Teaching Formal Methods with Perfect Developer

David Crocker
Escher Technologies Ltd.
dcrocker@eschertech.com

Abstract. In order to teach formal methods successfully, the subject must be presented in a way that students find relevant, manageable and rewarding. This position paper discusses some of the difficulties in teaching formal methods and suggests solutions. We present an outline of the *Perfect Developer* tool and describe how we teach our own employees to use it for formally verified software development.

1 Challenges facing Formal Methods teaching staff

Formal methods can be successfully taught if the students are willing, the concepts are within their reach, they are given motivational feedback and they are provided with good tools. Each of these areas presents challenges.

1.1 The difficulty of attracting students

In many universities, formal methods are taught as an optional module that competes with other modules for students. Several lecturers have informed us that the formal methods module has been less popular in recent years. Based on our discussions, we postulate the following reasons for this:

- Students are more focused on gaining skills that industry demands;
- Formal methods are seen as out-of-touch with the real world of object-oriented software construction;
- Formal methods are perceived as difficult and only for the more mathematically inclined.

Although there is little we can do about the first of these (other than accelerating the spread of formal methods into industry), the other two can be addressed by a suitable choice of methods and tools.

We also believe that the name "Formal Methods" is itself unattractive because the word "formal" has connotations such as inflexibility, bureaucracy, and absence of creativity. We prefer to use the term "Verified Software Development" (or even "Bug-Free Software").

1.2 The need for mathematical skills

Many formal methods require their users to be skilled in mathematics on two counts:

- The notations used are often based on mathematical notation. For example, although most programming languages use some variation of the symbol "&" or the keyword "**and**" to denote conjunction, many formal languages insist on using the mathematical symbol "∧" instead[1].
- Most tools supporting formal methods require the user to assist in constructing a substantial proportion of the proofs needed to discharge the verification conditions.

Many British universities no longer require entrants to their Computer Science courses to have 'A'-level or equivalent mathematics. We have observed that in industry, only a small proportion of software developers are comfortable with mathematical notation. Therefore, unless formal methods are to be the preserve of the elite, formal notations should borrow syntax from programming languages instead of logic and predicate calculus.

The need for the user to assist in constructing proofs not only makes formal methods hard to use but also renders them unsuitable for most industrial applications due to the detrimental effect on productivity. The solution to both the usability and the productivity problems is to make the construction of proofs entirely automatic.

[1] The author, being unused to mathematical notation until a few years ago, used to have difficulty remembering whether this symbol denoted conjunction or disjunction – and it has just taken more than ten minutes to find this character in any font supported by the word processor.

1.3 Motivation

Students and software developers need to be provided with positive feedback at regular intervals to motivate them to continue working hard. One of the best forms of positive feedback is the satisfaction of seeing a completed program or subsystem running correctly. The satisfaction provided by completing a specification is tiny in comparison (although some satisfaction may be gained by successfully animating a specification). Formal methods fail to provide sufficient feedback on two counts:

- Many formal methods stop with the development of a specification;
- Where a formal method does go right through to code (or a course includes manual implementation of formal specifications), it is likely that the running program is produced after a very long time, or that it is a trivially small. In contrast, if software is developed without using formal methods, techniques such as Extreme Programming can be employed, which provides a running program very quickly (albeit with little functionality) and regular increments to the functionality thereafter.

The motivational benefits of Extreme Programming can be realised in a formal context by introducing the paradigm of Extreme Specification[2], in which the functionality of a specification is incrementally improved. Positive feedback may be provided at each stage by way of formal verification, animation, or (preferably) generation of a program from the specification. The latter requires a tool that provides for automated refinement of specifications to ready-to-compile code.

1.4 Tool support

Recent years have seen huge advances in the software tools supporting mainstream programming languages and methods. Good tools allow the user to capture his/her ideas rapidly, process them according to the semantics of the language concerned, and provide diagnostics that help the user to pinpoint problems quickly. Tools supporting formal methods should ideally be as slick and helpful as the best integrated development environments, rather than appearing to be remnants of yesterday's technology.

2 Overview of *Perfect Developer*

Perfect Developer (which started life as the *Escher Tool*) facilitates the process of writing specifications, verifying them, refining them to code within the same notation, verifying the refinements, and translating them automatically to ready-to-compile code in a standard programming language. This is also the process supported by the B-method [1]. However, *Perfect Developer* is based on the concept of classes familiar to object-oriented software developers rather than on an abstract machine paradigm, and focuses more heavily on automating proof and refinement.

The product was created with the aim of making formally verified software development affordable in industry at large. Its design goals included:

- It must be easy for the majority of software developers learn and use it;
- It must deliver high productivity;
- It must use the object-oriented paradigm that has become standard in most of the software development industry;
- It must be interoperable with modern programming languages and tools.

To meet these goals, we designed an object-oriented language for software specification and refinement [2]. Although the language has the power of first-order predicate calculus (and a few higher-order facilities), it is based on programming language syntax rather than mathematical symbols. It supports single inheritance with single dynamic dispatch.

Verification conditions are generated and proofs are constructed by a non-interactive theorem prover. This design decision was taken because productivity is severely degraded if even a small proportion of proofs require manual assistance. For this approach to be viable, the prover must be successful on almost all the verification conditions that correspond to true theorems; in practice it achieves a success rate of around 98% in commercial applications. The proofs can be output in a choice of human-readable formats.

[2] This name was suggested by Helen Treharne of Royal Holloway University.

As well as allowing manual refinement to be expressed in the language, the tool will attempt to refine specifications automatically. Either way, the system then refines them further using an internal notation that extends the visible language with lower-level concepts that parallel programming language features. From this notation they are translated directly to the target programming language (Java and C++ are currently supported). Hence we achieve a fast turnaround from specifications to ready-to-compile code, allowing for fast prototyping and the use of agile software development approaches such as Extreme Specification.

In recent years the Unified Modeling Language (UML) has become the dominant notation for graphically depicting the structure of object-oriented software systems. We therefore provide a mechanism to generate skeleton specifications from models exported by most UML tools.

Perfect Developer requires as its host a fast PC running a recent version of either Windows or Linux. More information is available on the Internet [3].

3 How we teach Perfect Developer to our staff

Of the staff that we have trained in *Perfect Developer*, most joined us within five years of graduating and had already learned an object-oriented programming language. Very few showed any inclination to develop a detailed specification from requirements before plunging into coding. Therefore, we needed to teach them not only the notation of *Perfect Developer* but also to specify instead of writing code.

The order in which we have taught topics has generally been based on the following outline:

a) The type structure of *Perfect*. The basic types can be covered very quickly; a little more time is needed to explain the differences between sets, bags and sequences.

b) Expression syntax. We introduce the **forall** and **exists** quantifiers as new expression types and initially apply them to collections rather than to complete types (the language provides for both) because this is easier for the less mathematical students to grasp.

c) Gradually we introduce more expression types, setting the student exercises to write expressions that correspond to natural language statements (e.g. “all numbers from 1 to 100 that are not a multiple of 3, in ascending order”). Many of the examples are Boolean expressions (because these will figure largely in the form of preconditions, invariants etc.). The student can use *Perfect Developer* to check that an expression is valid (although not necessarily a correct solution!).

Only when the student has a thorough grasp of writing predicates and other expressions do we continue with the next stage. We consider it essential for the student to learn how to express what he will try to achieve before writing a program to do it; otherwise a student may attempt to use *Perfect* as a high-level programming language by deliberately writing loose specifications and writing refinements to achieve what is really wanted.

d) Classes (including class invariants, methods, constructors and property declarations[3]). The expression-writing skills that were acquired are used in writing the properties and partial method specifications. The class data is initially kept very simple.

e) Postconditions (in *Perfect*, postconditions have a syntax separate from expressions because they have explicit or implicit frames).

The student can now write complete specifications of simple classes, verify them and generate code (using automatic refinement). We extend his/her ability to more complex classes by covering:

f) Recursion variants.

g) Abstract data modelling (designing a collection of variables and constraints that represent the required data in the simplest possible way, with no redundancy).

At this point, the trainee can be put to work developing components under supervision. We typically wait a few weeks before moving on to:

h) Inheritance and dynamic binding.

i) Refinement of methods to code.

j) Refining abstract data to a more efficient representation.

We have an online tutorial [4] under development that conforms to this outline.

[3] A property declaration is a statement of some expected behaviour of the class, component or system.

4 Experiences and conclusion

So far we have trained seven staff in *Perfect Developer*. Of these, three were recruited because they had at least some relevant mathematical background (not because they had to learn *Perfect Developer* but because they would be working on the theorem prover). Of the remaining four, two did not get on well with *Perfect Developer* (in part because we could not persuade them to move from a programming to a specification mentality) and left the company; the other two became proficient and productive within a few weeks.

In the academic year 2002-2003, several universities ran student projects with *Perfect Developer* and one university used it to assist in teaching a formal methods module. We expect another four to six universities to teach formal methods with *Perfect Developer* during the current academic year and we look forward to hearing of their experiences.

References

1. The B-Book: Assigning Programs to Meanings. J.-R. Abrial, Cambridge University Press, 1996. ISBN 0-521-49619-5.
2. *Perfect Developer* Language Reference Manual version 2.10. Available online (October 2003) via **http://www.eschertech.com** by following the "Support" and "*Perfect Developer* self-help" links.
3. Follow the "Products" link from **http://www.eschertech.com** (October 2003).
4. Introducing *Perfect Developer*: A Tutorial. Available online (October 2003) at **http://www.eschertech.com/tutorial/contents.htm**.