Testing, Proof and Automation. An Integrated Approach

Simon Burton John Clark John McDermid
Department of Computer Science.
University of York,
Heslington, York.
YO10 5DD, England.
+44 1904 432749
{burton, jac, jam}@cs.york.ac.uk

Abstract

This paper presents a discussion on the complementary roles of testing and proof within automated software verification and validation processes. We demonstrate how a combination of the two approaches can lead to greater levels of automation and integrity. In particular we discuss the use of automated counter-example generation to support proof, and automated proof as a means of automating and checking test case generation. The high levels of automation are made possible by identifying repeating structures in the proofs, restricting the specification to a subset of an otherwise expressive formal notation and exploiting a general–purpose theorem proving tool with built-in constraint solvers.

1 Introduction

In the past, testing and proof have not been easy bedfellows. Despite their shared goal of increased software quality, proof has been seen as being for the cognoscenti, testing for software engineering's working class. The authors believe that this artificial dichotomy is harmful and that testing and proof can be used together to good effect. Even without the benefits of formal refinement, formal specifications can contribute greatly to the quality of a software product. They allow for a concise, unambiguous and explicit specification of the desired behaviour of the system. As such, they are a good basis for automated test activities. Additionally, testing to generate counterexamples to proofs can save much effort and produce illustrative examples for debugging.

The use of formal specifications themselves is still seen by many as a barrier to the widespread industrial usage of formal methods. For the benefits of formal methods in V&V to be fully exploited in industry there is a need to "disguise" the formality in some way [15]. Recent work [5] has shown that formal specifications and the corresponding proof obligations for specification validation can be generated from more intuitive engineering notations with mathematical underpinnings. Such an approach not only enables engineers with the domain knowledge to use specification notations they are comfortable with, but the translation to formal specification has the effect of restricting the subset of the formal notation used and imposes regular structures on the proofs that need to be discharged to validate certain properties (such as completeness and determinism) in the specification. These restrictions, coupled with the subset of data types used for particular domains can be exploited to develop powerful targeted heuristics for automating the V&V activities. The approaches discussed in this paper are assumed to be undertaken in the context of formal specifications generated in this manner.

In the rest of the paper we describe how a combination of testing, proof and restricted structures in the specification can be used to enhance both the integrity and automation of several areas of the software verification and validation process. This symbiotic relationship between testing and proof is made feasible by extending previous work on testing from formal specifications and making use of a flexible theorem proving tool with integrated constraint solvers.

The paper is structured as follows. Section 2 discusses the role of testing in the automatic generation of counter-examples to proofs. Section 3 describes how proof can be used as a means of verifying automated test case generation strategies and also as a means of performing the automation itself. We also show how more effective testing strategies can be developed based on the automatic generation of formally specified test cases and how proof can be used as a testing oracle. Section 4 summarises the main contributions of the work and presents some conclusions.

2 Testing and Proof

Proof conjectures can arise at various points in the V&V process. For example, to ensure that a specification satisfies certain "healthiness" criteria such as completeness and determinism or to verify that a program is a correct refinement of its formal specification. In all cases, invalid conjectures can waste a large amount of proof effort. Therefore, before a long and arduous manual proof is embarked upon it is re-assuring to have a good degree of confidence in the validity of the conjecture. Use of constraint solving techniques to generate counter-examples not only saves proof effort but can provide illustrative information to use when tracking the fault. The generation of counter-examples to verify properties of a specification couched in terms of proofs is a form of testing. Typically sample data are generated and then tested to see whether they break the specification. If this is the case, a counter-example has been found.

Constraint solving in general is known to be intractable [13]. However, in practical situations, one never needs to solve "general" constraints but a particular subset that have restricted structures and particular input space characteristics. These properties can be exploited to automate the search for counter-examples. The authors use the Z [16] type checker and theorem prover CADi$\mathbb{Z}$ [19] to automate this task. In this sense our usage of CADi$\mathbb{Z}$ is similar to that of the Nitpick Z-based specification checker [12] that used model-checking techniques to generate counter-examples to specification assertions. However, CADi$\mathbb{Z}$ has the additional flexibility that general purpose proof tactics can be written (using a lazy functional notation [20]), that can be invoked interactively from within the tool and applied to any proof obligation on the screen. Proof tactics have been written that attempt a best effort at automatically proving conjectures of certain types (e.g. completeness checks). If the proof fails or is inconclusive, the tactics then perform some simplification to transform the conjecture into a suitable form for the integrated constraint solvers. A number of constraint solvers can then be invoked to attempt counter-example generation, these include a model-checker (SMV [3]) and a simulated annealing based heuristic search [6]. The amount of simplification required before the constraint solvers can be efficiently applied will depend on the structure of the proof obligations.

Such automated proof tactics have been used to good effect when a large number of similar proof conjectures were needed to be solved [5]. A situation which would have otherwise been time consuming if done manually and could have led to "reviewer blindness" leading to missed error cases. Different constraint solvers have been found to be effective for different input domains. For example model-checking is only practical for discrete input domains, whereas optimisation-based search techniques are also suited to infinite state spaces and non-linear constraints.

3 Proof and Testing

Formal specifications are a good basis for testing. They allow for a concise and unambiguous representation of the requirements and are amenable to proof and automated analysis. Test generation techniques for model-based formal specifications [14, 8, 17] such as Z [16] or VDM-SL [10] are typically based on the principle of partitioning the specification into equivalence classes [9]. Equivalence classes are partitions of the specification input space that are assumed, for the purpose of testing, to represent the same behaviour in the specification. Such techniques are amenable to automation and tool support. However, as in all cases where automation is introduced, and especially for high integrity systems, the integrity of such tools is of great importance. For automated testing to be able to provide confidence in the conformance of the software to its specification, the test generation strategies must be both verified and validated. In other words, they must not only be shown to be correctly implemented but must also be shown to be adept at finding errors in the implementation.

3.1 Verification of Automated Testing Strategies

There are various criteria that can be used when verifying that test partitioning strategies have been correctly implemented. For example, the tests can be shown to completely cover the valid input space of the original specification. If this were not the case, important parts of the implementation, that could possibly contain faults might remain untested. If the resulting tests are represented using the same formal notation as the original specification, these verification activities can be performed using proof. The completeness of the generated tests ($T_1 \mathinner{\ldotp\ldotp} T_n$) with respect to the original specification (Spec) can be verified by proving a conjecture of the following form:

$$\begin{array}{l} \mathit{Theorem}1 : \\ \vdash \forall \mathit{Inputs} \bullet \mathit{Spec} \Leftrightarrow T_1 \lor ... \lor T_n \end{array}$$

An example partitioning strategy identifies expressions in the specification of the form $A \lor B$ and partitions these into the following test cases $A \land B$, $\neg A \land B$ and $A \land \neg B$ [8] where A and B could be complex predicates themselves. The conjecture used to prove that these partitions preserve the valid input state-space of the original specification would therefore take the following form:

$$\begin{array}{l} \mathit{Theorem}2 : \\ \vdash \forall \mathit{Inputs} \bullet A \lor B \Leftrightarrow \\ (A \land B) \lor (\neg A \land B) \lor (A \land \neg B) \end{array}$$

This theorem can be proven in a few simple steps. The same proof steps can be used regardless of the structure of the expressions represented by A and B. In general, a proof can be derived for each partitioning strategy and used to verify the outcome each time that strategy is applied. Using the proof tactic mechanism in CADi$\mathbb{Z}$ the authors have automated these proofs for a number of common partitioning strategies. Whenever a strategy is applied, the corresponding correctness proof can be automatically invoked on the result. This ensures that, whatever the means of test generation, the *result* can always be shown to be valid or otherwise. The tool can be instructed to record the individual proof steps taken in applying a proof tactic and these can be printed in a form amenable to human scrutiny. Therefore, if the tool cannot be trusted, a rigorous argument can be developed to support the validity of the proof steps.

Given a formal definition of a testing strategy as an equivalence (e.g. *Theorem 2* above), the derivation of the test cases themselves can also be automated using general purpose proof tactics. The principle is similar in operation to the use of Disjunctive Normal Form (DNF) to simplify an expression into a disjunction of conjuncts that can each be used as separate test cases. Where conversion to DNF uses simple logic rewrite rules to distribute disjunctions, more targeted equivalences can be formulated based on common testing heuristics.

Test partitioning based on the formal specification of the testing heuristics has been implemented using CADi$\mathbb{Z}$ proof tactics. Generic partitioning strategies are specified as equivalences in the form of *Theorem 1*. A proof tactic is invoked upon the predicate to be partitioned to instantiate the generic equivalence with the operands of the predicate and simplify the whole specification to reveal a disjunction of partitions. Each test case is equivalent to the original specification where the input space has been constrained according to one of the partitions. The completeness of the partitioning strategy is left as a side conjecture to prove to ensure that the partitioning was valid. This can be automated by extending the partitioning tactic with the general purpose proof for the strategy as described above. The following simple example demonstrates how test partitions are derived. The example specification calculates the square root ($r!$) of a positive integer ($n?$) and the test partitions are generated using boundary value analysis of the $\geq$ operator (based on the premise that errors often occur on or around the boundary [2]).

The specification is given as the following Z schema[1]

[1] ? and ! are Z convention for inputs and outputs respectively.

SquareRoot

$$n?, r! : \mathbb{R}$$

$$n? \geq 0 \land$$
$$r! \times r! = n?$$

The boundary value analysis test heuristic for real numbers is specified as the following equivalence. Note that the "just-off" the boundary case is chosen here as 0.1. This value may vary for different applications and would chosen by the tester based on various application attributes such as the resolution of the concrete types used to implement the abstract $\mathbb{R}$ type.

$$\vdash \forall x, y : \mathbb{R} \bullet x \geq y \Leftrightarrow$$
$$(x = y) \lor (y < x \leq y + 0.1) \lor (x > y + 0.1)$$

An un-partitioned test specification for the schema is described as an existential quantifier as follows:[2]

$$\vdash \exists n?, r! : \mathbb{R} \bullet n? \geq 0 \land r! \times r! = n?$$

The partitioning theorem is now introduced and instantiated with the local operands. The partitioning theorem is left as a side condition that should be proven before the test cases can be considered valid. This results in the following theorem:

$$\forall x, y : \mathbb{R} \bullet x \geq y \Leftrightarrow$$
$$(x = y) \lor (y < x \leq y + 0.1) \lor (x > y + 0.1)$$
$$\vdash \exists n?, r! : \mathbb{R} \bullet n? \geq 0 \land r! \times r! = n? \land$$
$$((n? = 0) \lor (0 < n? \leq 0 + 0.1) \lor (n? > 0 + 0.1))$$

The side condition is proven (e.g. using a predetermined proof tactic) and the existential quantifier simplified to leave the following three test cases.

$$\vdash \exists n?, r! : \mathbb{R} \bullet n? \geq 0 \land r! \times r! = n? \land$$
$$n? = 0$$

$$\vdash \exists n?, r! : \mathbb{R} \bullet n? \geq 0 \land r! \times r! = n? \land$$
$$0 < n? \leq 0.1$$

[2]This can be roughly interpreted as: there exist some values for $n?$ and $r!$ that satisfy the specification and can therefore be used as suitable test data.

$$\vdash \exists n?, r! : \mathbb{R} \bullet n? \geq 0 \land r! \times r! = n? \land$$
$$n? > 0.1$$

Once the test partitions have been produced, satisfying test data can be generated by "solving" the existential quantifications using the constraint solvers in CADi$\mathbb{Z}$. The partitioning method described above supports work by Stocks and Carrington [17, 18][3] who proposed a framework for the derivation and specification of test cases based on the Z notation (the Test Template Framework). The method of test case derivation described here complements that work by providing a mechanism for automatically applying the test heuristics to reveal the test partitions that can then be structured using the Test Template Framework.

3.2 Validation of Automated Testing Strategies

Mutation testing [7] is a fault-based testing technique that deliberately injects faults into a program in order to assess a test set's adequacy at detecting those faults. Based on the number of injected faults detected (mutation score), conclusions about the general fault finding ability (mutation adequacy) of the test set can be formed. Mutation testing provides a means of validating the test strategies discussed in the previous section. Automatic test case generation, as described above, can provide a statistically significant number of test cases for various strategies. The mutation adequacy of each of these strategies can then be assessed to compare their relative effectiveness at detecting faults [1].

Expressing a test case specification as a formal specification from which the test data are generated also opens up the possibility for some additional manipulation to increase the mutation score of the data. Mutation techniques can be applied at the specification level to create specifications that represent an abstract description of potential faults in the implementation (as first suggested by Budd and Gopal [4]). If test data can be generated from the original specification that identify (kill) the mutants, that data is also likely to achieve a relatively high score at the program level. The data generated from the specification would have been "hardened" in some sense against

[3]As well as building on other important work in the area such as [11, 14, 8].

the likelihood of encountering co-incidental correctness in the implementation.

In general, the number of mutants that can be generated for an expressive formal specification notation such as Z would be extremely large. However, in practice, only a subset of the notation would be used for any particular application domain. In this case, the number of possible mutants would be limited. From within this subset more selective choices of which mutation strategies to apply can be made by analysing the mutation score of particular testing strategies. Hardened test data can then be generated from the test cases by strengthening the predicate of the test case to improve the probability that data are generated to kill the chosen set of specification mutants. In some cases, one set of test data could be generated to kill a number of mutants. However, where the hardening predicates are inconsistent, several sets of test data may need to be generated. Take as an example, the following simple test case for a system which averages two numbers:

$$\exists\, A, B, \mathit{Result} : \mathbb{N} \bullet \mathit{Result} = (A + B)\, \mathit{div} 2$$

The test data can be hardened against the mutation where the $+$ is replaced by a $-$ by adding an inequality to the test case.

$$\exists\, A, B, \mathit{Result} : \mathbb{N} \mid (A + B) \neq (A - B) \bullet \\ \mathit{Result} = (A + B)\, \mathit{div} 2$$

The hardening predicate in this case was $(A + B) \neq (A - B)$. This represents a necessary condition for detecting the mutant but, in general, will not always be sufficient. Depending on other mutations that may arise elsewhere in the implementation, the new test case can not be guaranteed to produce data which kills the mutant, but is more likely to do so than without the hardening predicate. Mutation analysis was briefly mentioned by Stocks and Carrington [17] as an alternative testing heuristic to domain propagation in their Test Template Framework. However, the authors believe there is still scope for research in investigating effective mutation strategies for Z-based test sets and whether mutation analysis can be combined with standard domain partitioning to provide more effective test sets. Therefore, future work will evaluate various criteria for designing the hardening predicates and their relative efficacy at increasing mutation scores when applying the tests to the implementation. If hardening predicates could be automatically generated based on a known set of specification mutations, it may be possible to use the feedback from traditional mutation testing approaches (for assessing the effectiveness of test sets) to automatically select the most effective test strategies for particular types of program.

3.3 Proof as a Testing Oracle

Test data generated from formal specifications are typically not at the same level of abstraction as is needed to test the implementation. Some refinement will be needed to exercise the implementation with the test inputs. For implementations which do not preserve the structure of the original specification this refinement may be difficult. In addition, some specifications may be non-deterministic, eliminating the possibility of pre-calculating expected test results.

An alternative to the structured decomposition of the specification into test cases and expected results, as discussed above, is to use a “generate and test” approach. Test inputs are chosen via any means (e.g. randomly) and the results of applying the inputs to the implementation are then checked for conformance with the specification. This approach can also be used in conjunction with the partition-based testing. A statistically significant number of samples can be chosen from each test partition to increase the confidence in the equivalence class hypothesis used to generate the test cases. In either case, the process of checking the test inputs and outputs against the specification requires a test “oracle”. If enough refinement information is known to transform the concrete inputs and outputs of the system into their equivalent in the abstract specification, the formal specification and automated proof tactics can be exploited to form an automated oracle. The specification is instantiated with the inputs and outputs and a proof tactic is used to simplify the expression to *True* (test passed) or *False* (test failed). Such simplification is ideally suited to automated theorem provers as it typically involves applying many “one-point” simplifications until the expression is reduced to either *True* or *False*.

4 Conclusions

In this paper we have shown how judicious use of testing and proof to support one another can lead to significant benefits for the software V&V process, both in terms of increased automation and integrity. The use of counter-example generation can save much wasted proof effort and the use of proof to support test case design can be used to demonstrate the correctness of the test partitioning techniques as well as offering a means of automation in itself.

The high level of automation is made possible because of the combination of restricting the subset of the formal notation used, the ability to predict the structure of the proofs that are required (and therefore the ability to re-use proof tactics many times) and the use of a powerful theorem proving tool with integrated constraint solving abilities. In the authors' experience in aerospace applications, these restrictions did not need to be contrived but occurred naturally as a property of the domain and the types of proof that were performed.

Some of the techniques described here (e.g. counter-example generation and automated test case and data generation) have already been applied to a large industrial case study [5]. Other techniques, (e.g. automated proof as a testing oracle and application of mutation testing concepts) require more research to fully explore their potential. In particular, the use of mutation testing techniques, both at the code and specification level appears a promising method of automatically generating effective and efficient test criteria for Z-based testing of particular application domains. This is an area of research that is made possible by the automated framework described in this paper and will be the focus of future work.

5 Acknowledgements

This work was funded by the High Integrity Systems and Software Centre, Rolls-Royce Plc.

References

[1] S.P. Allen and M.R. Woodward. Assessing the quality of specification-based testing. In Sandro Bologna and Giacomo Bucci, editors, *Proceedings of the third international conference on achieving quality in software*, pages 341–354. Chapman and Hall, 1996.

[2] Boris Beizer. *Software Testing Techniques*. Thomson Computer Press, 1990.

[3] Sergey Berezin. The SMV web site. http://www.cs.cmu.edu/~modelcheck/smv.html/, 1999. The latest version of SMV and its documentation may be downloaded from this site.

[4] Timothy A. Budd and Ajei S. Gopal. Program testing by specification mutation. *Computer Languages*, 10(1):63–73, 1985.

[5] Simon Burton, John Clark, Andy Galloway, and John McDermid. Automated V&V for high integrity systems, a targeted formal methods approach. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, June 2000.

[6] John Clark and Nigel Tracey. Solving constraints in LAW. LAW/D5.1.1(E), European Commission - DG III Industry, 1997. Legacy Assessment WorkBench Feasibility Assessment.

[7] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, pages 34–41, April 1978.

[8] J Dick and A Faivre. Automating the generation and sequencing of test cases from model-based specifications. *FME'93:Industrial Strength Formal Methods, Europe. LCNS 670*, pages 268–284, April 1993.

[9] John B Goodenough and Susan L Gerhart. Towards a theory of test data selection. *IEEE Transactions On Software Engineering*, 1(2):156–173, June 1975.

[10] The VDM-SL Tool Group. *The IFAD VDM-SL Language*. The Institute of Applied Computer Science, September 1994.

[11] P A V Hall. Towards testing with respect to formal specifications. *Second IEE/BCS Conference On Software Engineering*, pages 159–163, 1988.

[12] Daniel Jackson and Craig Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on software engineering*, 22(7):484–495, July 1996.

[13] A. K. Mackworth. Consistency in networks of relations. *Artifical Intellegence*, 8:99–118, 1977.

[14] Thomas J Ostrand and Marc J Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[15] Martyn Ould. Testing - a challenge to method and tool developers. *Software Engineering Journal*, 39:59–64, March 1991.

[16] J. M. Spivey. *The Z Notation: A Reference Manual, second edition*. Prentice Hall, 1992.

[17] Phil Stocks and David Carrington. Test template framework: A specification-based case study. *Proceedings Of The International Symposium On Software Testing And Analysis (ISSTA'93)*, pages 11–18, 1993.

[18] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions On Software Engineering*, 22(11):777–793, November 1996.

[19] I. Toyn. Formal reasoning in the Z notation using CADi$\mathbb{Z}$. *2nd International Workshop on User Interface Design for Theorem Proving Systems*, July 1996.

[20] Ian Toyn. A tactic language for reasoning about Z specifications. In *Proceedings of the Third Northern Formal Methods Workshop, Ilkley, UK*, September 1998.