

Integrating Safety Analysis with Automatic Test-Data Generation for Software Safety Verification

Nigel J. Tracey; University of York; York, UK

John Clark; University of York; York, UK

Prof. John McDermid; University of York; York, UK

Dr. Keith Mander; University of York; York, UK

Keywords: software fault tree, automated testing, optimisation, safety verification

Abstract

Typically verification focuses on demonstrating consistency between an implementation and a functional specification. For safety critical systems this is not sufficient, the implementation must also meet the system safety constraints and safety requirements.

The work presented in this paper builds on the authors' previous work in developing a general framework for dynamically generating test-data using heuristic global optimisation techniques. This framework has been adapted to allow automated test-data generation to be used to support the application of software fault tree analysis. Using the framework a search for test-data that causes an identified software hazard condition can be performed automatically. The fact that a hazardous condition can arise may be discovered much earlier than with conventional testing using this automated approach. If no test-data is located then SFTA, or other forms of static analysis, can be performed to give the necessary assurance that no such data exists.

A number of extensions to this basic approach are also outlined. These are, integration with fault injection, testing for exception conditions and testing for safe component reuse and integration. Preliminary results are encouraging and show that the approach justifies further research.

Introduction

Typically verification focuses on demonstrating consistency between an implementation and a functional specification. For safety critical systems this is not sufficient, the implementation must also satisfy the system safety constraints and safety requirements. Safety is normally defined in terms of hazardous conditions which

may lead (or contribute) to an accident. It is important to note that not all errors cause hazards and conversely not all software, which correctly implements its specification, is safe. The objective of safety verification is to provide evidence for the safety case showing that each of the identified hazardous conditions cannot occur (or that the probability of failure is acceptably low). This can be achieved using two distinct approaches – dynamic testing or static analysis. These are described in the next section.

The work presented in this paper builds on the authors' previous work in developing a general framework for dynamically generating test-data using heuristic global optimisation techniques. Dynamic search techniques such as genetic algorithms and simulated annealing are used to optimise a cost-function, by repeated program execution and cost-evaluation. The cost-function is designed so optimal values represent desirable test-data. This paper shows how this automatic approach to test-data generation can be integrated with analysis techniques to provide efficient and cost-effective assurance in both the functional correctness and safety of the software. Such test-data also aids in the collection of software safety verification evidence required for the system safety case.

The remainder of this paper is structured as follows. The next section describes the application of both dynamic testing and static analysis to safety verification. The approach to test-data generation for safety property verification is then described. A number of other safety-related applications of the test-data generation technique are introduced and their integration with static analysis approaches detailed – the applications include testing of exception conditions, testing of safety in the presence of hardware faults using fault injection and the testing of the safety of software reuse.

The paper closes with some preliminary results, conclusions and identifies areas requiring further research.

Safety Verification

As already mentioned safety verification can be achieved using two distinct approaches (ref 1) : dynamic testing and static analysis.

Dynamic safety testing: Dynamic testing requires execution of the software under test (SUT), the results of this execution are evaluated with respect to the safety requirements. Dynamic testing allows confidence to be built in desirable properties of the software. This confidence is built by execution of the SUT in a suitable environment with test-data designed to illustrate desirable properties. Testing can only ever be used to show the presence of faults not their absence. Despite this, testing normally focuses on testing standard correct software behaviour. However, for software safety testing negative testing (testing of erroneous and out-of-range inputs, fault tolerant behaviour in response to hardware errors, etc) is vitally important.

Static safety analysis: Static analysis does not require execution of the SUT. Static analysis examines a representation of the SUT. Using static analysis allows general conclusions to be made (i.e. the results are not limited to the specific test-case and test-data selected as with testing). This allows assurance to be gained in properties of the software. However, this does not imply that static safety analysis is better than dynamic safety testing.

With static analysis it is necessary to verify the accuracy of the safety properties of the model with respects to the actual software. Validation of the assumptions underlying the analysis is also required. The optimal approach is to use both dynamic safety testing and static safety analysis as complementary techniques to achieve the goal of generating evidence for the software safety case and ultimately providing safe software. Indeed effective test planning should aim to use the results of static safety analysis to target where the testing resource can best be used. By integrating the two techniques the goal is to both increase confidence in the safety of the software, but also to reduce the costs involved in safety verification. The costs can be reduced because the two techniques are being used in

harmony, each to their own relative strengths, rather than as disjoint techniques.

The integration of static safety analysis and dynamic safety testing is the focus of this paper. The results of static analysis are used to give criteria for the dynamic testing. These criteria are used to enable automatic generation of the test-data required for the dynamic testing. In turn the dynamic testing is used to complement and reduce the costs involved in performing static analysis. Static analysis can be either formal or informal. This paper examines the use and integration of the informal technique of software fault tree analysis with automated test-data generation for dynamic safety verification.

Software Fault Tree Analysis

Formal proof is one process that can be used to provide confidence in the safety properties of a software system. However, many of the benefits of a formal approach can be realised by using a disciplined and rigorous process to focus on the problem without actually carrying out the proofs (ref 1). The key to achieving increased confidence in the safety of the software is to focus on the safety properties to the exclusion of all superfluous details. One such technique is that of software fault tree analysis (SFTA). Leveson and Harvey developed SFTA in the early 1980s (ref 2). To apply SFTA the first step is to assume that the system has failed in some manner, i.e. a failure condition holds at some specific point in the program's execution. The process is then to work backwards through the software to determine the set of possible input conditions that enable the failure condition to occur (or preferably to show that such conditions cannot occur). The failure conditions, or hazards, are initially identified using some form of hazard analysis technique, e.g. functional hazard assessment (FHA). SFTA basically analyses paths backwards through the software statement-by-statement to eventually identify inputs (or internal states) that may cause the identified failure condition and potentially hazardous outputs.

Fault tree templates are required for each statement of the underlying language. These templates are used to generate the software fault tree and are dependent upon the semantics of the underlying programming language. Figure 1 shows the fault tree template for an Ada (ref 3) assignment statement.

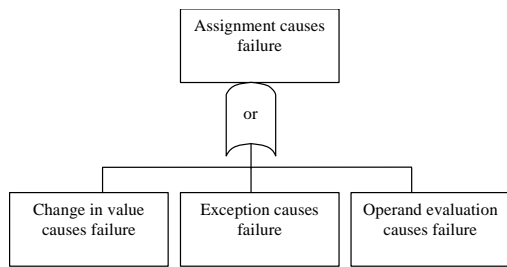


Figure 1 – Fault Tree Template for Ada Assignment Statement

At the root of this template it is assumed that the assignment statement itself causes the failure condition. This is then broken down to indicate that the failure condition will occur if any of the following happens – the change in value causes the failure, the operand evaluation causes the failure or an exception raised during execution of the assignment statement causes the failure. The application of SFTA effectively hoists a hazardous condition backwards through the software using fault tree templates. The end result is a weakest pre-condition that describes the possible system states and inputs that can lead to the specified hazardous condition occurring. SFTA can be considered a graphical representation of weakest preconditions and axiomatic verification (ref 4), but using

conditions that describe hazards rather than conditions describing the functional behaviour. FTA continues until either a true or false weakest pre-condition is found or an input statement is reached. A false condition indicates a contradiction in the tree, this indicates that the post-condition (i.e. the hazardous condition) can never occur via that execution path. Conversely, a satisfiable weakest pre-condition in the tree means the software will cause the hazardous condition (given suitable inputs) and hence is unsafe. Figure 2 shows a fault tree for a simple sequence of statements (ref 5) and shows how the conditions are propagated backwards through the statements. The effects of exceptions have been ignored in this example, but are considered later in this paper. In this example the variable X is both an input and output and Y is an input. The failure condition under analysis here is when the output variable X is greater than 100. The else-branch on the right-hand side of the fault tree shows a contradiction, i.e. assigning 10 to X can never cause the fault where $X > 100$. The then-branch on the left-hand side of the fault tree requires further analysis. Analysis is required to determine whether $f(Y) > X - 10$ can occur with any input values of X and Y and also to determine whether the assignment of $f(Y)$ to X can cause $X > 100$ for any input values of Y . Both of these conditions must be satisfied for the fault condition to occur.

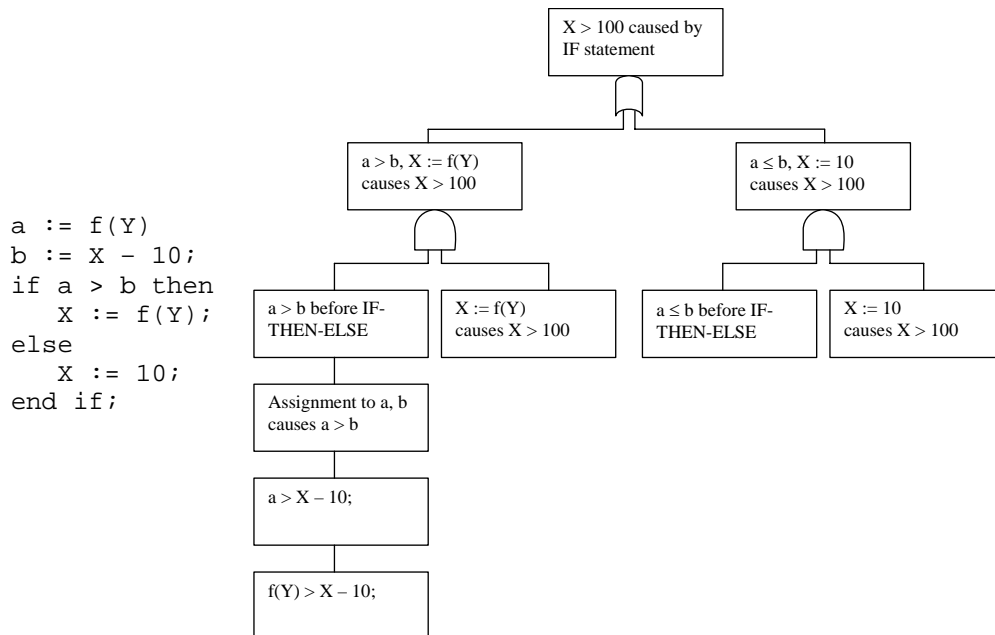


Figure 2 – Example Code and Fault Tree for the Fault Condition $X > 100$

Test-Data Generation for Safety Properties

SFTA can be used to guide the application of testing effort when testing for safety properties. In the example above (figure 2), the function $f()$ has not been analysed and integrated into the fault tree. However, the information that is already gained about the calling environment can be used to reduce the test-set. So, in this example, the function $F(Y)$ will only need to be tested when a is greater than b (in fact, $f(Y) > X - 10$) as it is already known that this is a pre-condition to the failure occurring.

Testing can also be used to aid the targeting of effort for the construction of a software fault tree. A single test that illustrates a fault condition is all that is required to show that there is no contradiction to be found and that the analysis should eventually end in a satisfiable pre-condition for the fault. If testing can find no such test-case then this gives increased confidence that the analysis will be able to find the desired contradiction showing that the fault condition cannot occur. If this kind of testing can be performed cheaply, then the total cost of performing the safety verification can be reduced. The remainder of this section outlines an automated approach to generating test-data to aid in the application of SFTA. The approach is based on the application of a generalised test-data generation framework that has been developed. The framework uses dynamic global heuristic search techniques, such as simulated annealing (ref 6) and genetic algorithms (ref 7), to automate the generation of test-data for a number of testing problems (ref 8-10).

To show that a particular implementation does not fully meet its safety specification it is necessary to find test-data that causes an implementation to violate a safety property. For a dynamic search to succeed, it needs to be given some guidance. This guidance is provided in the form of a cost-function. The cost-function relates test-data to a measure of how *good* it is, with respect to the objective. In this case the objective is the violation of a safety property. Obviously, any particular test will either meet this objective or it will not. However, this is not sufficient to guide the search. A cost-function is required which will return *good* values for test-data that *nearly* meets the objective and *worse* values for test-data that is a long way from meeting the objective. The SUT is executed repeatedly and the cost-function calculated. The values from the

cost-function are used by the search technique to guide the search to new test-data until, hopefully, the desired test-data is located.

We will begin by considering test-data generation for safety properties required to hold after execution of a software unit. In this case the safety property itself is vital in determining the cost-function.

Typically, safety properties will arise from software level hazard analysis, e.g. SHARD, although discussion of such techniques is outside the scope of this paper. These properties consist of relational expressions connected with logical operators. The cost-function can be designed such that it will evaluate to zero if the safety property evaluates to false and will be positive otherwise. The larger the positive values, the further the test-data is from achieving the desired safety violation. This property gives an efficient stopping criterion for the search process, i.e. the search stops once a test-data with a zero cost has been found. The cost-function is calculated as shown in table 1. In the table K represents a failure constant which is added to further punish undesirable test-data.

Table 1 – Cost-Function Calculation

Element	Value
Boolean	if TRUE then 0 else K
$a = b$	if $\text{abs}(a-b) = 0$ then 0 else $\text{abs}(a-b) + K$
$a \neq b$	if $\text{abs}(a-b) \neq 0$ 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 $(b-a) + K$
$a \vee b$	$\text{Min}(\text{cost}(a), \text{cost}(b))$
$a \wedge b$	$\text{Cost}(a) + \text{cost}(b)$
$\neg a$	Negation propagated over a

A simple example will help illustrate how this guides the search process. Consider a simple program as shown in figure 3 and the hazardous condition, $Z > 100$.

```

type T is Integer range 1 .. 10;
procedure SUT(X, Y : in T;
              Z : out Integer)
is
begin
  Z := Y;
  for I in X .. Y loop
    Z := Z + 10;
  end loop;
end SUT;

```

Hazard Condition $Z > 100$

Figure 3 – Example SUT

If, say, the search technique started with a random input of $X = 3$ and $Y = 7$ then the cost would be calculated as follows.

$$\begin{aligned}
 \text{Cost}(Z > 100) &= \text{Cost}(57 > 100) \\
 &= 100 - 57 + K \\
 &= 43 + K
 \end{aligned}$$

At this point the calculated cost is returned to a search technique that uses this value to guide the selection of new test-data. Simulated annealing uses the cost value to guide its search to optimal solutions within a neighbourhood. Improved solutions (i.e. test-data with a lower cost value) are always accepted, however worse solutions are accepted in a controlled manner. The idea is that it is better to accept a short term penalty in the hope of finding significant rewards longer term. In this way simulated annealing hopes to escape from locally optimal solutions. Genetic algorithms use the cost value to select solutions to combine and survive into later generations. This idea is based on evolution in nature. Better solutions are combined (bred) in the hope of attaining the best features of each. If the search technique moved on to consider the test-data $X = 2$ and $Y = 10$ (in the case of simulated annealing this solution might be in the neighbourhood of the previous solution), the cost would be calculated as follows.

$$\begin{aligned}
 \text{Cost}(Z > 100) &= \text{Cost}(100 > 100) \\
 &= 100 - 100 + K \\
 &= K
 \end{aligned}$$

Again, at this point the calculated cost is returned to a search technique to guide the selection of new test-data. It can be seen that while this test-data still does not violate our safety property the cost-function indicates that this new test-data is much closer to doing so. If the search technique

were to now generate $X = 1$ and $Y = 10$, it can be seen that the cost will evaluate to zero. This indicates that test-data has been found that causes the software to violate the given safety property. In this case, the output value of Z is 110, which was identified as a hazardous condition.

We have now shown how optimisation can be used to automatically search for test-data that violates safety properties that must hold after execution of the SUT. However, to provide useful support to SFTA it would be useful to generate test-data to violate safety conditions at specified points during the execution of the SUT.

As already discussed SFTA back-propagates hazard conditions through the software statement-by-statement. This effectively gives a pre-condition at each point in the SUT that if satisfied will cause the hazardous condition to occur. The cost function, described above, can be adapted to guide the search to test-data that satisfies a hazard pre-condition at specified points in the SUT. These points and the hazard pre-conditions would be identified when applying SFTA. Since branch predicates determine the path followed they are vital in determining an effective adaptation to the cost-function.

Branch predicates consist of relational expressions connected with local operators. Hence the same cost-function can be used. However, in order to evaluate the cost-function for branch predicates it is necessary to execute an instrumented version of the SUT. The instrumented SUT contains procedure calls that monitor the evaluation of branch predicates and hazard pre-conditions. The branch evaluation calls replace the branch predicates in the SUT. These functions (**Branch_1** In figure 4(b)) are responsible for returning the boolean value of the predicate they replaced and adding to the overall cost the contribution made by each individual branch predicate that is executed. They function as follows.

- If the target node (basic-block containing the identified hazard pre-condition) 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

```

function F (X : My_Integer)
  return Integer is
begin
  if X < 0 then
    return X * X;
  end if;
  Do_Control
  ((X-85)*(X+85)*(X-90)*(X+90));
  return ((X-85)*(X+85)*(X-90)*(X+90)) /
  (X-1)*(X-5);
end F;

```

(a) Original Program

```

function F (X : My_Integer)
  return Integer is
begin
  if Branch_1 (X) then
    return X * X;
  end if;
  Safe_1 ((X-85)*(X+85)*(X-90)*(X+90));
  Do_Control
  ((X-85)*(X+85)*(X-90)*(X+90));
  return ((X-85)*(X+85)*(X-90)*(X+90)) /
  (X-1)*(X-5);
end F;

```

(b) Instrumented Program

Figure 4 – Example program and Instrumentation

of $\neg(\text{branch predicate})$ to the overall cost for the current test-data.

- If the current test-data causes an undesired branch to be taken (i.e. target node is non-longer reachable) 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 to the overall cost is deferred until exit from the loop. At this point the minimum cost evaluated within the loop 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 target node being executed add nothing to the cost. However, it is also possible for the user to specify the complete path that is to be followed. This further constrains the search to locate test-data that satisfies the hazard pre-condition and follows the specified path.

The hazard pre-condition monitoring calls (e.g. `safe_1` in figure 4(b)) work exactly like the previous cost-function. However, they are now evaluated at a specified point within the SUT.

To demonstrate how this works in practice, consider the problem of generating test-data that causes a safety violation in the program in figure 4(a). Using SFTA it has been determined that at the point of the call to `Do_Control` it is unsafe for the parameter of the call to be negative. The following illustrates how the cost function works. If, say, the search technique started with a random input of $X = -5$ then the cost would be calculated as follows.

Expression	Cost Contribution
Branch_1: $\neg(X < 0)$	$(0 - (-5)) + K = 5 + K$

At this point the cost-function terminates the execution of the SUT. This is because with the current test-data the wrong branch is taken. The search technique takes the cost value and uses it to guide the selection of new test-data, say $X = 50$.

Expression	Cost Contribution
Branch_1: $\neg(X < 0)$	0
Safe_1: Param < 0	$(26460000 - 0) + K$

At this point the cost-function terminates the SUT and returns the cost to guide the search. The search technique will then select new test-data, $X = 80$ and the process will repeat.

Expression	Cost Contribution
Branch_1: $\neg(X < 0)$	0
Safe_1: Param < 0	$(1402500 - 0) + K$

It can be seen that this new test-data results in a substantially lower cost-value. Again the cost-function terminates the execution of the SUT and the search technique generates new data, $X = 87$. With this data the both `Branch_1` and `Safe_1` will evaluate to zero, hence the overall cost is zero. This indicates that $X = 87$ is suitable test-data for its given purpose, i.e. calling `Do_Control` with a negative parameter value.

Exceptions

An important aspect of arguing the safety of software is showing that there are no safety violations arising from unexpected exceptions or exception handling. The importance of exceptions for safety is illustrated by the emphasis on exceptions in the SFTA templates for the Ada programming language (ref 11).

Typically with safety-critical systems run-time checks are removed from the final production system. For this to be valid it is important to show that the system is free from run-time exceptions. Indeed it is often easier to reason that a program is free from exceptions than to reason about the safe implementation of exception handlers.

The techniques outlined above, for hazard-condition testing, can be extended to address the problem of generating test-data to raise specified exceptions. In the case of exceptions the condition required to *raise* the exception is used in place of the hazard pre-condition instrumentation. This exception condition instrumentation is derived from the run-time checks that would be inserted by a compiler. If software is assumed to be free from run-time exceptions, the safety of this assumption needs to be verified. The cost-function for this problem needs to indicate whether the target exception is raised for the generated test-data.

Consider the program in figure 2(a) again, this time the target is to generate test-data that raises a divide-by-zero exception in the second return statement. The version of the SUT differs slightly from that shown in figure 2(b). The SUT here is instrumented for exception conditioning monitoring (`Excep`). These calls calculate the cost contribution made by the run-time exception checks. The check of interest in this example is the check that $(X-1) \times (X-5)$ evaluates to zero, causing a divide-by-zero error. Starting with random test-data of $X = 0$, the cost would be calculated as follows.

Expression	Cost Contribution
Branch_1: $\neg(X < 0)$	0
Excep: $(X-1)*(X-5)=0$	$\text{abs}(5-0)+K = 5+K$

This test-data does not cause the exception to be raised, hence the execution of the SUT is terminated and the calculated cost value returned to the search technique. The search technique will then use this value to guides its selection of new test-data, say $X = 5$ and the cost is re-evaluated by executing the SUT. This time the total cost is zero indicating that test-data has been located that raises the desired exception.

To achieve the desired confidence in run-time exception freeness the ideal would be to combine automated dynamic testing (as outlined above) with static techniques such as proof. If any test-

data can be found that causes an exception then clearly any proof attempt would fail. By integrating the automated testing approach the aim is to reduce the total costs involved in showing exception freeness. The automated exception testing can also be used in systems that are not assumed to be free from exceptions. In this case the desired test-data is that which raises the exception that in turn causes a safety property to be violated. To achieve this the cost value for test-data receives contributions both from the exception condition and hazardous condition monitoring. This complements the analysis required to apply SFTA as can be seen from the fault-tree template in figure 1. Test-data that illustrates an example of an exception causing a hazardous condition means that the SFTA will be unsuccessful in its search for a contradiction.

Fault Injection

There is some evidence that safety problems are more prevalent in error-handling code (such as fault tolerant code) than in *normal* code. In any case, it is often required for the software to meet safety constraints even in the presence of hardware failures (ref 1). It can be very difficult to verify such requirements. Integration of fault-injection techniques with the automated testing approach described in this paper can be used to address this problem. For example, consider a system that is required to meet all of its safety constraints even in the presence of single-bit stuck-at faults in the memory. By using fault injection to simulate these stuck at faults during the dynamic search for test-data that violates a specified safety constraint such requirements can be tested. Any test-data located would indicate that the system is not safe when stuck-at faults exist in the system's memory. Of course, failure of the test-data generator to locate data does not allow the conclusion that the system is safe in the presence of such faults. However, the failure of the aggressive targeted search for test-data does give a level of confidence is tolerant to the specified faults.

Reuse

Software reuse and the integration of components are important areas where safety problems can often found (ref 1). When reusing components it is vital that the environment within which they are reused meets the assumptions that were made during the

component development. The same is true for integration of components. If these assumptions are not met then the reused or integrated component may not function correctly and indeed may no longer be safe. At the point of the call to the reused or integrated component all possible input-data must meet the pre-condition of the component. Again the automated test-data generation framework can be extended to address the search for counter-examples. The goal here is to find test-data that causes the pre-condition to be false at the point of call. The cost-function remains basically the same as before, however now instead of safety or exception condition instrumentation, at the point of call to the reused or integrated component the truth of its pre-condition is monitored. This combined with the branch condition instrumentation guides the search to test-data that executes the call of the component, but also breaks the pre-condition.

Evaluation

The work on integrating an automated test-data generation with SFTA is in its early stages and therefore only preliminary results are available. This work is based on earlier specification testing research (see ref 9). In the case of safety property testing, safety conditions obtained from SFTA are used in place of full functional specifications. The results obtained using specifications have shown the approach to be extremely effective as a pre-proof step.

Application of the testing technique to testing exceptions has been carried out using the software for a civil aircraft engine. The software is approximately 200,000 lines of safety critical code written in SPARK-Ada (ref 12). First, the SPARK Examiner (ref 13) was used to extract the run-time verification conditions from the source-code. The Examiner is a static analysis tool that is able to extract proof obligations, which if discharged confirm the absence of software related run-time exceptions.

The SPARK automatic simplifier was able to discharge 89% of the verification conditions automatically. This left 11% that required manual proof effort. Prior to attempting proofs the testing framework was targeted at generating test-data that illustrated conditions under which an exception would be raised. For a number of the possible exceptions test-data was generated that showed situations in which the exception

would be raised. A detailed investigation into these situations showed that violation of the run-time rules was not actually possible in the system.

The use of protected arithmetic operators that are well defined in the presence of divide-by-zero, overflow and underflow prevented a number of the possible exceptions. Also the physical values returned from sensors and global configuration data protected against other possible exceptions. However, the test-data is still useful in these situations as it can be used to form the basis of a code review. Any changes to the protected operators, physical values returned by sensors or global configuration data would need to be investigated to examine if the exception protection had been disturbed.

Conclusions

The key to successful verification of safety is intelligent integration of dynamic and static techniques. Testing alone can never measure or assure safety. Neither are static techniques sufficient on their own, as they depend on mathematical analysis of models. The accuracy of these models in relation to the constructed system must also be verified. What has been presented here is a technique for such integration. An automated framework is targeted at generating test-data that aids in the process of applying static analysis techniques, such as software fault tree analysis.

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 illustrating a safety violation does not indicate there is not a safety problem, 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. As the test-data generation approach is completely automated this confidence can be gained cheaply. Confidence is gained by the failure of the test-data generation to find counter-examples illustrating safety-violations, exceptions being raised or unsafe reuse, prior to investing the time and money necessary to continue with static techniques.

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 check by other means. This is important as the algorithms used are stochastic and it is extremely difficult to reason about their application to arbitrary code. The early results from the approach are extremely encouraging and justify further investigation.

Acknowledgements

This work was funded by grant GR/L42872 from the Engineering and Physical Sciences Research Council (EPSRC) in the UK as part of the CONVERSE project.

References

1. Nancy G. Leveson, Safeware – System Safety and Computers, Addison-Wesley, 1995.
2. Nancy G. Leveson and Peter R. Harvey – Analyzing Software Safety. IEEE Transactions on Software Engineering, SE-9(5). Page 569-579. September 1983.
3. ISO/IEC 8652:1995 – Ada 95 : Language Reference Manual. 1995.
4. Stephen J. Clarke and John A. McDermid – Software fault trees and weakest preconditions: a comparison and analysis. Software Engineering Journal, Volume 8, Number 4. Pages 225-236. July 1993.
5. Nancy G. Leveson and Peter. Harvey – Software Fault Tree Analysis. Journal of System and Software, Volume 3, Pages 173-181. June 1983.
6. S. Kirkpatrick, C. Gelatt and M. Vecchi – Optimization by Simulated Annealing. Science, Volume 220. Pages 671-680. May 1983.
7. J. H. Holland – Adaptation in Natural and Artificial Systems. University of Michigan Press. 1975.
8. Nigel Tracey, John Clark and Keith Mander – The Way Forward for Unifying Dynamic Test Case Generation: The Optimisation-Based Approach. In the proceedings of the 13th IFIP International Workshop on Dependable Computing and Its Applications (DCIA). Pages 169-180. South Africa, January 1998.

9. Nigel Tracey, John Clark and Keith Mander – Automated Program Flaw Finding using Simulated Annealing. In the proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). Pages 73-81. Florida, USA, March 1998.

10. Nigel Tracey, John Clark, Keith Mander and John McDermid – An Automated Framework for Structural Test-Data Generation. In the proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE). Hawaii, USA, October 1998.

11. Nancy G. Leveson and Stephen S. Cha – Safety Verification of Ada Programs Using Software Fault Trees. IEEE Software. Pages 48-59. July 1991.

12. John Barnes – High Integrity Ada: The SPARK Approach. Addison-Wesley, 1997.

13. Praxis Critical Systems – SPARK Ada Documentation – Release 4.0. December 1998.

Biographies

Nigel J. Tracey, Research Associate, High Integrity Systems Engineering Group, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, telephone – +44 1904 432749, facsimile – +44 1904 432767, e-mail – njt@cs.york.ac.uk.

Nigel Tracey is a research associate at the University of York. His research interests include test automation and the testing safety-critical real-time systems software.

John Clark, Safety Critical Systems Lecturer, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, telephone – +44 1904 433379, facsimile – +44 1904 432767, e-mail – jac@cs.york.ac.uk.

John Clark specialises in advanced verification techniques and design synthesis. Current work includes the application of heuristic search techniques for the systemsis of secure protocols and secure distributed architectures.

John McDermid, MA, PhD, CEng, FBCS, FIEE, FRAeS, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, telephone – +44 1904 432726,

facsimile – +44 1904 432708, e-mail – jam@cs.york.ac.uk.

John McDermid was appointed to a chair in Software Engineering at the University of York in 1987 where he heads the High Integrity Systems Engineering Group (HISE). Professor McDermid is the Director of the Rolls-Royce funded University Technology Centre (UTC) in Systems and Software Engineering and the British Aerospace funded Dependable Computing Systems Centre (DCSC)

Dr. Keith Mander, Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK, telephone – +44 1904 43???, facsimile – +44 1904 432767, e-mail – mander@cs.york.ac.uk.

Keith Mander has been head of the Department of Computer Science since 1992. Current research interests include integration of structured and formal methods of system specification, process improvement and software testing.