

Issues in the teaching of Z

J.H. Turner

*School of Computing, Staffordshire University,
Beaconside, Stafford, ST18 0AD, UK*

Abstract

Z has been taught at Staffordshire University for over five years, and to more than 1500 students. Experiences arising from this work are described and the following five key issues are discussed. Do students see Z as *computing* or as *mathematics*? Do students have the prerequisite skills in abstraction, algebra, modelling and problem solving needed for Z? How can work in Z be successfully assessed? Do students understand *proof*? Can Z cross-fertilise skills in other areas of computing? These issues are described and ways to address them are suggested.

1. Background

Z has been taught at Staffordshire University since 1988. It has been taught throughout the curriculum: on both degree and HND programmes; on Software Engineering, Computing Science and Information Systems courses; to first, second and final year students; in modules ranging from those concerned with the practical modelling of systems to those concerned with more theoretical work in proof.

In total at least 1500 students have been introduced to Z, with in excess of 100 person years of student time and five person years of staff time being devoted to the subject.

Our experience is that formal methods are unpopular with students. They are perceived as being difficult, as lacking in vocational relevance and as being unexciting. However, we argue that the study of Z benefits students. We make the case for this below.

2. Why Z?

The reason for using Z rather than VDM, say, is mainly historical, being based initially on the backgrounds of the staff who were available when the choice was made. Since then a considerable investment - in terms of teaching materials, and including a course text [1] written by the teaching team - has been made specifically in Z, and this makes change less appealing.

Beyond that, though, is the deeper question: why teach a formal method? Formal methods were introduced when it was decided to provide more robust software engineering foundations to our computing courses. "Hacking" was to be discouraged and a more methodical approach to software development encouraged. Formal methods were only part of this change. It included an emphasis on "design" right from the start of introductory programming courses. An additional attraction of formal methods was that, because they were more directly relevant to computing, they were expected to be more appealing to students than the more traditional mathematics that they replaced.

Our reasons for teaching formal methods have changed somewhat. They now have more to do with the student learning process and with the knowledge expectations of computing graduates, than with the advantages of using formal methods for software development *per se*.

Although the methods advocated in software engineering have enormous benefits for experienced practitioners, it is not clear whether they represent a good way to learn the subject in the first place. For a novice the act of writing a program is in many ways a learning process, not just about the application being worked on, but also about the language he or she is using and, indeed, the process of programming itself. Learning often comes from experiment and, it is suggested, this is unlikely to be maximised by forcing students to follow a rigorous regime of writing formal specifications, design documents and test plans. Moreover, quite naturally, students want to "see" results as quickly as possible, so enforcing from the start good SE practices is unlikely to increase their motivation, because it may delay the time before any results are visible.

In spite of the reservations described above it is felt that there is still a need for formal methods to be taught from the start of a course. This is because the study of formal methods can provide some of the strong foundational skills needed for students to be successful at computing. These skills include:

- abstraction;
- modelling;
- problem solving;
- notation.

Experience shows that Z forces students to get to grips with these issues to a greater extent than appears to be the case with other subjects.

A further "need" for Z is to do with expectations. Something like 70% of all computing degree students now study Z, so it has become something that is almost expected of computing graduates.

Where Z replaces a subject that has less academic kudos, it serves to increase a course's academic validity. However, if there is an internal market that gives students a large choice as to what modules to take, a formal methods module may only be taken by a tiny minority of them. This fact should be taken into account when assessing the validity of a course as a whole.

We go on to describe five issues that are important in the teaching of Z.

3. Issues

Do students see Z as *computing* or as *mathematics*?

Many computing students suffer from a phobia of mathematics. Therefore, many will be worried by a module that carries a "maths" label. Consequently, we try to teach Z in as *maths-free* a way as possible. Thus, proof is not dealt with in initial courses, real systems are modelled as soon as possible, and diagrams are used frequently. However, it is our experience that many students still see Z as being mathematics and, almost in consequence, do not like it.

This raises important issues about what students perceive as being within the scope of mathematics. For many, common programming activities, such as finding the lowest number in an array, is considered a mathematical activity. Such an extremely wide scope attached to mathematics may explain the popularity of courses on the interface of computing and business, which omit most technical computing issues.

This raises the question: are we, in trying to hide the mathematics, only making things worse? Perhaps we should make it clear to students that certain mathematical skills are required if computing is to be properly studied and that these skills are essential to any practitioner of the subject.

Do students have the prerequisite skills in abstraction, algebra, modelling and problem solving needed for Z?

It is possible in a short time to teach enough Z to enable students to model complex systems. However, such rapid progress is only possible if students have basic prerequisite skills in mathematics. For instance, although the use of brackets is not usually thought of as part of Z, students unfamiliar with the notation will find the study of Z much harder. Or again, although abstract model building is by no means unique to Z, students unfamiliar with the concept will have much more ground to cover.

It is possible to get a GCSE pass, with a C grade, in Mathematics without doing any algebra. This represents an insufficient foundation for the study of much of computing. Students with such a qualification would benefit from additional study in basic mathematics. But to be worth doing it would have to

396 Software Engineering in Higher Education

be done well, surface learning would not be useful. This would take time, perhaps one fifth of a year's work even for students with an aptitude for the subject, and would probably be unpopular with just those students who needed it most.

How can work in Z be successfully assessed?

To a large extent, from the student perspective a course is determined by its assessment. Barnett [2] writes: "the evidence is that students, in forming their learning strategies for a course of study, will take the examinations as their point of departure and assess what has to be achieved in order to avoid failure". Thus, if a course can be passed on the basis of surface knowledge, we should not be surprised if this is the knowledge that students attempt to gain. Moreover, if a course has an assessment regime that is harder than others, its subject content will be branded as being difficult, not just its assessment.

The most obvious way to assess students' work in Z is to use assignments in which students are asked to specify systems. This is useful in that it appears that students are engaged in a practical activity. However, where the assignment is part of the summative assessment, and particularly where it counts towards a degree classification, there are problems:

- It is impossible to guarantee the integrity of the work. There is a whole spectrum of possible activities, ranging from the direct copying of work to the unacknowledged working together in teams, which reduce the learning experience. Such activities lower students' motivation and confidence, and make it impossible to know whether they have the prerequisite skills needed for later courses. Additionally, given that one of a university's roles is as a certification agency, there is an external necessity that the integrity of the assessment process can be guaranteed.
- There is a tendency for students to put more effort than they should into documenting the work. Although essential in industry, time spent repeatedly documenting work is time lost in understanding the subject; thereby reducing the effectiveness of students' study time.
- Where there are large student numbers it may be difficult to find the staff time to mark the assessments properly. This has indirect effects as well: it tends to increase the time before students get feed-back, thus reducing its effectiveness; it becomes difficult to justify the accuracy of the assessment process, leaving open the possibility of appeal.

However, an alternative can be used. Z lends itself to multichoice testing. This does not need to be at a surface level. Tests can be designed which assess much deeper skills. One way to proceed is for students to be given a scenario and a rough version of the state schema following from this. Then, each question is formed by providing in English more information about the system,

and this is followed by various Z statements which purport to describe this. Students are asked to indicate which of the alternatives describe the new information fully. With large classes this approach has considerable practical advantages; being objective and quick to mark. The author would argue the marks are valid, with the scores correlating well with other, subjective, information about students' performance. Moreover, it has been found that students are motivated by using these questions as part of the learning process. This is because they give students immediate feedback. A problem found by students when working on a specification on their own is that it is difficult to get good feed-back; even model answers being of limited use because students may have gone off on different, but equally valid, tacks to that indicated by the answer. Given the answers to multichoice tests students will usually know where they have gone wrong, but some of the time they will not. Such occasions give the opportunity for a focused and, often, highly effective teaching session with a lecturer,

Do students understand *proof*?

Mathematical proof techniques are now taught less in schools. Consequently, most students who start a computing course without having Mathematics 'A' levels, and many who have, will have little idea of what is meant by a mathematical proof; such students find a rigorous treatment of formal methods very difficult, indeed many find the concept of proof itself difficult to grasp, often confusing proof and testing.

This raises the question of how best to teach proof techniques. It is the author's view that Z does not provide a good vehicle for this, and that the old-fashioned school approach of teaching geometry is actually better, because the domain is, in a sense, less abstract. However, few courses will find the time, perhaps a tenth of a year's teaching, to do this. But, without these foundations the area of proof will be permanently closed to students.

It could be argued that this is unimportant, because practitioners rarely formally prove the properties of systems, but this misses the point: it is the way of thinking that proof techniques generate that is important, with the connection between proof and design being profound.

Can Z cross-fertilise skills in other areas of computing?

Z does not exist in isolation from other branches of computing. Apart from the foundational skills described above, it can be used as a way to illustrate other topics, such as recursion.

Students often find recursion difficult. One reason for this is that, after the obvious factorial example, recursion is often illustrated using pointers. Thus, two difficult concepts are met at the same time. This problem disappears if a functional programming language is used, but introducing such a language early in a course would be a major change. Z allows recursion to be dealt with in a relatively clean environment, with sequences providing fertile ground for examples of recursion. To begin with students can be given a function and asked

398 Software Engineering in Higher Education

to "dry-run" it to find the result for a particular value. Our experience is that students find this far easier than designing recursive specifications themselves.

4. Conclusion

Z deserves a place on any computing course, because it is sound for both pedagogic and academic reasons. However, if it is present in a course, many questions need addressing.

Z is a very good vehicle for teaching some of computing's foundational skills, but it does itself require prerequisite (sub-foundational) skills. These are at no higher a level than would have been covered in 'O' level Mathematics, but students with only a GCSE in the subject cannot be guaranteed to have them. (This is particularly the case with algebraic skills.) Such a lack of foundational skills is not inevitable. We have taught many foreign students, and it is noteworthy that in the main they, especially those from France, have gained a much stronger mathematical background at school and, in spite of language problems, they have performed much better, on average, at Z than British students. This suggests that things could be improved if schools adopted a different approach.

By its very nature, Z is discriminating. It is likely, therefore, that any realistic criterion-based assessment method will lead to marks with a high variance, and with students who have made unsatisfactory progress being clearly identified. This raises the issue of what happens on courses where many students fail? This will be of particular concern if other subjects are less discriminating and have far fewer students failing.

If students find a subject difficult, they will usually prefer to do something else. Z is not perceived by many students as being an easy option (or, to honest, vocationally relevant or exciting), so given the choice many would choose not to do it. Thus, even though lecturers may feel Z "is good for" them, should students be forced to study the subject?

One way to address these problems would be through the introduction of an agreed common SE curriculum that formed part of all computing courses, and which was assessed by common national examinations.

References

1. Rann, D., Turner, J. H. and Whitworth, J. *Z: A Beginner's Guide*, Chapman and Hall, 1994.
2. Barnett, R. *Improving Higher Education: Total Quality Care*, The Society for Research into Higher Education and Open University Press, p 207, 1992.