ((Z - 1) * (Z - 5)) << 0
    return ((X ** 4) / ((Z - 1) * (Z - 5)));
  end if;
end F;

```

(b) Instrumented Program

Figure 3: Example program and instrumentation

At this point the prospective parents are selected and the offspring population formed by applying the crossover (the values of X and Y from either parent are randomly selected to form the offspring) and mutation operators (introduces completely new random values). Crossover combines attributes of two parents to form new solutions called offspring in an attempt to combine the best features of the parents. In this example the offspring is formed by which parents X and Y values will be present in the offspring. Mutation is responsible for introducing new solutions into the search. Consequently the mutation operator will mutate the offspring's X and/or Y values to new random values with a specified probability. Applying these genetic operators gives the following offspring.

No.	X	Y	Cost	Description
6	-3	-1	3 + K	Crossover 1, 3
7	3	-5	21 + K	Crossover 3, 4, Mutate X
8	1	-5	45 + K	Crossover 2, 4
9	2	0	1 + K	Crossover 2, 5, Mutate Y
10	2	3	0 + K	Crossover 2, 5 Mutate Y
Ave. Cost			24	

The next generation is then selected. In this example a hybrid of elite survival and random selection is used. Elite survival selects the three best solutions, these are then combined with two random solutions

to form the next generation of solutions. This new population is as follows.

No.	X	Y	Cost	Description
11	3	2	0 + K	Elite survival of 5
12	2	0	1 + K	Elite survival of 9
13	2	3	0 + K	Elite survival of 10
14	1	-5	45 + K	Random survival of 4
15	-3	-1	3 + K	Random survival of 6
Ave. Cost			19.8	

It can be seen that the average population cost-value in this new generation has fallen, indicating that the population is getting fitter. The process then repeats generating new offspring by selecting parents from this new population. This gives the following offspring.

No.	X	Y	Cost	Description
16	3	3	5 + K	Crossover 11, 13
17	2	-1	0	Crossover 12, 15
18	-3	3	3 + K	Crossover 13, 14
19	-1	2	1 + K	Crossover 11, 15, Mutate X
20	2	-1	0	Crossover 13, 15
Ave. Cost			7.8	

At this point the search can stop as solutions 17 and 20 both have a cost of zero and hence represent test-data that will raise the desired exception. Figure 4 shows the cost surface for this divide-by-

zero exception. There are 121 possible test inputs to this simple program, of these it can be seen only 6 raise the divide-by-zero exception. These can be seen as the points with a zero cost in figure 4 — $(0, 1)$, $(1, 0)$, $(2, -1)$, $(3, -2)$, $(4, -3)$ and $(5, -4)$.

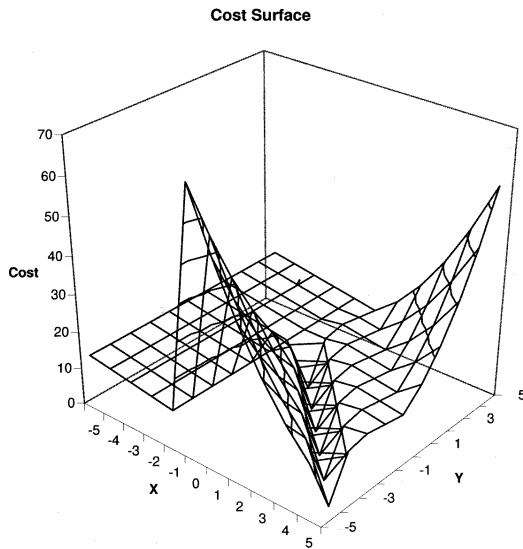


Figure 4: Cost service for divide-by-zero exception

As can be seen from this example that the cost-function gives a quantitative measure of the suitability of the generated test-data for the purpose of raising a specified exception in the SUT.

6 Evaluation

The evaluation uses the code for a civil aircraft engine controller. The software is approximately 200,000 lines of safety-critical code written in SPARK-Ada. The code encompasses a number of different types of functionality — state-based, control-laws, redundancy provision, background maintenance and health monitoring. In the final production version of the engine controller code the run-time checks are disabled. It is therefore important that there is no possibility of a run-time exception. As the code is written in SPARK-Ada only `Constraint_Errors` need to be considered. Figure 5 shows a safety-critical routine from the engine controller code.

The process for integrating automatic testing and proof for exception freeness consists of the following four steps:

Step 1: Use the SPARK-Examiner to extract the verification conditions for exception freeness. Discharging these conditions guarantees the exception condition cannot occur.

```

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 5: Smooth Signal Subprogram

Step 2: Use the SPADE-Automatic Simplifier to discharge as many of the obligations automatically as possible.

Step 3: Target testing at remaining obligations in an attempt to locate examples that show a proof will fail.

Step 4: Prove remaining conditions.

The evaluation follows this process. First, the SPARK-Examiner was used to extract the verification conditions for exception freeness from the engine controller code. The SPARK automatic simplifier was able to discharge 89% of these verification conditions completely automatically. For the example given in

figure 5 the Examiner generates 13 verification conditions, of which 11 are discharged automatically by the simplifier.

Approximately half of the remaining verification conditions simply required information about compiler-dependent type ranges. The simplifier was then also able to discharge these automatically. The smooth signal subprogram does not contain any compiler-dependent types, therefore no additional verifications conditions were discharged. The test-data generation tool set was targeted at the remaining verification conditions. The aim was to generate test-data which illustrated a condition under which the exception would be raised. Where the test-data generation was successful the test-data illustrated a condition under which the run-time rules of the Ada language would be violated and hence an exception raised. Figure 6 shows the remaining two verification conditions for the smooth signal subprogram (the irrelevant hypotheses have been removed).

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

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

Figure 6: Verification Conditions for Smooth Signal

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.

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. 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 prevents a large 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 also prevented a number of exception conditions occurring in practice. This can be seen in the smooth signal example, the sensors reading `currentval` and `goodval` can only give values such that $currentval - goodval$ is always in range. The overflow of integer counters was another potential cause of exceptions. On closer inspection none of these overflow conditions could arise. Typically the counters would be reset after a number of iterations according to fixed global data stored as part of the engine configuration. In all cases this global data was passed in by the calling environment and contained values such that the counter could never overflow. This was again true in the smooth signal example, here the global configuration ensures the counter was reset after at most 9 iterations (i.e. `SmoothSignal` is never called with `CountThresh` above 9). Again, the test-data generated here 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 which the test-data generation was unsuccessful, proofs were attempted. In all cases these were successful in discharging the verification conditions.

The lack of inter-procedural information allows test-data to be generated that, in practice, the system would never see. For example, as discussed above the `SmoothSignal` is never called with a `CountThresh` greater than 9. One possible approach to address this problem is to use the SPARK-Ada pre and post-condition annotations. These would provide the necessary information so that the test-data generation could avoid generating test-data that the routines would never be exercised with in practice. Indeed the supply of such information allows even more of the verification conditions to be proved automatically through simplification by the SPADE 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 [24, 15] and consequently the automatic testing approach to gain confidence is still useful. Another possible solution to the lack of inter-procedural information is to

apply the test-data generation at a higher-level. As presented here the test-data generation is performed by executing the software unit under test. Instead it could be applied at the sub-system or even system level. The search would still target test-data specific exceptions at the unit level, but by generating input data for the sub-systems or system. Obviously, this will increase the size of the search space and also the complexity of the cost surface. At some point the cost surface will not provide enough information to the optimisation technique and the search process will break-down. Future work will look at how far the optimisation techniques can be pushed and where they start to break-down.

7 Conclusions

Many of the approaches for automated software test-data generation presented in the literature are inflexible or have limited capacity. Optimisation techniques in contrast 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 cost-function. This paper presents an extension of the framework to address the problems of testing exceptions. This extension is useful as the testing of exception conditions has very much been a 'poor relation' in testing research.

As with all testing approaches, only the presence of faults can be shown, not absence. Indeed, the failure of the test-data generation to find suitable test-data for an exception does not indicate the exception cannot occur, only that the search for test-data failed. However, given an intensive directed search for test-data, failure to locate test-data does allow increased confidence. This paper has shown how an automated testing strategy can be integrated into a process for proving exception freeness for safety-critical software systems. Here the automated testing is used to gain confidence in the likely correctness of the software, prior to investing time and money in proofs. It is suggested that applying this kind of testing should be a certification level test. Introducing additional complexity to mitigate against exceptions found when testing at the unit level would be undesirable. For example, the programmer may attempt to introduce additional code into the *SmoothSignal* routine to mitigate against the exception. However, the programmer is probably unaware that this exception cannot occur in practice due to system level protection. Therefore, this extra code only increases complexity, testing and maintenance costs and potentially reduces reliability.

An important fact is that the tools provided to sup-

port 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.

8 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 SEBPC project. It builds on work started under grant GR/K63702 also from the EPSRC. Rolls-Royce Plc. assisted with the evaluation, including the provision of the safety critical software for a civil aircraft engine controller system.

References

- [1] John Barnes. *Programming in Ada 95*. Addison-Wesley, 1995.
- [2] John Barnes. *High Integrity Ada: The SPARK Approach*. Addison-Wesley, 1997.
- [3] B. Beizer. *Software Testing Techniques*. Thomson Computer Press, 2nd edition, 1990.
- [4] John Clark and Nigel Tracey. Solving constraints in LAW. LAW/D5.1.1(E), European Commission - DG III Industry, 1997. Legacy Assessment Workbench Feasibility Assessment.
- [5] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.
- [6] R. Demillo and A. Offutt. Experimental results form an automatic test case generator. *ACM Transactions on Software Engineering and Methodology*, 2(2):109–127, April 1993.
- [7] 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.
- [8] Radio Technical Commission for Aeronautics. RTCS/DO-178B: software considerations in airborne systems and equipment, December 1992.
- [9] Matthew J. Gallagher and V. Lakshami Narashimhan. ADTEST: A test data generation

- suite for ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, August 1997.
- [10] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
- [11] Denis Howe. The free on-line dictionary of computing. <http://wombat.doc.ic.ac.uk>.
- [12] IEC. 61508 – functional safety of electrical / electronic / programmable electronic safety-related systems. International Electrotechnical Commission, Draft Standard, December 1997.
- [13] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.
- [14] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [15] Steve King, Jonathan Hammond, Rod Chapman, and Andy Pryor. The value of verification: Positive experience of industrial proof. In *to appear in Formal Methods 1999 Technical Symposium*, 1999.
- [16] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.
- [17] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.
- [18] B. Korel. Automated test data generation for programs with procedures. In *International Symposium on Software Testing and Analysis*, pages 209–215. ACM/SIGSOFT, 1996.
- [19] J. L. Lions. Ariane 5: Flight 501 failure report. Technical report, ESA/CNES, July 1996.
- [20] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, September 1976.
- [21] MoD. 00-55 requirements of safety related software in defence equipment. Ministry of Defence, August 1997.
- [22] Praxis Critical Systems. *Spark-Ada Documentation 2.0*, 1995.
- [23] C. Ramamoorthy, F. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, 1976.
- [24] Praxis Critical Systems Rod Chapman. Private communication, 1997.
- [25] Nigel Tracey, John Clark, and Keith Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis*, pages 73–81. ACM/SIGSOFT, 1998.
- [26] Nigel Tracey, John Clark, and Keith 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, 1998.
- [27] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE, October 1998.
- [28] Nigel Tracey, John Clark, John McDermid, and Keith Mander. Integrating safety analysis with automatic test-data generation for software safety verification. In *Proceedings of the 17th International Conference on System Safety*. IEEE, August 1999.
- [29] Jeffrey M. Voas and Gary McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. Wiley, 1998.
- [30] Alison Lachut Watkins. The automatic generation of test data using genetic algorithms. *Proceedings of the 4th Software Quality Conference*, 2:300–309, 1995.