Teaching and Learning Formal Methods, Improving Productivity

J F Miller

Department of Computer Science and Information Systems, University of Limerick

Abstract

This paper discusses requirements for the teaching of formal methods and introduces the notion of productivity in learning. The three attribute model of learning is used to support the use of drag and drop as a universal educational paradigm. An example of this technique is given and a call is made for the establishment of a teaching DIY superstore containing reusable components from which experimental teaching tools for formal methods can be rapidly constructed.

1 Introduction

This paper addresses the issue of increasing the usage of what are loosely called formal methods in the design of software. That systems still fail to work correctly, or even to work at all, is not a situation which reflects credit on the software profession. The new air traffic control system for Heathrow airport and the UK pensions computer upgrade are but two recent examples [1]. The reasons for these debacles are not established and clearly there are many factors involved, ranging from managerial failure to lack of skills on the part of individuals. It is this latter point with which the author, as engineer and academic in approximately equal parts, is concerned.

Since this paper is not about formal methods per se we would like to use an informal definition of the term. Put simply, it is mathematical techniques which describe situations. These techniques are used to make descriptions precise, unambiguous and they must have a calculus associated with them which allows the making of soundly based deductions. In the following we will be mainly concerned with Z.

There are many studies of using formal methods for application development [2]. These studies demonstrate that the methods can be effective even if there is not conclusive evidence of economic benefit. Apart from high reliability application areas, for which mathematical competence is a prerequisite, each study started with the introduction of a method and the training of staff. This took place before the application proper could be considered. Thus it would appear that there is a less than critical mass of practitioners in industry to draw upon and consequently it might be appropriate to ask why this is so.

Formal methods as a research area has been established for at least two decades and courses at undergraduate level have been available for many years. Surely sufficient postgraduates and graduates from these courses should now be available?

In answering this question we must appreciate three factors. Firstly that formal education often provides grounding in the theory while it is industry which provides experience from which grow skills and confidence. This is most certainly so where theory is fundamental to the operation of the industry, as in engineering disciplines which are based on universal laws mathematically expressed. Secondly, and sadly, the software industry is not so mathematically inclined and in many cases graduates can appear effective by substituting, what is often very significant, time and effort for analysis. Thus there is no drive from industry to further develop skills because the situation appears to be under control.

Thirdly the driving forces of a commercial environment are often directed entirely to results, code production, and there is little an individual can do to change management priorities. It requires great confidence to use ones own methods on an informal basis, especially if these delay the start of deliverables, and one is the only person willing to use them.

In addition, the software industry would appear to be driven by a need to keep abreast of the latest versions of intricate and complex artifacts because these are the qualities for which it advertises.

Consequently formal methods teaching subsides into oblivion and the feedback which is given to following cohorts is that provided one can pass the examination no great weight need be placed on lectures in this area.

We therefore suggest that a fifth column of software engineers, who are much more confident and skilled in the use of mathematical techniques than currently, must be developed. The emphasis must be on developing their confidence and skills before they enter industry so that such techniques become an integral part of the graduate armory.

In the following we suggest a method by which the teaching of formal methods could be so improved

and their use widened.

The educational issues which are involved are discussed, learning objectives are specified by applying the three attribute model and teaching material is developed by using a task oriented approach. We will also suggest that the direct manipulation of educational material by drag and drop is a key tool in the improvement of learning effectiveness and productivity.

To provide facilities for experimentation in, and the rapid development of, teaching tools we will discuss the establishment of a DIY superstore of items, themselves formally specified, and suggest a starting possibility.

2 Educational Issues

We as educators must carry our share of the blame for the fact that this situation continues. It might be because of the courses we run, that is to say what is taught, how it is taught and how it is assessed. As part of the latter particular attention must be paid to what it is we actually do assess. What we give marks for is what students will work at, and because they study under severe resource constraints there is no reason for them to do otherwise.

For example Parnas [3] excoriates programming courses on the basis that they teach about artifacts rather than fundamental issues and methods, that is the details of programming languages rather than design principles. Implicit in his criticism is the view that these courses are taught and examined as though code is the sole deliverable. Not only is this true in Computer Science departments but also in Engineering Schools who teach their own programming language courses.

This author would go even further and suggest that in software projects, a major teaching method, the design issues are downplayed. It is here that we commit what is perhaps our most serious crime by allowing unrestricted access to development environments. Small wonder that students are under the impression that success is ensured only by long hours of debugging and that alternatives do not exist.

Similarly Dromey [4] makes the point that we expect learners to write code far too early and that they should read and analyse programs instead. Informal tests which this author has carried out suggest that students do take time to learn the categories of constructs, recognise where they begin and end, and determine their constituent parts. The semantics of constructs and evaluating post environments of program fragments are yet more hurdles. Thus on moving to program writing the basics are incomplete. Furthermore the way in which a design can be generated is not explained, but see Wing [5] for advice, so the starting point is almost always existing imperfectly understood code.

In section 3 we define the terms specification and design which are needed as a precursor to the task analysis for learning. In most texts these items seem almost synonymous with a pseudocode program.

It might appear that formal methods courses could not possibly have these criticisms since they are concerned with mathematics rather than applications. However this need not necessarily be so. The aim of such courses is normally the use of mathematics in making specifications precise.

However if we assess formal methods courses primarily by the writing of specifications, particularly under examination conditions, then we have fallen into exactly the same trap as described above for programming courses. Students will concentrate on the production of specifications, just as they do on code, before they fully understand the mathematics. There is the further complication that the, imperfect, code which is produced in programming courses is, after long debugging effort as discussed above, a valued product in industry. Specifications unfortunately are not.

Lastly specifications have to exceed a minimum size if they are to demonstrate the benefits of mathematical exposition, a size which for beginners makes comprehension difficult and obscures the point [6].

The whole exercise therefore appears to students to be one of taking a difficult theoretical course with no subsequent career value.

In deciding what might be done we must look at learning in more detail. Learning is a value added process, the object being to increase the three attributes of knowledge, skills and understanding in a learner. Knowledge is that which may be retrieved from memory, from ones internal data base, without further outside reference. Skills are procedures which once started may be continued without further thought until concluded. Understanding is more difficult to define, but here we will define it to be the ability to choose appropriate theories in order to make predictions about a circumstance.

By differentiating between these learning attributes we raise the possibility of using different teaching methods, each adapted to the attribute in hand, thus increasing the learning gain. In addition the process of categorisation leads to deeper insights into the learning process and therefore how we might improve the acquisition of design ability.

3 Learning and Teaching

Knowledge, factual material to be recalled at will, consists of the definitions of technical terms, recognition of special notations, such as those in the predicate calculus, and theorems, substitutions or equivalencies.

Skills, processes, involve more than knowing certain facts. The known items must be recognised in order that the correct procedure can be invoked. These can be procedures for manipulating expressions such as sequences of substitutions and reductions which lead to simplification, proof or evaluation.

Understanding in a subject domain is demonstrated by going beyond knowledge and skills to the making of predictions about a situation. It requires the ability to choose, from known applicable theories, those concepts which are relevant and to match them to items in the subject area. Then by using relevant relations between the concepts, deductions about the situation can be made. In the case of formal methods the deduced result is a specification and subsequently a design. However we need definitions of these two terms, in particular to see the difference between them and also in order to design teaching material.

A specification is a collection of statements which express truths about a situation. These can be divided into two groups, those which are definitions, and these are usually stated as preliminaries, and those which are always true. An important component of the former are those which define the situation under consideration.

Notwithstanding the existence of refinement methods, the process of transition from specification to code, we will define a design as being; definitions of data structures and major variables; specifications of the code blocks into which the implementation is to be divided; and a specification of the way in which the blocks will be combined.

As we seek to increase these attributes and to improve the rate at which they are acquired by the use of tools, we must be careful to distinguish between two kinds of productivity. Software development tools of various kinds exist and it is tempting to assume that they are applicable in teaching. However we must be clear that two different kinds of person are involved.

A professional tool is designed to increase the productivity of a professional in some production task. It expects knowledge of the subject area, notation, conventions and short cuts and is characterised by a steep learning curve and a long period of familiarisation. There is some tangible output, a program, documentation or whatever.

These assumptions are not valid in the case of a learner for whom the tool produces no tangible output. In addition the complexity of using the tool can come between the learner and proper acquisition of the material being taught. Word processors, logic simulators, network analysers and spreadsheets being prime examples.

In other areas there is a well established principle that, wherever possible, simplified tools are used in order to demonstrate fundamental principles. Where the education market is large they have the advantage of being cheaper. Nevertheless the effort to develop simplified software tools is significant and it is still a research area.

As has already been mentioned productivity in learning must be increased so that the learner acquires confidence in the material as well as a sufficient level to pass an examination. These not necessarily being the same! Similarly productivity is not the same as effort. We do not necessarily require more effort, but more effective use of that effort which the learner expends.

True learning takes place when there is reflection on the activities being undertaken [7]. Reflection implies the sequence of making a hypotheses, evaluating the truth of the hypotheses and then analysing why false ones are invalid. Learning productivity is then the rate at which this sequence can be carried out, the more rapidly this can be done the more rapid the learning.

Although there are various 'methods' [8] for making the process more systematic, normal learning takes place without productivity aids. There are no machine tools, in the manufacturing sense, for learning.

Notwithstanding the availability of computer aided learning there is still no consensus of opinion as how to extract the claimed benefits. Nevertheless we propose here to use the ultimate machine tool, the computer, in various ways to increase the rate at which correctable mistakes can be made by the learner. We will propose different scenarios for the different attributes discussed above and limit the application area to that of formal methods. In addition we propose that the learning material is directly manipulated instead of being assembled from keyboard input.

In a command line interface to an application, or operating system, the command name and parameter syntax must be remembered and entered correctly and for many this is a frustrating task. Trying to recall the exact name of a command or file one knows exists, but cannot bring to mind, is a task which should be reserved for masochists and not for computer users. Icon driven interfaces and those which permit items to be manipulated directly do not suffer from these drawbacks. Note that we clearly distinguish between the interface and the operating system itself and do not wish to enter into value judgments on the latter.

A large part of formal methods teaching is the learning and use of notations. Thus the hypotheses to test both knowledge and skills have a syntactic structure which can be examined to determine whether they are valid or not. In addition the components to be placed in the correct syntactic position can be chosen from a set

Teaching and Learning Formal Methods, Improving Productivity which contains both valid and invalid alternatives. However a command line interface requires that the learner type text. Only when the line is complete can it be parsed, any errors identified, and the learner's misconceptions deduced from the error pattern. A complex process acting on a large error space.

In contrast direct manipulation permits choices to be made from a palette of alternatives and drag and drops to be made to positions in a syntax diagram. This reduces complexity for the user and also in the determination of the correct tutorial responses. It allows feedback at each step because the steps are independent of each other, the correct responses are known and the error space is controlled. Most importantly the rate at which errors can be corrected is increased.

As an example we refer to the layout below, taken from [11] which is to be used for practice in writing declarations. The source palette contains possible components for the required declaration and some distracters. The targets are the token positions in the declaration syntax diagram. In addition the learner thinks in higher level 'chunks' than when typing characters at a keyboard.

The classic, apparently unpublished, paper on methods of this kind is by Andreae [9] who first used direct manipulation in teaching fundamental electric network transformations. A certain class of networks can be reduced to a single component and one source by combining sources and components in certain ways. Each reduction is accompanied by a calculation so as to ultimately give the final source and component values. The usual approach to consolidate learning is to practice on test networks and compare the two final numerical values with those given in tutorial material. These two values alone do not provide enough information for the learner to find the sources of errors without redoing the reduction from the beginning. This is because at one or more stages there might have been wrong choices of components to reduce, which is a pattern recognition problem, or mistakes in calculations, or both.

Andreae provided a display of the network which was then redrawn after each transformation. The components being reduced were selected by mouse clicks on the diagram and the transformation to be applied to them was selected from a palette. In this way the error space was restricted and immediate feedback at each reduction could be provided. Another important factor in increasing effectiveness was the way in which direct manipulation allowed the problem solving aspects to be turned into a game. Consequently the learner could assess learning progress. This idea has been investigated for some aspects of discrete mathematics [10].

Using the attribute model and direct manipulation we have criteria for categorising the material to be taught and a teaching framework in which productivity of learning can be improved. We now look at how the material should be organised.

4 Task Analysis for Teaching

In this section we look in more detail at the material to be taught and adopt a task oriented approach. Instead of listing the topics which are regarded as important we attempt to specify what it is that a qualified person should be able to do. If we take declarations as an example then in a topic oriented approach we specify the data types and syntax as material to be taught. Tutorial material then consists of specifications of declarations

Teaching and Learning Formal Methods, Improving Productivity which the learner has to construct. These are then checked against solutions. The task approach, in contrast, identifies what must be done to write a declaration. This is; determine the required data type and then the keyword for that type; choose a meaningful identifier for the variable name and finally place these in the correct syntax to make a valid declaration. These actions require the appropriate knowledge to be performed correctly.

The learner must be able to do two other things which are higher order tasks. Firstly to identify incorrect declarations in case a mistake has been made, this is a skill, and secondly given a description of a situation to identify a relevant variable in order to construct a declaration. The latter requires understanding, understanding of the use of variables and the purpose of declarations.

Hence task analysis identifies what has to be taught in the categories and the direct manipulation diagram method, above, can provide learning in all these aspects by suitable choice of question, source palette options and target syntax. Analysis of learner errors is simplified because the properties of each target and source are known and therefore remedial advice is more easily constructed.

It is not the purpose of this paper to go into great detail on formal methods topics we will simply give examples of the kind of material which might appear in the various attribute categories.

Factual material consists of the names of symbols, the different parts of constructs such as predicates, set descriptions, schemas, together with notational conventions such as decorations, delta and theta for schemas, and elements of the schema calculus. All this is concerned with learning the alphabet of the mathematical language and is therefore a collection of pattern recognition problems. As in all language learning practice with feedback is the key to success.

For skills we have the use of, what is the now known, alphabet in recognising and constructing phrases and sentences. Evaluating expressions are simple examples while decoding set constructions to produce specimen members, evaluating predicates, evaluating schemas given an environment are more complex. Translating both ways between narrative text phrases and their mathematical equivalents is a major skill required in the understanding of specifications.

Understanding, and hence confidence, is a prime requirement for our fifth column. For this the learner needs to be able to relate the specification to a text description of a situation. Variables, events, pre and post conditions, sequences of events and their effects must be identified. To adopt Dromey's approach the learner should be given specifications and asked to find relevant items in text descriptions well before there is any attempt at synthesis. Similarly synthesis is best approached partially, that is from an incomplete specification, or from one with errors, which must be completed or corrected.

If formal methods of one kind or another are to become a key weapon for graduates then there is much research to be done both in teaching methods and the development of software tools for improved productivity. This requires considerable effort. And in the next section we outline a way for easing the situation.

5 DIY Super Store

Software tools for teaching formal methods, although subject to the same principles as in other areas, are themselves a topic for research. Their development requires considerable investment and often the same aspect has to be reimplemented in different experiments.

We propose here a rapid method for implementing teaching experiments by using a DIY superstore of basic items which anyone may use and to which everyone is encouraged to contribute.

A hardware DIY superstore provides a copious supply of materials, components and tools. There are many examples of the same item all with variations and adapted for niche uses. For example there are screws with different heads, of different materials and finishes, different drive methods and for different purposes. Almost every item is a partial solution in some larger scheme and there are very few complete items, perhaps flatpack furniture is in this category, which can be installed directly without further work. Even for tools there is a plethora of kinds and manufacturers all providing specialist facilities and competing for attention.

Buyers can browse the store and, in contrast to using a builders merchant or factor where one has to know what one wants, can 'try by eye' to find solutions to ill defined problems. Indeed, many users say that it is the freedom to browse, and the variety of material on view, which generates solution ideas and often leads to components being used in novel ways. Ways not thought of by their makers. Manufacturers stimulate this environment by providing a constant stream of improvements, new items and new methods whose worth is rapidly evaluated by buyers themselves.

We propose a DIY superstore specifically for teaching aspects. This provides software items on the same basis as above, except that there are some close prescriptions on the format of the goods which the store stocks. This format is best described by considering the item categories of materials, components and tools. However let it be stressed that success of the superstore concept requires that many variants and versions of the same basic item are available.

Specifications are the raw materials from which useful things are constructed. We would have store

Teaching and Learning Formal Methods, Improving Productivity areas for different formal methods and each item would be an example of the use of that method, together of course with the normal plaintext description. Simply to specify some aspect, an interface, even an object as simple as a predicate, in a way that will ultimately lead to an implementation is both extremely useful and academically respectable. Since we will be using machine representations of specifications then this is an ideal candidate to specify along the lines of a hypertext document [12].

Implemented items form components which are combined into larger units, with or without being modified in some way. However for maximum benefit we require that the implementation is in source with a totally integrated formal specification and design and accompanied by an analysis of extensions. This ensures that the component can be fully understood by the user, or if not it can be explored in a structured way, and that modifications are implemented via respecification rather than by immediate code changes [13]. The modified component then becomes a candidate item without further work. Some contributors may choose to implement directly, or extend and implement, superstore specifications. A selfless approach.

A tool is a productivity element, although in this paper there is not a rigorous distinction between a tool and a component. We will simply use the term to mean an item which makes the construction of teaching software experiments easier. In practice this may mean items such as specification viewers, syntax checkers, interfaces for input of specifications— for use by tutors rather than by students — data dictionaries to keep track of specifications and evaluators of various kinds. In maintaining the superstore concept they must be small, accompanied by specifications and transparent for understanding and modification. In contrast the elegant viewer of [14] which, although a powerful tool, may require significant learning effort before becoming effective in a teaching role.

A showcase, such as proposed, allows contributions both from academic staff and proles (see Diller's joke [15]) depending upon the topic and level. At one end of the scale almost a journal paper and at the other a final year project, term paper or a Masters assignment. However although we have defined the requirements for superstore stock this paper does not deal with quality control of the items on display.

There is a clear distinction between the superstore concept for development of formal methods teaching software described here and that of a function library as is provided by many programming languages. Function libraries are meant for immediate use, usually implement known algorithms and therefore require no formal specification in the sense used here, are not transparent— particularly if proprietary— and are not open to modification. The closest analogy is to Internet user groups, devoted to a particular machine, language or hobby, who exchange tips and hints.

6 Conclusions

In this paper we have argued for an improvement in the teaching of formal methods on the basis that they must become a crucial component in the graduate armoury. We are not in any way arguing that they provide the magic bullet, but that a professional is not properly qualified unless he is competent in the mathematics of his discipline. As an engineer the author would even support Parnas [16] in arguing for more science in Computer Science.

We have looked at the material to be taught in a different way and we have argued that direct manipulation holds advantages in computer assisted learning productivity. We have also made a call for more research into the teaching aspects of formal methods and proposed a mechanism for distributing the development load.

7 Acknowledgment

The author wishes to acknowledge the Head of Department for encouragement in publishing this paper.

8 References

1. Guardian and Independent newspapers, various dates Jan–Mar 1999.
2. Panel Session. In: ZUM '95 Proc. 9th Int.Conf. of Z Users, Limerick, 1995.
3. Parnas DL. Teaching Programming as Engineering. ZUM '95 Proc. 9th Int.Conf. of Z Users, Limerick, 1995.
4. Dromey RG. How To Solve It By Computer. Prentice Hall, 1982.
5. Wing JM. Hints for Writing Specifications. ZUM '95 Proc. 9th Int.Conf. of Z Users, Limerick, 1995.
6. Diller A. personal communication.
7. Freudenthal H. Mathematics as an Educational Task. D Reidal 1976.
8. Buzan T. Use Your Head. BBC Books, 1974.

9. Tomlinson M. Circuitbreaker, A game for teaching circuit simplification. In: Adreae, J. H. Progress Report UC–DSE/39 (1990), Department of Electrical and Electronic Engineering, University of Canterbury, Christchurch, New Zealand.
10. Wiltshire D. Aid for Teaching Z Mathematical Toolkit. Final Year Project Report (unpublished), 1996 School of Electronic and Electrical Engineering, University of Birmingham, UK.
11. Kelly J. Programming Language Teaching Aid. Final Year Project (in progress), 1999, Department of Computer Science and Information Systems, University of Limerick, Ireland.
12. d'Iverno M. Priestly M Structuring Specifications in Z to Build a Unifying Framework for Hypertext Systems. ZUM '95 Proc. 9th Int.Conf. of Z Users, Limerick, 1995.
13. Edmond D. Information Modeling. Prentice Hall, 1992, p328.
14. Mikusia L. et al, Z Browser – Tool for Visualisation of Z Specifications. ZUM '95 Proc. 9th Int.Conf. of Z Users, Limerick, 1995.
15. Diller, A. Introduction to Formal Methods. John Wiley and Sons 1990, p48.
16. Parnas DL. Education for Computing Professionals. Computer, Vol 23, No 1, 1990.