Teaching formal methods lite via testing

Mark Utting* and Steve Reeves

Department of Computer Science, The University of Waikato, Hamilton, New Zealand

SUMMARY

A new style of formal methods course is described, based on a pragmatic approach that emphasizes testing. The course introduces students to formal specification using Z, and shows how formal specification and testing can benefit each other, in both the validation and verification phases. It uses a tools-based approach, with practical work that reinforces formal specification techniques as well as traditional software engineering skills, such as unit and system testing, inspection and defensive programming with assertions. The two main results are to identify several practical uses of formal specifications that are not widely practised or taught, and to demonstrate that teaching them results in a more interesting and relevant formal methods course. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: testing; specifications; assertions; teaching; formal methods

1. INTRODUCTION

Like many other universities, The University of Waikato offers a course on formal methods as part of the undergraduate computer science curriculum. The course, entitled ‘Advanced Software Engineering’, is a compulsory subject in the software engineering programme of the four-year BCMS (Bachelor of Computing and Mathematical Sciences) degree and an optional subject in other programmes. It generally attracts about 20 to 25 students. Topics covered in the course usually include formal specification using Z [1], code verification using Floyd–Hoare logics [2] and simple forms of proof such as calculating preconditions, verifying invariants and proving simple safety properties. In 1997, the student evaluation results for the course showed that most students were disinterested in much of the content and many saw it as ‘irrelevant’ and ‘not practical’.

Consequently, the course was redesigned to focus on practical applications of formal methods, as described in Sections 2 through 4. The redesigned course, based on this so-called ‘formal methods lite’
approach, was run in 1998 with positive outcomes, but also some surprising findings, as described in Section 5. Conclusions and areas for further improvement are discussed in Section 6.

2. COURSE DESIGN PHILOSOPHY

Although typical software engineers might see formal methods as being useful for safety-critical systems, where a high degree of quality assurance is required, they usually do not believe that it is cost-effective to apply formal methods to more typical software applications. The authors of this paper agree with this, and believe that formal development or verification of code is currently too time-consuming to be cost-effective and that many possible benefits of formal specification are currently not being practised or taught.

The most commonly cited benefits of formal specification are

- the process of writing the formal specification leads to a deeper understanding of the system being specified and typically uncovers many sources of ambiguity and incompleteness in the informal requirements, and
- the formal specification forms a good basis for detailed design and code development (either formal or informal), because it is a concise, precise description of the system that is to be built.

Although these are major benefits, it is not easy to present convincing objective evidence for the cost-effectiveness of formal specification from these alone. The improved understanding and precision that result from formal specification is difficult to measure quantitatively, so claims about its benefits are often subjective. The same is true for the second benefit, given that typical software engineers will be developing code informally, rather than formally.

Another criticism is that it is almost as easy to introduce mistakes into a formal specification as it is into a program, so there is no guarantee that a formal specification will be a more accurate reflection of a client’s real requirements than a program would be. Current formal specification practice is not strong on validation techniques, other than typechecking to detect obvious mistakes and perhaps inspection to cross-check the formal specification against informal requirements documents.

A better approach to answering these questions is to emphasize other potential benefits of formal specifications, that are not widely practised or taught. Current practice is to write a formal specification, typecheck it, then move on to later (informal) stages of development, with the formal specification treated as a reference document only. In contrast, the authors’ dream is to see the formal specification become the central document in the whole software development environment, being continually used by various tools throughout the software lifecycle. Even when code development is done informally, there are useful ways of relating specification and code. In other words, the philosophy behind this course is that, having gone to the effort of writing a formal specification, one should then get as much benefit out of it as possible.

Here are some of the specific techniques that need to be practised more widely, and that this course emphasizes. Note that the first three are validation techniques, while the remaining three are ‘soft-verification’ techniques that cross-check the specification against informally-developed program code during execution. Here these techniques are used with Z, but they could equally well be used with other specification languages, such as VDM.
(i) **Using animation to validate the specification.** This allows the software engineer (or the client in some circumstances) to interact with the animated specification. Of course, not all specifications can or should be animated [3]. However, specifications that specify output values as an expression over input values can be animated, albeit on small data sets and very slowly. Probably the majority of specifications are written in this form, but there are many exceptions too. Nevertheless, this is a useful way of exploring the functionality of systems. Even if a specified operation does not specify unique outputs, it can sometimes still be animated by prompting the user to supply possible output values (or exhaustively trying all output values, if the state space is very small), then checking those values against the specification. There has been quite a lot of research in this area (see [4,5] for overviews of approaches to Z animation); it is quite often taught in formal methods courses and many tools are available‡ (though most do not handle nondeterministic specifications).

(ii) **Using test cases to validate the specification.** It is useful to provide examples of acceptable input/output values for each specified operation. These can then be automatically checked against the specification each time the specification is modified, just like the use of typechecking. For example, given a Z schema \(\text{OP} \) with inputs \(? : T_i \), outputs \(! : T_o \) and state variables \(s, s' : T_s \), the specifiers can check that values \(a, b, c, d\) satisfy \(\text{OP}\) by evaluating

\[
(\exists ? : T_i; o! : T_o; s, s' : T_s \\
\quad | \quad ? = a \land o! = b \land s = c \land s' = d \implies \text{OP})
\]

For the majority of specifications, formulae like this can be automatically evaluated to true or false by a good theorem prover, such as Z/EVES [6]. This is such a useful way of testing specifications that Z/EVES provides a replacement operator that abbreviates the above to:

\[
\text{OP}[? := a; o! := b; s := c; s' := d]
\]

It is also useful to provide counter-examples for operations, and check that they evaluate to false. The use of positive and negative test cases allows the behaviour of the operation to be bounded on both sides (thus detecting both overspecification and underspecification) and is a useful way of checking that the operation specifies what was intended. The Java Modelling Language [7] (a Java extension with specification facilities§) has built-in support for such test cases, integrated elegantly into the axiomatic semantics.

The authors firmly believe that every Z specification should include a set of positive and negative test cases, either interspersed with the schemas or as an addendum to the specification. These test cases can assist readers’ understanding and can be mechanically checked against the specification. Although this technique is quite obvious, and is taught in some specification courses, it is still not widely practised.

(iii) **Validating specifications by inspection.** Inspection is an important software engineering technique that can be applied to formal specifications [8]. This provides a good opportunity to teach the usual team member roles and inspection processes, as well as being a good exercise in critiquing specification style.

(iv) **Generating test sets from formal specifications.** Systematic unit and system testing is one of the primary techniques used to ‘verify’ software correctness in traditional software engineering [9,10].

‡See http://archive.comlab.ox.ac.uk/z.html for a list of Z animators.
Testing can account for a large proportion of the total effort in the software lifecycle, up to 50%. When a formal specification is available, research has shown that it provides a good guideline for producing comprehensive black-box test sets [11–14]. A formal specification can also often be used to solve the ‘oracle problem’ in testing—determining if the output of a test run is correct or incorrect [15]. This becomes more difficult when the specification and implementation are at different abstraction levels, but it is usually possible to apply a retrieve function to the implementation outputs to relate them back to the specification abstraction level. This works for data refinements that are functional [16], which is the vast majority.

The Test Template Framework (TTF) of Stocks and Carrington [12,17,18], is a useful method for generating test sets from specifications. For each operation specified in Z, the TTF requires a tree of Z schemas to be constructed, where the root of the tree is the valid input space of the operation, the nodes of the tree are subsets of that input space and the leaves of the tree correspond to individual tests of the operation. Standard heuristics, such as boundary-value analysis and cause-effect analysis, can be used to design the tree, and the TTF provides several additional heuristics that are guided by the structure of the Z specification.

In the course described here, students are required to develop TTF trees by hand, so that the connection between the specification, the heuristics and the resulting TTF tree and test set is clear. However, in real applications, tool support would be useful. Some prototype tools have been developed in this area [19], but much more work is needed.

Testing using formal specifications is an exciting and fruitful area for further research and tool development, because it promises to use formal specifications to generate test sets more cost-effectively and solve the oracle problem. These benefits alone are powerful reasons for writing formal specifications. Although there has been quite a lot of research in this area (see [18] for an overview) it is not widely taught and tool support is still lacking.

(v) **Transforming formal specifications into assertions within code.** Assertions are widely recognized as an important technique for improving the quality of code [20] and catching errors during debugging and testing [21].

The basic idea is to insert assertion statements that check the correctness of data structures and algorithms and detect illegal conditions as soon as they arise. When the program is executed, these assertions monitor correctness and raise errors whenever an assertion fails. Typically, the assertions can be disabled via a compiler switch, so that they impose no overhead on production programs.

A formal specification of a procedure is a useful basis for generating assertions. The postcondition of the specification generates assertions at the end of the procedure, to check the correctness of the algorithm within the procedure. The precondition of the specification generates assertions at the beginning of the procedure, to check that clients have called the procedure correctly. As noted above, it is necessary to insert retrieve functions into these assertions when the specification and implementation are at different abstraction levels.

Assertions that are generated from pre/postconditions are typically much more useful for catching errors than assertions that just check for runtime errors like null pointer dereferencing and array overflows (which are the kinds of assertions that are more commonly written by programmers). However, the two kinds of assertions are complementary, and both should be taught.

Again, an automated or partially-automated system for translating specifications into assertions within code would be a useful tool for catching errors within implementations and another powerful reason for writing formal specifications. The closest tools to this are language extensions that integrate...
formal specifications and code into the one framework, such as Larch [22], the Java Modelling Language and Escher\(^6\), but the main focus of these languages is specifying behaviour, rather than executing the specification and implementation in parallel as a debugging aid.

(vi) **Using formal specifications to specify components of systems such as the classes of an object-oriented system.** Formal specification is often presented as if it is normally used to specify the behaviour of a complete application. However, it can equally well be used to specify the behaviour of small subsystems (components), such as a single C++ class.

A major attraction of component-based specification is that it is not necessary to specify the whole system to gain most of the benefits of formal specification. Once a component \( C \) is specified, its specification can be used to generate test sets and assertions etc. for \( C \). All client programs that use \( C \) also benefit, because when they call \( C \) procedures incorrectly, the precondition checking within the \( C \) procedures is likely to detect the client’s error immediately. So, component-based specification is cheap to apply, and its benefits are immediate. Furthermore, it is often easier for beginners to specify a single component than a whole system, because the component is smaller and more concrete and managing complexity and finding good abstractions are less critical for small systems than for large ones. Another attraction is that it is useful to partially specify a component. For example, a partial specification of a class might involve simply defining a class invariant that checks numeric ranges more precisely than C++ does (e.g., 1 \( \ldots \) 10 rather than just int). This still provides some useful documentation and self-checking capabilities for that class.

At the other extreme, the full functionality may be specified, then used to perform comprehensive assertion checking by adding abstract data fields to the class (perhaps using Z-like data types from a library, or the C++ standard template library), updating them in parallel with the concrete data fields and checking a retrieve function [16] between the abstract and concrete data fields after every method. Like techniques (ii) and (iv) above, this approach has the advantage that the concrete implementation calculates the new concrete state and output values, so the specification can be used simply as an oracle (checking that the initial and final abstract states satisfy the specification), rather than having to be executable in order to compute outputs itself. (A retrieve relation would allow more data refinements to be formalized [16], but would be less useful for this ‘parallel-execution’ technique, because in general it requires the specification to be executable, so that the abstract outputs and final state can be calculated before the retrieve relation is checked.)

Finally, teaching specification of classes in object-oriented languages is particularly attractive, because it interacts nicely with subtyping (‘interfaces’ in Java [23]). For example, typical correctness rules for subtyping are similar to the rules for data refinement of an Abstract Data Type (ADT), except that the subtype may add extra public data variables and methods [24]. Thus teaching class-based specification gives natural opportunities to introduce or reinforce concepts like data refinement, invariants and programming by contract [25].

Because of these attractions (particularly the fact that it can be applied partially and to any part of a system), specification of components, rather than whole systems, is likely to become one of the main applications of formal specification in the future. Two promising examples of such an approach are the Extended Static Checkers for Modula-3 and Java [26] and the Java Modelling Language [7].

which both add specification constructs to Java and provide various ways of using them to check the correctness of the Java code.

Those are some of the main techniques that are included in the course. Note that they all centre around formal specification, and that none of them require formal verification of code, or extensive knowledge of theorem proving. The only theorem proving that is needed is an evaluator for Z expressions and predicates with concrete values given for each variable. This evaluation can often be fully automated.

The Z/EVES [6] theorem prover is used throughout the course, because it performs Z typechecking, can also be used to expand schema expressions and calculate preconditions, and it has a powerful semi-automatic prover that is capable of evaluating many concrete Z expressions automatically.

For programming tasks, C++ [27] is used because the students are familiar with it, it is still the standard object-oriented programming language in industry, and because it is one of the most formal-methods hostile languages in common use. This also shows students that the techniques can be applied to any language, not just one with clean semantics like Dijkstra’s guarded command language [28].

The overall goal is to expose students to formal specification techniques that are leading edge yet practical, and at the same time reinforce traditional software engineering skills. This ensures that even those students who will never use formal specifications again will go away from the course with improved practical skills in areas like safety-analysis techniques, black-box testing, inspection teams, using assertions effectively and rules for object-oriented subtyping, plus an understanding of how formal specifications can improve the effectiveness and precision of these practical areas.

3. COURSE OUTLINE

This section briefly describes the course design. The course is divided into six modules, each running for two weeks and assessed via a practical assignment or a written test.

The six modules are as follows.

**Module 1. Computers that kill.** This module covers techniques for measuring and improving the safety of computer-based systems, including risk analysis, hazard analysis and various software engineering techniques for improving quality.

**Module 2. ‘What does it do?’ (reading Z specifications).** An introduction to formal specifications and the Z specification language.

**Module 3. Test, test and test again.** This module covers: animation of specifications; validation of specifications against test data using Z/EVES; an overview of testing theory; and black-box techniques for generating test sets from formal specifications (using the Test Template Framework).

**Module 4. ‘What do I want it to do?’ (writing Z specifications).** How to write and structure Z specifications. Validation of specifications using Z/EVES for typechecking and evaluation of test examples. Inspection techniques for specifications.

**Module 5. From specs to code and back again.** An introduction to the relationships between specifications and code, based on Hoare logic. The use of assertions to detect errors in programs [20]. Generating assertions from specifications. Theory of data refinement and techniques for handling data refinement within assertions.

In addition to the six assignments (8% each), and a final exam (34%), the course includes a specification project (18%) that students do in pairs. Note that modules three and five are focussed on specification-based testing, and, in module four, students are encouraged to validate their specifications via test cases. In other words, about half the course involves specification-based testing.

4. TWO-EDGED ASSIGNMENTS

This section describes the 1998 version of the two assignments that relate most closely to current industry practice (testing and assertions). Each practical is double-edged, in that it develops formal methods skills at the same time as exercising practical software engineering skills.

4.1. Assignment 3: animation and testing

The students were given a Z specification that specified two classes (ADTs), called IDSet and GradeSys. The IDSet class simply stored a finite set of student identity numbers. All its operations were deterministic and capable of being animated. The GradeSys class used five instances of the IDSet class to record student grades A to E. This was a good opportunity to illustrate lifting (or promotion) schemas in Z.

The assignment had three parts. A summary of the instructions given to students follows.

1. Validation by animation (20 marks).
   Choose a sequence of IDSet operations that exercises all of the operations several times, and show how it can be executed by Z/EVES. Your sequence should contain about 10 to 12 operations.

2. Validation by testing (20 marks).
   To show that we can still validate schemas via testing, even when they are not executable, let us define an extra IDSet operation that is non-deterministic. It outputs any one of the students in the given IDSet.

   ![Diagram]

   Use Z/EVES to test each of the public IDSet operations, plus this Someone operation. For each operation, give two positive test cases (that meet the specification) and one negative test case (that contradicts the specification).

3. Test set generation using TTF (60 marks).
   Generate a comprehensive set of black-box tests for an implementation of the student grade system, using the Test Template Framework.
Your tests will be applied to ten implementations of that system. One of the implementations has no known bugs, while the other nine each have one bug inserted. Thirty marks will be given for the number of bugs that your tests detect (3 marks per faulty implementation that you detect). The other thirty marks will be allocated according to how well you have used the TTF and how systematic and well-designed your test set is.

To make things easier for you, and to enable you to judge your progress, six of the implementations (five faulty ones, plus the probably correct one) will be available for downloading from the course website. To keep you in suspense, and to motivate you to develop a comprehensive test set, the remaining four faulty implementations will not be available to you.

In addition to submitting your test set, you must also submit a file called ttf.zed that shows how you developed the test set. This should contain a picture of the TTF tree that you developed, define a Z schema for each node of the tree and explain your reasons for applying particular test generation strategies at each point in the tree.

Note that this practical developed

- formal methods skills—Z comprehension and writing simple Z schemas, and
- software engineering skills—design of black-box test sets.

On average, students submitted about 20 test files with about 15 commands in each file, and found 7.4 of the 9 errors, some of which were quite subtle. Their TTF trees were generally well-designed, and these results suggest that the quality of the resulting test sets was quite good.

In the 2000 semester, some students applied the TTF techniques to programs developed in other courses that had already been informally tested and used in applications. They attempted to quantify the quality of their TTF-generated test sets by applying code coverage and mutation testing techniques to the program under test. The results shown in Table I suggest that the TTF techniques produced quite good quality test sets, with high code coverage and good detection of the kinds of errors introduced by the mutations. Two real errors were found in the programs even though they had been tested previously. However, this is a small sample of very small programs. There is a need for much more research on the effectiveness of TTF specification-based testing.

### 4.2. Assignment 5: assertions

The primary goal of this assignment was for students to learn several ways of adding assertions to C++ classes and to understand the basic concepts of data refinement. Rather than using the standard C++ assert statement, several specific statements (PRE(), POST(), ASSERT(), and CLASSINVARIANT()) were used, to distinguish the different kinds of errors more clearly. A summary of the instructions the students were given follows.

---

**Surprisingly, not one student objected to the concept of being able to get a maximum of 27/30 for this section! Perhaps they were confident of finding errors in the ‘correct’ implementation? Anyway, in following years, 10 faulty implementations were used so that 30/30 was achievable.**

**The mutations were formed by changing operators like `<` to `<=`, `++` to `--`, `==` to `==` etc. Only one mutation was made in each version of the program under test. A mutation was considered to have been ‘detected’ by the test set if it gave different output to the original nonmutated program, or if it crashed when the original did not.**
You are given two C++ classes (containing some errors), and will also have to implement one more class yourself. After you submit the resulting classes, they will be exercised by my special test data (carefully designed to expose the errors) and several client programs that misuse the classes in various ways. If you have done a good job of adding assertions, then any errors that arise will be trapped by your assertions, and you will get top marks. This simulates the real-life scenario of you writing some C++ classes, and adding assertions in order to:

- catch any coding errors within those classes (in this case, I’ve inserted some non-obvious errors on purpose, so that I know what they are and can mark the assignment fairly across the whole class);
- catch any client programs that use your classes in the wrong way (I’m pretending to be that stupid client programmer who has not read your class documentation carefully enough, so is writing code that misuses your classes).

The assignment consists of the following tasks.

1. **(60 marks)** Add extensive assertion checking to the `HashSet` class (which implements the `IDSet` Z specification), by adding abstract specification variables to the class, executing the abstract and concrete algorithms in parallel and checking their consistency within the class invariant procedure (which should be called after each method). The retrieve function from the concrete to abstract variables is . . .

   After doing this, you should have a pretty water-tight system! An extreme case of software redundancy—running two sets of data structures and algorithms in parallel, just like safety-critical systems do for hardware. If both the abstract and concrete subsystems complete execution, and produce results that agree, we can be pretty confident that the answers are correct.

   But note the ‘if they complete execution’ caveat! The concrete algorithms might still fail due to errors causing nasty C++ runtime errors, such as array bounds overflow or NULL pointer dereferencing. So, it is worth looking briefly through the concrete code that I gave you and adding `ASSERT()` checks anywhere that you think C++ runtime errors might arise. It is better to catch these with assertions than with core dumps!

2. **(40 marks)** Implement the `LogGradeSys` class which is specified in `grades.zed`. It is a subtype of the `GradeSys` system, so in C++ your `LogGradeSys` class should publicly inherit from the `GradeSys` class.
Make sure you override the class invariant, and define the stronger invariant (it is good to call the parent invariant, to avoid duplicating its code). Similarly, you will have to override some methods to alter their functionality as required by the specification, and add the new methods. However, the original GradeSys class did not contain any assertions, so you should also override most of the other methods so that you can add more assertions.

Unlike part 1 of this assignment, where we went wild with assertions (and duplicated the data structures), in this class I want you to take a more moderate approach.

Try to add assertions that check the precondition of each method. For the postconditions, it is not always convenient to check every detail of the state change and the output results, but try to check a few key aspects of each method. For example, when adding a new student to a grade, perhaps you can check that the size of that grade increases by one. Also, look for any C++ runtime errors that you can catch. The aim is to get a reasonable level of checking, without spending huge amounts of time writing assertions and without adding so many expensive assertions that the program becomes unusably slow when run in debugging mode. This moderate approach is what you would normally do when writing (non-critical) commercial software.

Note that this practical developed

- formal methods skills—understanding the role of preconditions, postconditions, invariants and data refinement, and
- software engineering skills—robust programming using assertions in C++.

The annotated C++ classes that they submitted were executed under 10 different scenarios: one to check that normal functionality was not compromised by the assertions, three that exercised errors within the IDSet implementation (these should be caught by postcondition assertions), three erroneous client calls to IDSet methods (should be caught by precondition assertions) and three erroneous client calls to LogGradeSys methods (should be caught by precondition assertions).

Most students annotated the IDSet implementation well enough to catch all of its three errors (executing the abstract and concrete algorithms in parallel is a stringent test of correctness). On average, they also correctly caught\(^\dagger\dagger\) 54% of the IDSet client errors and 60% of the LogGradeSys client errors.

5. RESULTS

The redesigned course was run in 1998, using the same textbook, lecturers and course title as in 1997. As hoped, student morale and course evaluations improved markedly (see Figure 1). Using a five-point scale (1 = excellent, 2 = very good, 3 = satisfactory, 4 = unsatisfactory, 5 = poor), the overall course rating improved from 2.9 to 2.3, with no students rating the course as unsatisfactory or poor (in 1997, 32% rated it unsatisfactory). More telling was that 76% of students would now encourage others to take the course (36% in 1997). A higher percentage of students (47%) stated that their interest in the subject was increased and only one student said his/her interest was reduced.

Practicals 3 and 5 were given the best rating on a usefulness scale, which is understandable because they relate most closely to current industry practice (testing and assertions).

\(^\dagger\dagger\) ‘Correctly caught’ means that a PRE () assertion was triggered. Students were given reduced marks if some other kind of assertion was triggered.
Figure 1. Results of student evaluations, 1997–1998.
Marking student assignments gave the informal impression that students were producing better quality Z specifications than students from previous years, in spite of less time being spent directly teaching Z. However, in one exam question that was similar in 1997 and 1998, there was no significant difference in the results.

One small explicit experiment was conducted on the students (with their knowledge). They were asked to write two specifications: one for a simple email system and one for a simple hierarchical filing system. About half of the students were asked to perform tests (along the lines of technique (ii) in Section 2) on just the email system, the other half of the students were asked to do the same on just the file system. The conjecture was that some qualitative differences would be observed between the specifications that were tested and those that were not.

During marking of this work two things became apparent. One was that even when testing showed that there was something wrong with the specification, the students did not seem to use this as a prompt to go back to the specification and repair it. Instead, they gave the results of the tests, perhaps mentioned that the test had failed, and moved on.

A variant of this behaviour seemed to be caused by wishful thinking. In this variant, Z/EVES would typically not have reduced the testing predicate to either true or false, but rather to some combinatorially complex but semantically simple expression. Having at the back of their minds the expected outcome of the test (e.g., that the test should have produced ‘true’) the students typically stated that the complex expression ‘obviously’ or ‘clearly’ (always a danger signal when those words are used!) evaluated to the expected truth-value even when, in fact, it evaluated to the opposite truth-value.

The other commonly observed problem with the testing was that the students used special cases for the tests, which were misleading as to the correctness of the specifications. A common case was to use an empty set for an initial state variable, which often did not adequately exercise the predicates within the specification (for example, universal quantifiers over that state variable became vacuously true). Of course, this lack of completeness is a fundamental limitation of testing, but most of the problems observed here resulted simply from poor test case design. A contributing factor may be laziness—it is simply easier to type in an empty set than a non-empty one.

At least, having seen these common problems, it will be easy to alert students to them in future—ensuring that they do not succumb will, of course, be much harder.

The course has been run along similar lines since 1998, but with a few improvements. In 1999 the textbook was changed to ‘The Way of Z: Practical Programming with Formal Methods’ [29] and the Jaza‡‡ animator was used in addition to Z/EVES. In 2000, some material on programs-as-predicates and refinement was included, replacing the Hoare logic material. However, modules 3 and 5 remain popular, due to their practical nature.

To conclude this section, a few interesting student reactions to the redesigned course are included. These were recorded on anonymous course survey forms in 1998.

- ‘Interesting concepts introduced. NEW IDEAS, possibly some ideas that aren’t used in the industry yet are taught in this course. Good thing, but possibly a bad thing also.’
- ‘Course overall was good. The testing and assertions sections are relevant to getting a good job and maybe Z will be in the future as well.’

In response to the question ‘How useful do you think your experience from this course will be in your future career?’, some students wrote:

- ‘Extremely useful. My understanding of testing is improved. ‘Writing more solid code’ has had an impact on all of my work this year and I’m sure it will continue to improve the standard of my work.’
- ‘Very. Gives me a good understanding of how programs (especially large and complex ones) should be developed.’

Of course, a few students were less convinced. One wrote:

- ‘Actual application may be minimal, but at least I can say I know the basics of formal methods.’

One student wrote as he submitted Assignment 3 (test set generation from Z specifications):

- ‘I found this assignment challenging and exciting. It did take a long time and even now I am not sure if I sent you my most up-to-date or correct files (I have test cases everywhere) but it has helped me to understand Z schemas, Z/EVES and the idea of testing. While I don’t think I could ever be a tester for large systems, I found testing programs in this structured way interesting and at times captivating (as I tried to hunt down errors).’

6. CONCLUSION

This paper has described how incorporating a variety of specification-based testing techniques led to a measurable improvement in a formal methods course, including increased student satisfaction, and much higher ratings on usefulness. The main techniques emphasized were: testing specifications to validate them, specification-based test generation; and specification-based assertion generation to improve the effectiveness of standard testing.

The authors agree with Jones [30] and Jackson and Wing [31] that a ‘formal methods lite’ approach is a key to applying formal methods to (non-critical) everyday software development. The six ‘lite’ techniques described in this paper need to be more widely used in the software industry as well as taught in universities. The authors urge other researchers and tool developers to begin or continue working in these important areas, including

- support for integrating examples into formal specifications and machine checking their correctness;
- animation of specifications, with graceful fallback for non-deterministic specifications;
- tool support for generation of test sets from specifications; and
- automatic or semi-automatic generation of assertions from specifications, or integration of specification notation into programming languages (e.g., JML).
Areas of potential improvement to the course include the provision of better tool support for some modules, such as the test set generation practical, and the use of JML [7] as a specification language for object-oriented components.

REFERENCES


