SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2000; **30**:61–79

Automated test-data generation for exception conditions

N. Tracey*,†, J. Clark‡, K. Mander§ and J. McDermid¶

Department of Computer Science, University of York, Heslington, York, YO10 5DD, U.K.

SUMMARY

This paper presents a technique for automatically generating test-data to test exceptions. The approach is based on the application of a dynamic global optimization 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 proving 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. The results of a number of simple case-studies are presented and show the technique to be effective. The major result shows the application of the technique to a commercial aircraft engine controller system as part of a proof of exception freeness. This illustrates how automated testing can be effectively integrated into a formal safety-critical process to reduce costs and add value. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: test-data generation; verification; exception conditions

INTRODUCTION

A failure occurs when software is prevented from performing its intended action. A subclass of failures, which are known as exceptions [1], may be due to erroneous inputs, hardware faults or logical errors in the software code. 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 [2]. Indeed in a case-study by Toy [3] more than 50% of the operational failures of a telephone switching system were due to faults in exception handling and recovery algorithms. In a more recent incident, the Ariane 5 launch vehicle was lost due to an unhandled exception destroying $400 million of scientific payload [4].

*Correspondence to: Nigel Tracey, Department of Computer Science, University of York, Heslington, York, YO10 5DD, U.K.
†E-mail: njt@cs.york.ac.uk
‡E-mail: jac@cs.york.ac.uk
§E-mail: mander@cs.york.ac.uk
¶E-mail: jam@cs.york.ac.uk

Contract/grant sponsor: Engineering and Physical Science Research Council (EPSRC), UK; contract/grant number: GR-L4872

CCC 0038–0644/2000/010061–19$17.50
Copyright © 2000 John Wiley & Sons, Ltd.
Received 21 July 1999
Revised 17 September and 15 October 1999
Accepted 16 October 1999

Several languages now provide standard mechanisms for dealing with exceptions, for example Ada, C++ and Java. Exceptions are typically divided into two classes – language predefined and user defined. Language predefined exceptions are used for error conditions that result from hardware faults or run-time violation of the language rules (such as divide-by-zero and numeric overflow). User defined exceptions allow the development team to define and control additional exceptional conditions in an application specific manner. Standard language exception mechanisms greatly aid the structuring of exception handling, separating the exception-related code from the normal logic of the software. This improves the software's readability and makes it easier to maintain. In this paper the Ada language [5] model of exceptions has been used. However, many of the ideas and techniques for testing exceptions will be equally applicable to many other languages' exception mechanisms.

Exception code is responsible for the detection and handling of system conditions that could potentially lead to failure. It is therefore important that this code is tested effectively. This paper presents an approach for automatically generating test-data for the problems involved in testing exceptions and exception handling. The aim is to provide automated support for the testing of exceptions. The technique presented is a further application of the authors' automated test-data generation framework. This framework is based on the application of global heuristic optimization techniques and generalizes and builds on the work of others in this area.

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. Indeed, it is not only safety-critical systems that require proofs of exception freeness. The application of conventional code generation and optimization techniques can require code to be exception free [6]. A process is also outlined to combine proof and testing for exception freeness. Through this integration it is hoped the costs of exception freeness proofs can be substantially reduced.

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 [7–9]. 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 [10–14]. 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 represent 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 [15]. The work of Korel *et al.* built on this using locally directed search techniques [8,16–18]. This is further

Copyright © 2000 John Wiley & Sons, Ltd.

expanded by Gallagher *et al.* [19]. 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 [20]. The use of global optimization techniques for dynamic test-data generation has been investigated more recently in an attempt to overcome this limitation [9,20–23].

The authors' work has built on the previous optimization-based testing work to develop an extensible, generalized test-data generation framework. It has already been used to generate test-data for a wide variety of testing problems – constraint-solving [24], 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 generation of this input data is done directly from the software implementation without reference to the specification. It is important to note that the expected outputs for the tests must still be derived from the specification. The following section introduces the specific problems involved in testing exceptions and their handlers.

THE TEST-DATA GENERATION PROBLEM

A control flow-graph is a directed graph which represents the control structure of a program. It can be described as follows [29]: $G = (N, E, s, e)$, where N is a set of nodes, E is a set of edges of the form (n_i, n_j) and s and e are unique entry and exit nodes such that $s, e \in N$. A node, $n \in N$, is a sequence of statements such that if any one statement of the block is executed, then all are executed. This is also known as a *basic-block*. An edge $(n_i, n_j) \in E$ represents a possible transfer of control from the basic block n_i to the basic block n_j. For branch instructions the edges are associated with a *branch predicate*. This describes the conditions which must hold for the branch to be taken.

A program is driven down a path in the control flow graph by the values of its input variables and the global state. These can be described as the vector $x = \langle x_1, x_2, \ldots, x_n \rangle$. Each variable will have an associated domain, D_i, which can be determined from the variable's type. The total input space can be defined as the cross-product of each of these domains, $D = D_1 \times D_2 \times \cdots \times D_n$. This allows a program input to be defined as $x \in D$.

Two issues are of importance when testing exception conditions – raising the exception and test coverage of the exception handler.

Raising exceptions

Firstly, consider the case of a user-defined exception. User-defined exceptions must be explicitly raised in Ada by executing the `raise` statement. Assuming the `raise` statement is contained in basic block $n_i \in N$ (more accurately a `raise` statement will always be the last statement in a basic-block as it causes an unconditional branch), the problem of testing the raising of a user-defined exception reduces to one of selecting test-data $x \in D$ that will cause n_i to be executed.

In the case of predefined exceptions the problem is slightly different. 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 [30] is required. The focus, in this paper, is the generation

Copyright © 2000 John Wiley & Sons, Ltd.

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

A (D) := B;
D := (C * C) + B;
```

(a) Original Program

```
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
```

(b) Code with Compiler Checks

Figure 1. Predefined exception checks.

of test-data which violates run-time language rules. In Ada there are a number of predefined exceptions [5]:

`Constraint_Error` – data going out of range.
`Program_Error` – control-structure violation.
`Storage_Error` – running out of storage space.
`Tasking_Error` – general communications failure between tasks.

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 potentially proofs to show adequate system safety. Our major concern with testing has been these kinds of systems, hence we have focused on the SPARK-Ada language [31,32]. SPARK-Ada allows testing for `Constraint_Error` exceptions to be the focus of the work to date. The SPARK-Ada tool-set [31] mitigates against other pre-defined 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 illustrates the exception detection code that is inserted automatically by an Ada compiler for a simple program.

The test-data generation problem in this case is to find, for each point where a `Constraint_Error` is possible, data that causes the run-time violation. This can be thought of in

Copyright © 2000 John Wiley & Sons, Ltd.

exactly the same way as test-data generation for user-defined exceptions. That is, test-data is required to execute the basic-block containing the desired `raise` statement (basic block 2 and 4 in the example).

Coverage of exception handlers

Test coverage of exception handlers involves achieving structural coverage of the code. A number of structural coverage measures have been defined, these are reviewed by Ntafos [33] and Zhu *et al.* [34]. So, in essence, the test-data generation program for structural testing is finding a set of program inputs X (such that $X \subset D$) that achieves the desired coverage. Details of how the test-data generation framework addresses this problem can be found in Reference [27]. For testing exception handlers the test-data in X must also raise the desired exception.

OPTIMIZATION TECHNIQUES

Heuristic global optimization techniques are designed to find good approximations to the optimal solution in large complex search spaces. General purpose optimization 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. Optimization 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 [35] and genetic algorithms [36] have been used as the optimization 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 [36]. 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.

Genetic algorithms work by maintaining a population of sample solutions each of whose *fitness* 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 is calculated. The fitness function will be discussed in detail in the next section, for now it is sufficient to think of it as providing a quantitative measure of each solution's suitability for the problem at hand. Using this information members of the population are selected to become parents. Only selecting the fittest solutions tends to cause the search to converge too quickly into local optima. Random selection overcomes this by selecting parents randomly. Tournament selection combines these by selecting the fitter of two solutions with some probability otherwise a random choice is made.

Once the parents have been selected they need to be combined to form the offspring. The simplest crossover scheme is single point crossover. This selects a random point in a solution and combines the values from one parent up to this point with values from the other parent after this mid-point. Uniform crossover randomly decides which parent each data value will be drawn from to form the new solution. The probability of this decision can be weighted towards one of the parents to give weighted crossover. A selection of the offspring are then mutated to introduce diversity into the population.

Copyright © 2000 John Wiley & Sons, Ltd.

```
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;

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

Figure 2. Genetic algorithm.

The next generation (i.e. new population) is then selected from these offspring and the old population. Again a simple survival of the fittest tends to converge into suboptimal solutions. Therefore, some hybrid of survival of the fittest and random selection is normally used. An outline of the genetic algorithm search process is shown in Figure 2.

OPTIMIZATION BASED TEST-DATA GENERATION

To enable a dynamic search to be used to locate test-data it needs to be given some guidance. This guidance is given in the form of a fitness function. The amount of guidance given by the fitness 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. The input domain of most programs, D, is likely to be very large. A fitness surface could be formed by applying the fitness function to every possible program input. It is this surface which is effectively being searched when attempting to generate test-data. The size and complexity of the search space limits the effectiveness of simple gradient descent or neighbourhood searches as they are likely to get stuck in locally optimal solutions [37], hence failing to find the desired test-data. It is for this reason that global heuristic optimization techniques are used.

The fitness function must provide a measure of how *close* particular test-data is to executing the desired `raise` statement. The fitness function needs to return good values for test-data that *nearly* executes the `raise` statement and poorer values for test-data that is *far* from executing the `raise` statement. Since branch predicates determine the path followed they are vital in determining an effective fitness function.

Branch predicates consist of relational expressions connected with logical operators. The fitness 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

Copyright © 2000 John Wiley & Sons, Ltd.

Table I. Fitness function calculation.

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 (fit (a), fit (b))`
$a \wedge b$	`fit (a) + fit (b)`
$\neg a$	Negation is moved inwards and propagated over a

criterion for the search process. The fitness function is calculated as shown in Table I. In the table, K represents a failure constant which is added to further punish incorrect test-data.

In order to evaluate the fitness function it is necessary to execute an instrumented version of the SUT. There are two types of procedure call added by the instrumentation – branch evaluation and exception monitoring. Figure 3(a) shows a simple program that can raise a user-defined exception and a number of `Constraint_Error` exceptions. Figure 3(b) shows the same program with instrumentation.

The branch evaluation calls replace the branch predicates in the SUT. These functions (`Branch_1` and `Branch_2` in Figure 3(b)) are responsible for returning the Boolean value of the predicate and adding to the overall fitness the contribution made by each individual branch predicate that is executed. A branch is termed a critical branch if either the user has specified the desired outcome of the branch evaluation; or the branch evaluation affects the reachability of the desired exception. For critical branches the branch evaluation calls work as follows.

- Add the fitness of either *(branch predicate)* or ¬ *(branch predicate)* to the overall fitness for the current test-data depending on the required outcome.
- Within loops, adding the fitness of critical branch predicates is deferred until exit from the loop. At this point the minimum fitness evaluated for that branch predicate is added to the overall fitness. This prevents punishment of taking an undesirable branch until exit from a loop, as the desirable branch may be taken on subsequent iterations.

Copyright © 2000 John Wiley & Sons, Ltd.

```
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.

The exception monitoring calls simulate the expansion of the run-time checks inserted by the compiler. As illustrated above using this technique the test-data generation problem for pre-defined run-time exceptions is the same as for user-defined exceptions, i.e. finding test-data to execute the basic-block containing the `raise` statement. These instrumentation calls (`Excep_1` to `Excep_6` in Figure 3(b)) encode the condition required to cause a run-time exception, and function as follows.

- If the exception is required to be raised then the fitness of the condition required to cause the exception is added to the overall fitness for the current test-data, otherwise add fitness of the negated exception condition.
- Again, as for branch evaluation calls, adding the fitness of exception conditions is deferred until exit from the loop.

The problem of structural test coverage of the exception handler is addressed in a similar manner. In this case, it is only the branch predicates that are important as they dictate the flow of control. The search technique aims to generate test data which raises the desired exception as detailed above, in addition the branch predicate fitness functions are used to guide the search to test-data that executes the desired path through the exception handler. The details of how optimization can be used to generate structural test-data are presented in Reference [27]. 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 optimization system. It is the execution of

Copyright © 2000 John Wiley & Sons, Ltd.

this system that dynamically executes the SUT, calculates the fitness values and ultimately generates the required test-data.

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 fitness 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 fitness function.

No.	X	Y	Fitness	Description
1	-3	4	$3+K$	Fails at `Branch_1`
2	2	-5	$32+K$	Fails at `Excep_6`
3	-5	-1	$5+K$	Fails at `Branch_1`
4	1	-5	$45+K$	Fails at `Excep_6`
5	3	2	$0+K$	Fails at `Branch_2`
Ave. Fitness			27	

At this point the prospective parents are selected and the offspring population formed by applying the uniform crossover and mutation operators. This gives the following offspring.

No.	X	Y	Fitness	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. Fitness			24	

The next generation is then selected, using a hybrid of elite survival and random selection. This new population is as follows.

No.	X	Y	Fitness	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. Fitness			19.8	

Copyright © 2000 John Wiley & Sons, Ltd.

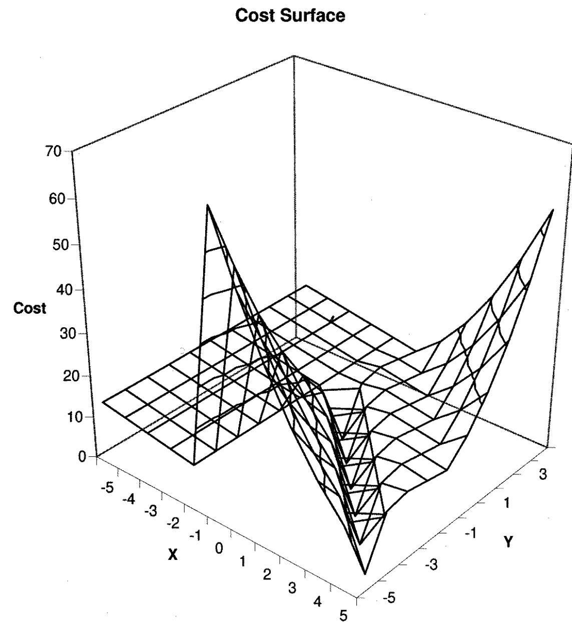

Figure 4. Fitness surface for divide-by-zero exception.

It can be seen that the average population fitness 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	Fitness	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. Fitness			7.8	

At this point the search can stop as solutions 17 and 20 both have a fitness of zero and hence represent test-data that will raise the desired exception. Figure 4 shows the fitness 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 – $(0, 1)$, $(1, 0)$, $(2, -1)$, $(3, -2)$, $(4, -3)$ and $(5, -4)$.

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

Copyright © 2000 John Wiley & Sons, Ltd.

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

Figure 5. Verification conditions.

optimization technique uses this to guide its generation of test-data until either it has successfully found test-data with a zero fitness or until no further progress can be made, perhaps due to locally optimal points in the fitness surface.

FREEDOM FROM EXCEPTIONS

Many systems, including aviation, commerce, medical and office 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 [38], civil aviation standard DO178B [39] and generic standard IEC-61508 [40]. 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 block with compiler checks there are actually five. However, if it can be proved that 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 correct implementation of exception handlers that might be used in full Ada [32].

SPARK-Ada and its associated tools [31,32] (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 which 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 and its associated check annotation, – – # *check B in Integer*. From this the SPARK examiner generates the verification conditions shown in Figure 5.

Copyright © 2000 John Wiley & Sons, Ltd.

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 be able to simplify them to true (or possibly false). In the cases when 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 that aims to show exception freeness for safety-critical software. By integrating this automated testing approach the aim is to reduce the costs involved in proving exception freeness. 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.

EVALUATION

This section presents the results of an evaluation of the optimization based approach to generating test-data for exception conditions. The evaluation has been performed in two parts. Firstly, a collection of small Ada 95 programs have been used to provide a preliminary assessment of the ability of the system to generate test-data to raise particular exceptions. Test-data to achieve exception condition coverage has also been targeted. Secondly, integration of the test-data generation and proof of exception freeness is evaluated using the code for a civil aircraft engine controller.

Simple examples

A number of Ada 95 programs have been used to evaluate the effectiveness of this test-data generation approach. The routines are between 10 and 200 lines of code. Square simply squares the input parameter. However, the data-types are defined such that an overflow exception will be raised with large input values. Int Sqrt is an integer square-root routine that uses a binary search algorithm. Find performs either a linear or binary search to locate a value in an array. A user-defined exception and handler is invoked if a binary search is requested on an unsorted array. Remainder calculates the

[‖] Saturation does not give mathematically accurate results. Where overflow would occur the result saturates at the ceiling of the range and similarly for underflow. For divide-by-zero the saturating operation returns the largest positive or negative value in the range depending on the dividend.

Copyright © 2000 John Wiley & Sons, Ltd.

Table II. Evaluation results (Key: INP - cardinality of the input domain, NE - Number of exception conditions, EF - exception conditions successfully executed, C - branch coverage of exception handlers, T - test-data generation time).

SUT Name	INP	NE	EF	C	T
Square	20 002	1	1	N/A	0.5s
Int Sqrt	10 000	3	3	N/A	1.1s
Find	$1e+44$	2	2	100%	2.0s
Remainder	$1e+8$	2	2	N/A	2.4s
Tomorrow	286 440	5	5	100%	4.8s
Convert	$1.7e+10$	7	7	N/A	17.2s
BigInt Div	$1e+50$	4	3	100%	36.2s
Total		24	23		

remainder and quotient given two input parameters. Tomorrow calculates tomorrow's day, date, month and year taking into account leap year calculations. User-defined error handling is used to validate the input date. Convert performs conversions between binary, octal, decimal, hexadecimal and Roman numeral strings. BigInt Div performs an integer division using arbitrary length integer abstract data types. Error seeding was used on a number of the programs to introduce errors that allowed exceptions to be generated. Table II shows the results of generating test-data for these programs**.

These results demonstrate that test-data can effectively be automatically generated to test exceptions. However, these results are only on relatively trivial programs. The authors' focus is the provision of automatic testing solutions for high-integrity safety-critical systems. A more detailed evaluation of the test-data generation technique is presented in the next section for such a system.

Aircraft engine controller

As part of the evaluation of the integration of test-data generation and proof for exceptions the technique has been applied to the code for a civil aircraft engine [41]. 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 6 shows an example routine from the engine controller code.

**The evaluation was performed on a Pentium Pro 200MHz system running Linux.

Copyright © 2000 John Wiley & Sons, Ltd.

```
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 6. Smooth signal subprogram.

The first step in this evaluation was to use the SPARK Examiner to extract the run-time verification conditions from the source code. The verification conditions detail the proof obligations for a proof of exception freeness. The SPARK automatic simplifier was able to discharge 89% of these verification conditions completely automatically. For the example given in Figure 6 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 verification 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

Copyright © 2000 John Wiley & Sons, Ltd.

```
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 7. Verification conditions for smooth signal.

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. The application of the test-data generation technique requires the execution of an instrumented version of the software as discussed earlier. The validity of testing the instrumented code is problematic for safety-critical systems. However, we view the test-data generation technique as a method of simply obtaining test-data. The development team should then use this test-data within their certified development environment to validate it. Figure 7 shows the remaining two verification conditions for the smooth signal subprogram (the irrelevant hypotheses have been removed).

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

Copyright © 2000 John Wiley & Sons, Ltd.

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 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 information that causes test-data to be generated that the system could never generate in practice. 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 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 [42,43] and consequently the automatic testing approach to gain confidence is still useful. Another possible solution to the lack of inter-procedural level 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 fitness surface. At some point the fitness surface will not provide enough information to the optimization technique and the search process will break-down. Future work will look at how far the optimization techniques can be pushed and where they start to break-down.

CONCLUSIONS

Many of the approaches for automated software test-data generation presented in the literature are inflexible or have limited capacity. Optimization techniques in contrast offer a flexible and efficient approach to solving complex problems. To allow the optimization based framework to generate test-data for a specific testing criterion it is necessary only to devise a suitable fitness 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 this 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

Copyright © 2000 John Wiley & Sons, Ltd.

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 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.

The results of the initial evaluation are very encouraging. The test-data generation system was able to find test-data to cause exceptions and cover the exception handling code efficiently.

Further work

The results presented above show that it is possible to use optimization techniques to generate test-data for the testing of exceptions. However, more research is required in order to assess the limitations of the approach. The expression structure in the safety-critical code was very simple. This allowed the automatic simplifier to discharge the vast majority of the verification conditions itself. The test-data generation could then be targeted towards only the remaining verification conditions. The application of the test-data generation to a system where exceptions can be raised from many expressions (and indeed subexpressions) may be very time consuming and no longer practical. A new search would be required for every possible point where an exception may be raised. However, safety-critical systems by their very nature tend to have simple control-flow and expression structuring as was the case with the engine-controller code. The simple exception handling code of the remaining evaluation programs made achieving coverage of this code easier. A more detailed investigation with more complex exception handling software is needed fully to assess the technique's ability to generate test-data to achieve structural coverage of exception handling code.

Further investigation into optimization techniques is required to discover their relative strengths and weaknesses. Optimization techniques have a large number of tunable parameters which could potentially have an impact on the ability to generate good quality test-data when the search space is complex. Optimization techniques will never be able to guarantee their results. However, it may be possible use software metrics to give guidance in a number of areas – to suggest which optimization techniques will give the best results; to suggest suitable parameter values for the optimization techniques; and also to give an indication as to the likely confidence that can be gained from the results.

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 provided access to source code and support in carrying out the aircraft engine controller case-study.

Copyright © 2000 John Wiley & Sons, Ltd.

REFERENCES

1. Denis Howe. The free on-line dictionary of computing. http://wombat.doc.ic.ac.uk.
2. Leveson NG. *Safeware: System Safety and Computers*; Addison Wesley, 1995.
3. Toy WN. Fault-tolerant design of local ESS processors. *The Theory and Practice of Reliable System Design*, 1981.
4. Lions JL. Ariane 5: Flight 501 failure report. *Technical report, ESA/CNES*, July 1996.
5. ISO/IEC 8652:1995. *Ada 95: Language Reference Manual*, 1995.
6. McHugh J. Towards efficient code from verfied programs. *Technical Report 40*, Institute for Computer Science, University of Texas, March 1984.
7. Beizer B. *Software Testing Techniques*, 2nd edn.; Thomson Computer Press, 1990.
8. Korel B. Automated test data generation for programs with procedures. *International Symposium on Software Testing and Analysis*; ACM/SIGSOFT, 1996; 209–215.
9. Jones BF, Eyres DE, Sthamer HH. A strategy for using genetic algorithms to automate branch and fault-base testing. *The Computer Journal* 1998; **41**(2):98–107.
10. Boyer R, Elspas B, Levitt K. SELECT – a formal system for testing and debugging programs by symbolic execution. *Proceedings International Conference on Reliable Software*, 1975; 234–245.
11. Clarke L. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering* September 1976; 215–222.
12. Demillo R, Offutt A. Experimental results form an automatic test case generator. *ACM Transactions on Software Engineering and Methodology* 1993; **2**(2):109–127.
13. King J. Symbolic execution and program testing. *Communications of the ACM* 1976; **19**(7):385–394.
14. Ramamoorthy C, Ho F, Chen W. On the automated generation of program test data. *IEEE Transactions on Software Engineering* 1976; **SE-2**(4):293–300.
15. Miller W, Spooner. D. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering* 1976; **SE-2**(4):223–226.
16. Ferguson R, Korel B. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology* 1996; **5**(1):63–86.
17. Korel B. Automated software test data generation. *IEEE Transactions on Software Engineering* 1990; **16**(8):870–879.
18. Korel B, Al-Yami AM. Assertion oriented automated test data generation. *18th International Conference on Software Engineering*; IEEE, 1996; 71–80.
19. Gallagher MJ, Narashimhan VL. ADTEST: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering* 1997; **23**(8):473–484.
20. Jones B, Sthamer H, Eyres D. Automatic structural testing using genetic algorithms. *Software Engineering Journal* 1996; **11**(5):299–306.
21. Xanthakis S, Ellis C, Skourlas C, Le Gall A, Katsikas S, Karapoulios K. Application des algorithmes genetiques au test des logiciels. *Proceedings of 5th International Conference on Software Engineering*, 1992; 625–638.
22. Watkins AL. The automatic generation of test data using genetic algorithms. *Proceedings of the 4th Software Quality Conference*, 1995; **2**:300–309.
23. Jones BF, Sthamer HH, Eyres DE. Generating test-data for Ada procedures using gentic algorithms. *Genetic Algorithms in Engineering Systems: Innovations and Applications*; IEEE, September 1995; 65–70.
24. Clark J, Tracey N. Solving constraints in LAW. LAW/D5.1.1(E), European Commission - DG III Industry, 1997. Legacy Assessment Workbench Feasibility Assessment.
25. Tracey N, Clark J, Mander K. Automated program flaw finding using simulated annealing. *International Symposium on Software Testing and Analysis*; ACM/SIGSOFT, 1998; 73–81.
26. Tracey N, Clark J, Mander K. The way forward for unifying dynamic test case generation: The optimisation-based approach. *International Workshop on Dependable Computing and Its Applications*; IFIP, 1998; 169–180.
27. Tracey N, Clark J, Mander K, McDermid J. An automated framework for structural test-data generation. *Proceedings of the International Conference on Automated Software Engineering*; IEEE, October 1998.
28. Tracey N, Clark J, McDermid J, Mander K. Integrating safety analysis with automatic test-data generation for software safety verification. *Proceedings of the 17th International Conference on System Safety*; IEEE, August 1999.
29. Deo N. *Graph Theory with Applications to Engineering and Computer Science*; Prentice-Hall, 1974.
30. Voas JM, McGraw G. *Software Fault Injection: Inoculating Programs Against Errors*; Wiley, 1998.
31. Praxis Critical Systems. *Spark-Ada Documentation 2.0*, 1995.
32. Barnes J. *High Integrity Ada: The SPARK Approach*; Addison-Wesley, 1997.
33. Ntafos SC. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering* 1988; **14**(6):868–874.
34. Zhu H, Hall PAV, May JHR. Software unit test coverage and adequacy. *ACM Computing Surveys* 1997; **29**(4):366–427.
35. Kirkpatrick Jr S, Gelatt CD, Vecchi MP. Optimization by simulated annealing. *Science* 1983; **220**(4598):671–680.

Copyright © 2000 John Wiley & Sons, Ltd.

36. Holland JH. *Adaptation in Natural and Artificial Systems*; University of Michigan Press, 1975.
37. Rayward-Smith VJ, Osman IH, Reeves CR, Smith GD (eds). *Modern Heuristic Search Methods*; Wiley, 1996.
38. MoD. 00-55 requirements of safety related software in defence equipment. Ministry of Defence, August 1997.
39. Radio Technical Commission for Aeronautics. RTCS/DO-178B: software considerations in airborne systems and equipment, December 1992.
40. IEC. 61508 – functional safety of electrical / electronic / programmble electronic safety-related systems. International Electrotechnical Commission, Draft Standard, December 1997.
41. Tracey N, Clark J, Mander K, McDermid J. Integrating automated testing with exception freeness proofs for safety critical systems. *Proceedings of the 4th Australian Workshop on Safety Critical Systems and Software*; The Australian Computer Society, 1999.
42. Chapman R. Praxis critical systems. Personal communication, 1997.
43. King S, Hammond J, Chapman R, Pryor A. The value of verification: positive experience of industrial proof. *Formal Methods 1999 Technical Symposium*, 1999.

Copyright © 2000 John Wiley & Sons, Ltd.