FORTEST: Formal Methods and Testing

Jonathan P. Bowen
South Bank University, Centre for Applied Formal Methods
SCISM, Borough Road, London SE1 0AA, UK
Email: *bowenjp@sbu.ac.uk* URL: *www.cafm.sbu.ac.uk*

Kirill Bogdanov
The University of Sheffield, UK
Email: *k.bogdanov@dcs.shef.ac.uk*

John A. Clark
The University of York, UK
Email: *jac@cs.york.ac.uk*

Mark Harman
Brunel University, UK
Email: *mark.harman@brunel.ac.uk*

Robert M. Hierons
Brunel University, UK
Email: *rob.hierons@brunel.ac.uk*

Paul Krause
Philips Research Laboratories &
University of Surrey, Guildford, UK
Email: *p.krause@surrey.ac.uk*

URL: *www.fortest.org.uk* *

Abstract

Formal methods have traditionally been used for specification and development of software. However there are potential benefits for the testing stage as well. The panel session associated with this paper explores the usefulness or otherwise of formal methods in various contexts for improving software testing. A number of different possibilities for the use of formal methods are explored and questions raised. The contributors are all members of the UK FORTEST Network on formal methods and testing. Although the authors generally believe that formal methods are *useful in aiding the testing process, this paper is intended to provoke discussion. Dissenters are encouraged to put their views to the panel or individually to the authors.*

1. Introduction

Formal methods and testing are sometimes seen as adversaries. It has been said that formal methods could eliminate testing. In practice, however, formal methods and testing will always be two complementary techniques for the reduction of errors in computer-based systems since neither technique is perfect in practice for realistic systems. When formal methods are used in development, it is still very important to undertake testing, even if the amount of testing can be reduced [9]. It could be considered unethical not to apply both techniques in systems involving the highest levels of criticality where human lives may be at risk [8]. However, the potential symbiosis of formal method and testing is still be in its infancy. This paper presents some ideas of future directions in the interplay of formal methods and testing.

The panel session associated with this paper presents the views of a number of participants on the UK EPSRC FORTEST Network concerning the interplay of formal methods and software testing. It is contested that the presence of a formal specification is beneficial in the determination of test cases for software-based products. In this context, the topics to be covered to promote discussion include aspects of the following (in no particular order):

- Blackbox testing
- Fault-based and conformance-based testing
- Theory versus practice
- Design and refinement for testing
- Testing in support of specification
- Testability transformation

*The FORTEST Network on formal methods and testing, of which all the authors are members, is funded by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant number GR/R43150. Full contact details for all the authors can be found on the website.

Proceedings of the 26 th Annual International Computer Software and Applications Conference (COMPSAC'02)
0730-3157/02 $17.00 © 2002 **IEEE**

- Using system properties
- Automated generation of test cases

FORTEST partners include leading academic and industrial sites in the UK with an interest in the application of formal methods to software testing. The Network, led by Brunel University, is running from November 2001 for three years. The academic partners consist of Brunel University, University of Kent at Canterbury, The University of Liverpool, University of Oxford, The University of Sheffield, South Bank University, University of Surrey and The University of York. The industrial partners are DaimlerChrysler, Philips Research Laboratories, Praxis Critical Systems Ltd, QinetiQ and Telelogic.

1.1. Background to the Network

With the growing significance of computer systems within industry and wider society, techniques that assist in the production of reliable software are becoming increasingly important. The complexity of many computer systems requires the application of a battery of such techniques. Two of the most promising approaches are formal methods and software testing. FORTEST is a cross-community network that brings together expertise from each of these two fields.

Traditionally formal methods and software testing have been seen as rivals. Thus, they largely failed to inform one another and there was very little interaction between the two communities. In recent years, however, a new consensus has developed. Under this consensus, these approaches are seen as complementary [29]. This opens up the prospect of collaboration between individuals and groups in these fields.

While there has already been some work on generating tests from formal specifications and models (e.g., see [27, 31, 49, 50] in the context of the Z notation). FORTEST is considering a much wider range of ways in which these fields might interact. In particular, it is considering relationships between *static testing* (verification that does not involve the execution of the implementation) and *dynamic testing* (executing the implementation).

FORTEST has formed a new community that is exploring ways in which formal methods and software testing complement each other. It is intended that this will allow these fields to inform one another in a systematic and effective manner and thus facilitate the development of new approaches and techniques that assist the production of high quality software.

The rest of this paper presents the views of a number of FORTEST Network members on various ways in which formal methods could be helpful for software testing.

2. Three Challenges in Blackbox Testing

Kirill Bogdanov, The University of Sheffield

2.1. Fault-based and conformance-based testing

Functional testing (rather than stress testing, usability, etc.) has been traditionally viewed as an approach to find faults by exercising a system, using systematic but qualitative techniques, aimed at coverage of a specification. The most often used approach is input-space partitioning and boundary-testing [17, 41]. While the way decisions are made in construction of partitions can be based on business risk, the essential conclusion is "we tried but did not find much."

Mutation testing and analysis makes it possible to try to check if a test suite finds specific classes of failures [40]; mutations could apply to both a specification and an implementation. One could either generate a test suite manually to 'kill' all mutants or use a technique (such as model-checking) to identify differences between an original and a mutated model [1, 21]. At the same time, one could derive test suites from a model in order to demonstrate behavioural equivalence of an implementation to this model by testing, for instance, using well-known automata-theoretic results [14, 46, 47] and their extensions to X-machines (EFSM) [30, 32] and Statecharts [5, 6]. Testing using relations from algebras [20, 45] can also be considered to be of this type of testing. Is testing using fault-based methods 'better' than conformance-based testing? Fault-based approaches are based on the generation of all possible faults of a given kind (perhaps an infinite amount) and finding tests to expose them. Conformance-based testing is based on building 'relevant' models and attempts to generate tests to expose all deviations from them. In the former, one could be specific about faults, but it could be more difficult to derive tests for them; in the latter, the types of faults testing looks for are not so easily tailored to particular business goals.

2.2. Design for test – a guide for software engineering?

It is typically said that semiconductors are built to include mechanisms to facilitate testing, and that software is often not. From that, one can elaborate as to how to build software so as to make it easier to integrate it with test harnesses (like separating a graphical interface from underlying engine). Curiously, built-in self-testing for chips is generally geared to identifying manufacturing defects, not design ones while for software all defects are design defects[1].

[1] [3] claims that usage of state-based testing was rather more effective in terms of state and transition coverage of an actual device per test than using standard 'engineering' test suites.

Proceedings of the 26 th Annual International Computer Software and Applications Conference (COMPSAC'02)
0730-3157/02 $17.00 © 2002 **IEEE**

There has been quite a bit of work done in the direction of making software more testable, still leaving many questions unanswered.

Is more testable software likely to fail more often [4]? If some of the testing code is left in, it can be invoked (accidentally or maliciously). If defects easily manifest themselves, they are easier to detect during development but could be annoying to users.

Structuring software according to testability. In general, metrics, such as McCabe cyclomatic complexity [19] can be used to make a suggestion how much a particular module would cost to build and perhaps how difficult it will be to test. One of the quotes emphasizing testability is "if you cannot test it – you cannot build it." It is possible to suggest testability as a criterion to help structuring a system, for instance,

- A particular part of a system is a unit if one can visit every internal state of it and attempt any transition from that state without having to access that part's internal data. This means we are suggesting against relying on helper modules with privileged access to internal data [38].

- In Statechart-based testing, one has to consider testing for presence and absence of all possible transitions from every state of a system. One could partition the system such that transitions from some parts of it cannot be erroneously used in other parts, reducing the size of a test suite by many orders of magnitude [5].

Testability could also guide a style of a specification (see section 3.4).

2.3. 'The devil is in the detail' – is testing from an abstract specification useful?

Do we need model-based testing if we can directly generate code from models and claim it correct by construction? There are in fact three separate issues here.

While usage of a certified compiler could be enough not to require certification of an object code, one has still to certify a model from which that code is generated. Such a model is often extensively simulated (for instance, by using testing [35]) and critical properties (such as freedom from deadlocks) are formally proven or demonstrated by model-checking [11]. Indeed, checking of models for specific properties is akin to fault-based testing mentioned in section 2.1. Here it is implicitly assumed that developers of a model know 'everything' while [37] demonstrated that a significant proportion of errors was in the software-hardware interfaces, resulting from lack of communication between the two teams of developers.

Testing is still needed as one clearly cannot model everything. For instance, if a memory chip is capable of storing all possible values indefinitely except for one value (such as due to noise on power lines), this cannot be found from a gate-level model. It is thus interesting to compare model-based testing methods in terms of their ability to find problems which cannot not be described by models from which tests are generated.

Testing from a high-level specification could be considered ineffective, because of a wide gap in abstraction between a specification and design. In a sense, this is handled by 'inconclusive' test results [20]; [6] reports that testing only covered a part of an implementation. Thus the question is what to do with the implemented behaviour which was not specified (one can argue that a failure of the Ariane 5 space rocket was related to an extra piece of code running which should have been disabled).

3. Formal Methods and Testing: What Happens and What Should

John A. Clark, University of York

Effective and efficient development requires wide-ranging integration of formal *and* non-formal techniques. there is much work done already but there is considerable potential for further growth.

To support this statement I shall now provide a partial categorisation of current work concerning formal methods and testing and indicate some issues arising in each.

3.1. Formal methods as testing

Some things are just better done formally. An obvious example would be testing whether security protocols can be attacked by a malicious party with access to the medium. The errors uncovered by current model checking approaches would most likely never be found by other testing means. For example, seventeen years after publication and after countless citations, a very well-known protocol was shown (by model checking) to contain an error [36]! More generally, various 'negative' properties are obvious candidates for using a fully formal approach. Some things that can be proved, should be. Dynamic testing sometimes has little to offer. Conversely, can we agree where formal approaches are lacking? More generally, a more detailed and explicit understanding is needed of what works best, where, and why.

3.2. Formal methods in support of testing

There is a considerable amount of work here, most usually termed "testing from specifications." A typical approach would be partition extraction on a model-based specification, solution of the constraints that arise, and refining the results to program test cases. Also the *test (generate) and check* paradigm would seem significantly underrated and under-used. One often cited problem concerning such techniques based is a marked reluctance of engineers to write formal descriptions. Approaches that map automatically from a "safe subset" of engineering-friendly graphical notations (e.g., Statecharts) to provide a formal and processable intermediate description (e.g., using the Z notation) are a promising avenue to follow [13, 12]. The principal challenge, and one we cannot ignore, would appear to be scalability.

3.3. Evolutionary testing in support of formal methods

If you want to prove something, it helps if it is actually true! Directed testing (e.g., via evolutionary approaches [53]) can act as a means of targeted counter-example generation. Thus, claims about exception-freeness and other specific safety properties might first be verified via guided dynamic search [42] and only then subject to proof. This may well save (doomed) proof effort which can be applied elsewhere.

3.4. Specification/design/refinement and testing

The overall aim is to achieve risk reduction cost effectively. This requires an integrated approach to design and verification. You may well write a specification in a particular style. Similarly, you may write a specification in a style you know a particular proof tool handles well. How should you specify for automatic test case generation? How should low-level testability issues affect specification and refinement style [16]?

3.5. Testing in support of specification

Specifications may be synthesised from test data by hypothesising a great number of possible assertions about the program state at various points and seeing which are invalidated by test data [18]. What remains is a 'specification.' Standard black-box and structural testing together with targeted counter-example generation [51] can support the creation of better formal approximations as specifications. This notion extends further. Various non-standard systems (e.g., neural networks) can now 'explain' their behaviour to the user. This is just another description we might like to find a counter-example to. Should we be more flexible in what we consider to be a specification? Are we making unhelpful and over-restrictive assumptions about what constitutes a 'system'?

3.6. Mathematical methods and testing

Do we distinguish too heavily between 'formal methods' (things done by the "formal methods community") and the mathematics used by mathematicians and engineers more generally? Can/should the current FORTEST group embrace more statistical concepts and links with numerical analysis, control theory and information theory? Much mathematics works! I think we should embrace it.

And finally, let us look ahead. The future may be nanotechnology! What are we doing to address the rigorous specification of nano-system properties, their rigorous refinement and the 'testing' of the various system descriptions that arise?

4. Testability Transformation

Mark Harman, Brunel University

Testing is increasingly recognised as a crucial part of the software development process and one which is important for almost all forms of software. Unfortunately, testing is also a hard problem, not least because of the difficulty in automating the identification of high quality test data.

In this section we describe, in overview, a new approach to transformation, in which programs which are hard to test are transformed into programs which are easier to test. This is a simple idea, but it involves a radical new approach to transformation. The transformations used need not preserve the traditional (functional equivalence) meaning of programs. Neither do we propose to keep the transformed program's; we only need them to help us generate test data.

Because *testability transformation* alters the program, we may also need to co-transform the test adequacy criterion against which we are testing. This allows us to shift the problem of testing an original program with an original criterion to the transformed problem of testing the transformed program with respect to the transformed criterion. Therefore, Testability Transformation is an example of an approach in which a hard problem is reformulated to a new problem which, though easier to solve, retains the properties of interest pertinent to the original, harder problem.

In the rest of this section, we shall look at problems for branch coverage, but the ideas presented here can be applied far more widely than this. Branch coverage criteria require that a test set contains, for each feasible branch of the program under test, at least one test case which causes execution to traverse the branch. Unfortunately, automatically

generating test data which have this property is not generally computable, and so research and development have focused upon techniques which aim to optimise test data with respect to such a criterion.

4.1. The problem

Generating branch coverage adequate test data is often impeded by the structure of the program under test. This section illustrates this by considering problems for *evolutionary* testing [34, 44, 52, 56], but the problem also applies in other approaches to white box test data generation.

Evolutionary Testing uses *metaheuristic* search based techniques[2] to find good quality test data. Test data quality is defined by a test adequacy criterion, which underpins the fitness function which drives the search implemented by the genetic algorithm.

To achieve branch coverage, a fitness function is typically defined in terms of the program's predicates. It determines the fitness of candidate test data, which in turn, determines the direction taken by the search. The fitness function essentially measures how close a candidate test input drives execution to traversing the desired (target) path.

Generating test data using evolutionary test data generation has been shown to be successful, but its effectiveness is significantly reduced in the presence of programming styles which make the definition of an effective fitness function problematic. For example:

- The presence of side effects in predicates reduces the ability to exploit the inherent parallelism in a predicate's syntactic structure.
- The use of flag variables (and enumeration types in general) creates a coarse fitness landscape, thereby dramatically reducing the effectiveness of the search.
- Unstructured control flow (in which loops have many entry and exit points) affects the ability to determine how alike are the traversed and target paths.

The presence of these features make a program less 'testable.'

4.2. The solution

When presented with problems of programming style, a natural solution is to seek to transform the program to remove the problem. However, there is an apparent paradox:

> Structural testing is based upon structurally defined test adequacy criteria. The automated generation of test data to satisfy these criteria can be impeded by properties of the software (for example, flags, side effects, and unstructured control flow). Testability transformation seeks to remove the problem by transforming the program so that it becomes easier to generate adequate test data. However, transformation alters the structure of the program. Since the program's structure is altered and the adequacy criteria is structurally defined, it would appear that the original test adequacy criterion may no longer apply.

Testability transformation therefore requires *co-transformation* of the adequacy criterion; this avoids the apparent paradox. An (informal) definition of a testability transformation is given below:

Definition 1 *Testability Transformation*
A *testability transformation* maps a program, p and its associated test adequacy criterion, c to a new program p' and new adequacy criterion, c', such that any set of test data which is adequate for p' with respect to c' is also adequate for p with respect to c.

Observe that, while traditional transformations are meaning preserving functions on programs, testability transformations are 'test set preserving' functions on pairs containing both a program and its associated adequacy criterion.

4.3. Heresy?

Testability transformation is novel, both in its application of transformation to the problem of automated test data generation and in the way in which it becomes necessary to change the traditional view of transformation in order to achieve this:

- **Disposable Transformation**
  Program transformation has previously been regarded as an *end* in itself, rather than merely as a *means* to an end. Hitherto, the goal of transformation research has been to transform poor programs into better ones. Thus research has tended to assume that the original program will be discarded once the transformed program has been constructed.

  By contrast, with the Testability Transformation approach, it is the transformed program which is discarded and the original which is retained. Testability transformation requires the transformed program solely for the purpose of test data generation. Definition 1 guarantees that such test data will be adequate

[2] Typically genetic algorithms and simulated annealing have been used, but we require only that the technique used is characterised by some fitness (or cost) function, for which the search seeks to find an optimal or near-optimal solution.

for the original. Once the test data is generated, the transformed program is no longer required.

- **Non Meaning-Preserving Transformation**
  Program transformation has traditionally been concerned with the preservation of functional equivalence. However, using the Testability Transformation approach, the transformed program must provide a wholly different guarantee; that test data generated from it is adequate for the original program. This means that the transformations applied need not preserve any notion of traditional equivalence, opening the possibility of a novel set of program transformation rules and algorithms.

4.4. Future Work

There are many ways in which we might pursue a research agenda which exploits the idea of transformation to improve testability. These could be characterised as follows:

- **Theoretical Foundations**
  The concept of transformations which preserve only program behaviour with respect to testing concerns will require a reformulation of the semantic relations to be preserved by transformation. Such a semantics could be captured using abstract interpretation [15].

- **Algorithmic Development**
  Existing work transformation to remove side effects [26] and to restructure programs [2, 48] may be reused or adapted for testability transformation. However, Testability Transformation will require new and radically different transformation algorithms. These new transformations need not preserve the meaning of a program, in the traditional sense. For instance, existing transformation algorithms would never transform the program `if(` E `)` S_1 `else` S_2 into `if(` E `)` S_2 `else` S_1.

  Such a transformation would clearly not be meaning preserving. Nonetheless, this transformation *does* preserve branch coverage and so is a valid testability transformation for branch coverage preservation.

- **Evaluation**
  Clearly any new approach requires evaluation before wider adoption. Initial work on flag removal transformations for evolutionary testing [25] has provided a proof of concept, demonstrating the improvement in test generation time and test data quality for branch coverage after flag removal. However, given the scope and size of the problem addressed by Testability Transformation, these results represent only the initial confirmation that the idea is sound.

- **Extensions**
  This section has considered the way in which transformation might benefit testing and evolutionary testing in particular. However, the idea may also be applicable for other forms of testing, for example mutation based testing and constraint based test data generation. The idea of a transformation which preserves only a property of interest, rather than the traditional semantics of the program may also find other applications. Traditionally, such a view has guided analysis (through abstract interpretation). It is hoped that testability transformation will be one in a sequence of many applications in which abstract interpretation can be used to guide *transformation* as well as analysis.

5. The Utilization of System Properties in Testing

Robert Hierons, Brunel University

While many test techniques are eminently sensible and extremely useful, often they have no real theoretical basis. It is thus difficult to say what we have learnt if a piece of software passes our test. This has encouraged the view that software testing is an imprecise process that is more of an art than a science. Here we will argue that test hypotheses and fault models may be used to overcome these problems and that there are a number of interesting open questions relating to this area.

5.1. Test hypotheses, fault models, and test that determine correctness

In order to introduce more formality into testing, the notions of fault models and test hypotheses have been introduced. A fault model [33] is a set Φ of behaviours with the property that the tester believes that the implementation under test (IUT) behaves like some unknown element of Φ. It may be possible to produce a test T that is guaranteed to determine correctness under this assumption: T will lead to a failure for every faulty elements of Φ. A similar notion is a test hypothesis [22]: a property that the tester believes that the IUT has. Given test hypothesis H it may be possible to find a test that determines correctness under the assumption that H holds.

Naturally, test hypotheses and fault models are related properties. Given test hypothesis H, there is a corresponding fault model: the set of behaviours that satisfy H. Given fault model Φ there is an associated test hypothesis: that the IUT behaves like some unknown element of Φ.

Fault models are largely met within the areas of protocol conformance testing and hardware testing. Here the specification M is typically a finite state machine (FSM) and a

fault model is some set of FSMs with the same input and output sets as M. One such fault model is the set of FSMs with the same input and output sets as M and no more than m states, for some predefined m. Since this fault model is finite and equivalence of FSMs is decidable, correctness is decidable, through black-box testing, under this fault model [39].

While the work on test hypotheses originated in the field of testing from algebraic specifications (see, for example, [7, 22]), it may be argued that test hypotheses lie behind many software test techniques. The classic example is the uniformity hypothesis used in partition analysis (see, for example, [23]). Here it is assumed that the behaviour of the IUT is uniform on some region of the input domain and thus that it is sufficient to take a small number of values from this region. Typically such tests are augmented by tests around the boundaries – an admission that the tester is not confident that the uniformity hypothesis holds.

5.2. Comparing test techniques

Currently most theoretical results, regarding the relative strengths of test criteria, relate to the subsumes relation. Here test criterion C_1 subsumes test criterion C_2 if and only if every test set that satisfies C_1 also satisfies C_2. While many test criteria are comparable under subsumes [43], the subsumes relation says nothing about the ability of test sets and criteria to determine whether the IUT contains faults. Given criteria C_1 and C_2, such that C_1 subsumes C_2, there is no guarantee that if some test that satisfies C_2 finds a failure then every test that satisfies C_1 finds a failure (see [24] for a critique of the subsumes relation).

It has been observed that often the following comparison would be more useful [24]: test criterion C_1 is at least as strong as test criterion C_2 if whenever some test set that satisfies C_2 determines that the IUT is faulty, every test set that satisfies C_1 determines that the IUT is faulty. This is written $C_2 \leq C_1$. However, since test criteria do not usually exclude any test input, given faulty IUT I, every practical test criterion is able to determine that I is faulty. Further, in general no practical test criterion will produce tests that are guaranteed to determine that I is faulty. Thus, practical test criteria are not comparable under $\leq$ [24].

In addition to the above limitations, the only relation between test sets is set inclusion: test set T_1 is at least as strong as a test set T_2 if and only if $T_2 \subseteq T_1$.

It thus seems that there is no way of producing strong, general comparisons between test criteria and test sets. However, in testing, the tester is not interested in general comparisons: they are interested in using a 'good' test set or criterion for the system they are currently testing. Thus, methods for comparing test sets and criteria might utilise system properties, possibly represented as either test hypotheses or fault models. Given a fault model or test hypothesis, there may be relationships between the effectiveness of particular test sets or criteria that do not generally hold.

Suppose, for example, that the input domain D of the IUT I has been partitioned into a set P of sub-domains. Suppose, further, that the uniformity hypothesis is made for each sub-domain $D' \in P$. Under this test hypothesis, a test set T_1 is at least as strong as a test set T_2 if and only if for all $D' \in P$, if T_2 contains an element from D' then T_1 also contains an element from D'. Assuming the test hypothesis holds, we know that if T_2 finds a failure then T_1 is guaranteed to find a failure. Test set T_1 being at least as strong as test set T_2 under hypothesis H might be written $T_2 \leq_H T_1$.

Observe that in general, $T_2 \leq_H T_1$ does not imply that $T_2 \subseteq T_1$ but that $T_2 \subseteq T_1$ does imply that $T_2 \leq_H T_1$. Thus $\leq_H$ is a weaker comparator than $\subseteq$.

This type of comparison may also be used to drive test generation: it is only worth extending a test set T by a test case t if $T \cup \{t\} \not\leq_H T$: the test set $T \cup \{t\}$ is capable of identifying faulty implementations that T cannot.

This type of relation might be extended to test criteria: test criterion C_1 is at least as strong as criterion C_2 under hypothesis H if for every test set T_1 satisfying C_1 and test set T_2 satisfying C_2, $T_2 \leq_H T_1$.

Where a test criterion C corresponds to testing to determine correctness under the test hypothesis H used, it is immediate that for every test criterion C', $C' \leq_H C$. Further, in one extreme case, where the hypothesis H_{corr} is that the implementation is correct, all criteria are comparable under $\leq_{H_{corr}}$. An open question is: are there other types of test criteria that are comparable under some test hypothesis H?

5.3. Issues

While test hypotheses and fault models allow the tester to reason about test effectiveness, the tester is left with the problem of deciding which assumptions to make. If the assumptions made are too strong and the IUT does not satisfy these, the results based on the assumptions may not hold. If the assumptions made are weaker than necessary then the tester may not be able to utilise properties that are of value.

Based on the above, it may thus be argued that one of testing research's major challenges is to find system properties, in the form of fault models or test hypotheses, such that:

- the system properties are likely to hold;
- the system properties are relatively simple to formally verify;

- the system properties assist the tester, either in producing a test that determines correctness or in reasoning about test effectiveness.

Naturally such test hypotheses and fault models are likely to be domain specific. An interesting question is: are there problem domains with common sets of system properties that satisfy the above conditions?

6. Test Automation

Paul Krause, University of Surrey & Philips Digital Systems Laboratory

Where do we stand with test automation? Let us take a positive view to begin with. What are the fundamental problems with software testing, and do we seem to be near a solution?

What are the problems? Well, for any complex software product testing is time consuming and resource hungry. In addition, it is error prone. Errors may be both faults of commission (the test plan or test cases may contain an incorrect assumption about the behaviour of the intended product) and faults of omission (the test plan may fail to adequately test certain aspects of the behaviour of the intended product). Test automation, both in terms of generation of tests, and their execution, seems to offer a panacea. So, where are we now?

For many classes of application, automated execution of test cases is becoming a routine activity. Indeed for some projects, it is essential in order to obtain the required degree of coverage within a limited budget. Most commercial test execution tools are designed for graphical user interface based testing. However, tools for non-intrusive testing of embedded software are now entering the market place. In this case we are interested in stimulating the software through simulating user actions, and then checking the response of the software by capturing the feedback to the user (usually via some character or image verification of an LCD or other video display). This is looking good. There is promise here to remove the tedium and expense of manual execution of test cases in favour of machine execution.

What about generating the test cases? There have been many advances in generating test cases from specifications. Test generation from formal specifications, such as Z or VDM, is now well covered [27, 31, 49, 50]. But the (apparently) less formal, although perhaps more widely used, specification techniques such as UML Sequence Charts, Statecharts and Cause-Effect Graphing also have sound and automatable test case generation techniques associated with them [28]. With the availability of techniques for automated generation of test cases from specifications, and automated execution of test cases, it may seem we are close to realising a dream of "one-button testing" of software products. However, there are a number of problems that still need to be resolved. Let us take a simple Use Case from an embedded application by way of illustration. This is a hypothetical Use Case for software embedded in a television, and elicits some of the functional requirements needed to support manual installation. The details of the behaviour will not correspond to any specific product – they are purely for illustration of the principles, although based on a currently available product.

Use Case: Manual Installation

1. This Use Case begins when the user selects the Manual Installation Sub-Menu
2. The Current Program Number is set to 1
3. The User first selects the System from UK, Europe or ??
4. The User may now set the Tuner to search for an Off-air Channel
5. The User may not interrupt the search operation until the Tuner has identified a stable signal
6. When the Tuner has identified a stable signal, the frequency of that signal shall be displayed
7. The User may accept the current value of the Program Number, or increase it if they wish to skip one or more Program Numbers
8. The User may now store the Tuner Frequency to be indexed by their chosen Current Program Number
9. Following a time-out of 3 seconds, the Current Program Number shall be incremented and control will return to Step 4
10. Other than while the Tuner is searching (Step 5) the User may exit this Use Case at any point

A number of options are available to progress development of an analysis model at this point. Rather than begin to refine this into a more detailed specification, we will just consider three approaches that could be used almost straight away as an aid to generating test cases. A more detailed and precise description of the behaviour of the product could be obtained by expressing this Use Case as a Sequence Chart, a Statechart or a Cause-Effect Graph. There are well-defined techniques for generating test cases from each of these.

This sounds quite promising now. We can re-express this Use Case in one of a number of languages each with a well-defined syntax. From any of these, we can generate test cases in an executable script. Seemingly we are there. Pass

the scripts to a test execution environment and then sit back and wait for the results? Unfortunately, there are a number of issues that still need to be resolved.

The first thing to notice is that the Use Case is written with a deliberate abstraction away from the details of the user interface. For example, although we state that the "Use Case begins when the user selects the Manual Installation Sub-Menu," we do not state at this point whether this Sub-Menu is selected using the local key-board, a remote control or (perhaps) a voice command. Neither do we express the precise sequence of commands needed to get to this point from some preferred starting point. This is a deliberate and often used policy. In the above example, the functionality described has stayed essentially unchanged across several generations of products. However, the details of the user interactions can stay quite volatile even up to the late stages of product development, so a recommended practice is to keep the user interactions separate from the logical events. This means that if we are to be able to automatically execute these tests non-intrusively against embedded software, then we need some way of automatically mapping the logical events (e.g., "select the Manual Installation Sub-Menu") onto a physical sequence of user interactions (send a *Menu* command, send three *Cursor Down* commands, send *Cursor Right* to select the Installation Sub-Menu...).

A second problem is controlling the number of test cases that are generated even for this relatively minor aspect of the product's behaviour. A television set for the European market may have as many as 125 potential channel numbers. Taking into account the scope for skipping channels or not, this implies a massive combinatorial explosion if we are to consider all possible scenarios (paths through the Use Case). How do we identify an automatable policy for selecting out a sufficient sub-set of all these scenarios that is executable in a finite time? The problem becomes fantastically complex if we consider combinations of test cases (e.g., how does channel change perform when we have previously installed two channels, three channels... 125 channels? And should we perform each of these combination tests for all possible pairwise channel changes?).

A third test generation problem is that we are at the moment only considering testing functionality. With complex consumer products a significant percentage of the field call rate is due to non-functional/performance issues. In the above example, we might have tested tuning speed against some benchmarks. But does tuning speed degrade as more channels are installed? Does it degrade upon repeated exiting and restarting of the Manual Installation Use Case? Is it possible to identify techniques or heuristics for identification of non-functional stress and performance tests?

These problems can be summarised as follows:

1. The number of test cases generated increases exponentially with the complexity of a product. For most software systems, executing a complete set of automatically generated test cases at the system level is not feasible. How do we select a finite subset of test cases without significantly impacting on the quality guarantees for the product?

2. For many product families, a clear separation of concerns is maintained between specification of functionality, and specification of the User Interface. How do we specify the mapping between logical events in the functional specification and physical events in the user interface specification so that the complete test descriptions can be generated for non-intrusive testing of embedded software?

3. Tools for automated execution of test cases are primarily sold as aids for testing functionality. Yet many of the issues that are raised in field calls are non-functional. What is industry's experience with automated testing of non-functional requirements?

My position is that these issues need to be soundly resolved before the full potential of automated testing can be realised. Indeed, the problem is even worse than this. I have omitted at least two functions in the above Use Case. On many televisions, once a stable signal has been identified by the Tuner, Automatic Frequency Control (AFC) is enabled for that Channel to follow the signal in case of any frequency drift. Alternatively, the User can disable AFC and manually fine tune to obtain an optimum signal. Never mind about the details. The important issue is that I failed to identify, or at least articulate, all that was relevant to this Use Case. So even if we can solve the above three problems satisfactorily, the confidence we can gain from automated generation and execution of tests will still be bounded by the confidence we have that all the relevant requirements have been identified – and indeed by the confidence that those we have identified are correct. Perhaps we need to learn that uncertainty is inherent in software?

7. Conclusion

The authors of this paper believe that there are benefits of applying formal methods to the software testing process in a number of ways. This paper has explored some of the possibilities, although not in an exhaustive manner. For example, formal methods could also be useful in the formalization of existing testing criteria [54] to help eliminate misunderstanding and in the precise formulation of new testing criteria [55].

The paper is intended to provoke discussion. We welcome both positive and constructive negative feedback on the ideas presented in this paper. The contact details of all

the authors and further information on the FORTEST Network can be found on-line under:

www.fortest.org.uk

References

[1] P. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. 2nd IEEE International Conference on Formal Engineering Methods (ICFEM '98)*, pages 46–54, Brisbane, Australia, Dec. 1998.

[2] E. A. Ashcroft and Z. Manna. The translation of `goto` programs into `while` programs. In C. V. Freiman, J. E. Griffith, and J. L. Rosenfeld, editors, *Proc. IFIP Congress 71*, volume 1, pages 250–255. North-Holland, 1972.

[3] M. Benjamin, D. Geist, A. Hartman, G. Mas, R. Smeets, and Y. Wolfsthal. A feasibility study in formal coverage driven test generation. In *Proc. 36th Design Automation Conference (DAC'99)*, June 1999.

[4] A. Bertolino and L. Strigini. On the use of testability measures for dependability assessment. *IEEE Transactions on Software Engineering*, 22(2):97–108, Feb. 1996.

[5] K. Bogdanov. *Automated Testing of Harel's Statecharts*. PhD thesis, The University of Sheffield, UK, Jan. 2000.

[6] K. Bogdanov and M. Holcombe. Statechart testing method for aircraft control systems. *Software Testing, Verification and Reliability*, 11:39–54, 2001.

[7] L. Bouge, N. Choquet, L. Fibourg, and M.-C. Gaudel. Test sets generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343–360, 1986.

[8] J. P. Bowen. The ethics of safety-critical systems. *Communications of the ACM*, 43(4):91–97, April 2000.

[9] J. P. Bowen and M. G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28(4):56–63, April 1995. Also in *High-Integrity System Specification and Design*, Springer-Verlag, FACIT series, pages 217-230, 1999.

[10] J. P. Bowen and M. G. Hinchey, editors. *ZUM'95: The Z Formal Specification Notation, 9th International Conference of Z Users, Limerick, Ireland, September 7–9, 1995, Proceedings*, volume 967 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[12] S. Burton, J. Clark, A. Galloway, and J. McDermid. Automated V&V for high integrity systems: A targeted formal methods approach. In *Proc. 5th NASA Langley Formal Methods Workshop*, June 2000.

[13] S. Burton, J. Clark, and J. McDermid. Testing, proof and automation: An integrated approach. In *Proc. 1st International Workshop of Automated Program Analysis, Testing and Verification*, June 2000.

[14] T. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, 1978.

[15] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.

[16] J. Derrick and E. Boiten. Testing refinements of state-based formal specifications. *Software Testing, Verification and Reliability*, 9:27–50, July 1999.

[17] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model based specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME '93: Industrial Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer-Verlag, April 1993.

[18] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[19] N. E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman & Hall, London, 1991.

[20] J.-C. Fernandez, C. Jard, T. Jeron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 348–359. Springer-Verlag, 1996.

[21] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. ESEC/FSE 99: Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, Toulouse, France, 6–10 September 1999.

[22] M.-C. Gaudel. Testing can be formal too. In P. D. Moses, M. Nielson, and M. I. Schaertzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, March 1995.

[23] J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, 1975.

[24] R. Hamlet. Theoretical comparison of testing methods. In *Proc. ACM SIGSOFT'89*, pages 28–37, 1989.

[25] M. Harman, L. R. M. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Genetic and Evolutionary Computation Conference (GECCO 2002)*, New York, USA, July 2002. AAAI. To appear.

[26] M. Harman, Lin Hu, Xingyuan Zhang, and M. Munro. Side-effect removal transformation. In *Proc. 9th IEEE International Workshop on Program Comprehension (IWPC'01)*, pages 310–319, Toronto, Canada, May 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[27] R. M. Hierons. Testing from a Z specification. *Software Testing, Verification and Reliability*, 7(1):19–33, 1997.

[28] R. M. Hierons, S. Sadeghipour, and H. Singh. Testing a system specified using statecharts and z. *Information and Software Technology*, 43(2):137–149, 2001.

[29] C. A. R. Hoare. How did software get so reliable without proof? In M.-C. Gaudel and J. C. P. Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1996.

[30] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution*. Springer-Verlag, Sept. 1998.

[31] H.-M. Hörcher. Improving software tests using Z specifications. In Bowen and Hinchey [10], pages 152–166.

[32] F. Ipate and M. Holcombe. An integration testing method that is proved to find all faults. *International Journal on Computer Mathematics*, 63:159–178, 1997.

[33] ITU-T. *Z.500 Framework on Formal Methods in Conformance Testing*. International Telecommunications Union, 1997.

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

[35] S. Liu. Verifying consistency and validity of formal specifications by testing. In J. Wing, J. Woodcock, and J. Davies, editors, *FM '99: Formal Methods, Volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 896–914. Springer-Verlag, September 1999.

[36] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer-Verlag, 1996. Also in *Software Concepts and Tools*, 17:93–102, 1996.

[37] R. R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, 1993. IEEE Computer Society Press.

[38] J. D. McGregor and T. D. Korson. Integrated object-oriented testing and development processes. *Communications of the ACM*, 37(9):59–77, Sept. 1994.

[39] E. P. Moore. Gedanken-Experiments. In C. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton University Press, 1956.

[40] L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, Aug. 1990.

[41] G. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

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

[43] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988.

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

[45] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Journal*, 19:53–77, 1997.

[46] A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Testing deterministic implementations from nondeterministic FSM specifications. In *Proc. 9th International Workshop on Testing of Communicating Systems (IWTCS'96)*, pages 125–140, 1996.

[47] T. Ramalingam, A. Das, and K. Thulasiraman. On testing and diagnosis of communication protocols based on the FSM model. *Computer communications*, 18(5):329–337, May 1995.

[48] L. Ramshaw. Eliminating goto's while preserving program structure. *Journal of the ACM*, 35(4):893–920, 1988.

[49] S. Stepney. Testing as abstraction. In Bowen and Hinchey [10], pages 137–151.

[50] P. A. Stocks and D. A. Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

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

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

[53] N. Tracey, J. Clark, J. McDermid, and K. Mander. A search-based automated test data generation framework for safety-critical systems. In *Systems Engineering for Business Process Change*, chapter 12, pages 174–213. Springer-Verlag, 2002.

[54] S. A. Vilkomir and J. P. Bowen. Formalization of software testing criteria using the Z notation. In *25th Annual International Computer Software and Applications Conference (COMPSAC 2001), Chicago, Illinois*, pages 351–356. IEEE Computer Society, October 2001.

[55] S. A. Vilkomir and J. P. Bowen. Reinforced condition/decision coverage (RC/DC): A new criterion for software testing. In D. Bert, J. P. Bowen, M. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 295–313. Springer-Verlag, January 2002.

[56] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001. Special issue on *Software Engineering using Metaheuristic Innovative Algorithms*.