Teaching Introductory Formal Methods and Discrete Mathematics to Software Engineers: Reflections on a modelling-focussed approach

Andrew Simpson[1] [0000−0003−3597−2232]

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD
United Kingdom
firstname.secondname@cs.ox.ac.uk

Abstract. Much has been written about the challenges of teaching discrete mathematics and formal methods. In this paper we discuss the experiences of delivering a course that serves as an introduction to both. The one-week intensive course, *Software Engineering Mathematics*, is delivered as part of the University of Oxford's Software Engineering Programme to groups of professional software and security engineers studying for master's degrees on a part-time basis. We describe how a change in the course's emphasis — involving a shift towards a focus on modelling-based group exercises — has given rise to some pleasing results.

1 Introduction

Much has been written about the difficulties of teaching discrete mathematics and formal methods, with problems associated with 'getting' abstraction, student motivation and what might be termed 'math-phobia' being recurring themes. Proposed solutions include the utilisation of a 'stealth-like' approach [19] ("we sneak up on our blissfully unaware students, slip a dose of formal methods into their coursework and development environments, then with a thunderclap disappear in a puff of smoke" [19]), a clear justification [29], a considered approach to links with the rest of the curriculum [30], and a focus on modelling [4].

Our focus in this paper is a one-week intensive course, *Software Engineering Mathematics*, which is delivered as part of the University of Oxford's Software Engineering Programme[1] to groups of professional software and security engineers who are studying for master's degrees on a part-time basis. The course aims to do two things: introduce students to formal methods and teach them core discrete mathematics concepts (in a fashion similar to, for example, the courses described by Warford [39] and Jaume and Laurent [18]).

Teaching part-time students who are predominantly drawn from the software engineering industry has its advantages when compared to teaching full-time undergraduate students — such students bring 'real-world' experience and problems to the classroom, which helps those delivering courses to make connections

[1] http://www.cs.ox.ac.uk/softeng/

between theory and practice, and to demonstrate potential benefits. In addition, such students tend to be very motivated — the financial and time investments required are, after all, significant. (The overall course costs are approximately £25K. In addition, the students are required to spend 11 weeks in Oxford, and commit several hundreds of hours to assignments and project work.) On the other hand, there are complexities associated with teaching such students: the diversity of prior academic and industrial experience, as well as a diversity of expectations, can make for an extremely heterogeneous mix of participants. A further challenge involves demonstrating that an appropriate application of the techniques being taught is relevant to the students' everyday activity — and, as such, justifies the aforementioned investments.

Of course, the difficulty of demonstrating the 'pay-off' of many Computer Science and Software Engineering tools and techniques is a challenge that has been recognised widely. For example, to quote Finkelstein [10]:

> "Software engineering is, in large part, about scale. Illuminating the essence of a software engineering technique and motivating the students with convincing arguments for its value, without giving examples which are so large as to submerge the student in extraneous detail is extremely difficult." [10]

The philosophy of the course under consideration in this paper is sympathetic to the view that an 'appropriate' and 'within context' application of formal and mathematical techniques is essential to demonstrating their potential value to professional software engineers. In many ways, this is consistent with the argument put forward by Woodcock *et al.* [42]:

> "One of the main difficulties in engineering is the cost-effective choice of what to do and where. No engineer gives the same attention to all the rivets: those below the waterline are singled out; similarly, a formalism need not be applied in full depth to all components of an entire product and through all stages of their development, and that is what we see in practice." [42]

In this paper, we show how a change to the emphasis of our Software Engineering Mathematics course — involving a shift towards a heavy focus on modelling-based group exercises — has given rise to some positive results. Our aims have much in common with those of Larsen *et al.*, who, in [23], describe "experiences developing and delivering courses that endeavour to equip students with generic skills of abstraction and rigorous analysis by means of lightweight formal methods using VDM and its support tools." Further, our journey has much in common with that described by Cowling [3]:

> "The starting point for this experience was the approach of teaching Z as a formal specification method, as presented in the standard textbooks. The problem that was soon found with this approach was that these texts did not suggest any method for constructing specifications, but instead focused on the various mathematical constructions that could

> be employed in the specifications. This focus left the students feeling a bit like the audience at a magic show, asking the question 'where did that bit of the specification come from', meaning that they were gaining little understanding of how they could actually use such methods themselves." [3]

In Section 2 we briefly consider related work. Then, in Section 3, we discuss the context of the contribution. In Section 4, we reflect upon how experience led us to the restructured version of the course that we now use. In Section 5 we present some indicative (and caveated) results. Finally, we conclude in Section 6.

2 Related work

Our focus is a course that exists at the academic–industry interface. This is an area covered by a number of authors, including Mead *et al.* [26], Fraser *et al.* [11], Vaughn and Carver [38], Subrahmanyam [36], and Almi *et al.* [1]. In addition, in a series of papers [13,14,28], Taguchi and colleagues discuss their experiences of educating Software Engineering professionals in Japan.

The importance of abstraction and modelling[2] to the practising software engineeri is recognised widely ("We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad cases is called 'abstraction'; as a result the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer" [8]; see also [7] and [40]); the difficulties of teaching abstraction and modelling is also acknowledged by many [16, 20, 22]. To quote Fincher and Utting [9]:

> "we know that abstraction is a very difficult step to take ... that learners find it difficult to grasp the principles embodied in a single example (or a series of single examples) then isolate it as the common referent they all share (that is, abstract from the details to the principle) and apply that principle in novel situations." [9]

Addressing these challenges is at the heart of this paper.

The course under consideration in this paper leverages the mathematical language of Z [35,41], and our contribution discusses the value of case studies. It is worth recognising that there is a rich history of Z case studies: running from the early contributions of the likes of Hayes, Morgan and Sufrin [12,27], through Jacky's *The Way of Z* [17], to more recent contributions such as [37].

3 Context

We now consider the context of the course. We start by discussing the Software Engineering Programme at the University of Oxford and then turn our attention to the Software Engineering Mathematics course.

[2] We would argue that, in this context, at least, the two go hand-in-hand — a 'complementary partnership' in the words of Kramer [21].

3.1 The Software Engineering Programme

The Software Engineering Programme at the University of Oxford, which was established in the early 1980s, built on the University Oxford's experience in delivering one-week intensive courses to professional software engineers in, for example, formal methods such as Z [35] and CSP [31]. An 'integrated programme' of six one-week modules was established in 1993; the Software Engineering Programme now offers one-week courses in over 40 topics. The programme also offers students the opportunity to study for MScs in Software Engineering and Software and Systems Security on a part-time basis.

At present, approximately 300 students are registered with the Software Engineering Programme. Students are drawn from a wide range of backgrounds, including large IT firms, government organisations, small companies, and the financial sector. The programme's requirements for entry are flexible, taking into account prior industrial experience, as well as academic background.

The wide diversity of the student body gives rise to a number of challenges: few assumptions can be made about the nature of previous industrial and academic experience, meaning that the complexities of teaching modelling techniques are slightly different to those associated with teaching cohorts of full-time student that are (typically) more homogeneous. Some of those complexities are discussed in [33], and it is worth reprising those arguments here:

> "The typical student on the Software Engineering Programme 20 years ago was a relatively experienced software engineer, who had been based in the industry for at least five years. This meant that the prior knowledge that one might use in delivering courses was relatively uniform. As an example, when teaching discrete mathematics, one might use a binary tree as a motivating example when discussing recursive functions. Unfortunately, this is no longer true: it is not unusual to be met by blank faces (by even those with a first degree in an IT-related subject) when mentioning binary trees. This is for (at least) two reasons. First, the level of abstraction has been raised: developers don't have to define their own tree-like structures as libraries exist that can be leveraged. Second, the student body of the Software Engineering Programme now reflects the healthy heterogeneity that is the workforce in software engineering, security, and related industries." [33]

Various aspects of the Programme have been written about previously [6,33, 34]. In addition, in [25], the authors considered the relationship between relational database design and the language of Z and explored how the relationship between the two paradigms is exploited within the teaching of the Programme. Finally, the use of a model-driven approach to support the Programme's information system (amongst others) is described in [5].

3.2 Courses and assignments

To gain a Postgraduate Certificate, a student needs to attend and submit an assignment for four courses (averaging at least 50% across all assignments, with no

more than one scoring below 45%); to gain a Postgraduate Diploma, attendance and subsequent submission for eight courses is required (averaging at least 50% across all assignments, with no more than two scoring below 45%); for an MSc, the requirement is 10 (averaging at least 50% across all assignments, with no more than two scoring below 45%), together with the successful completion of a dissertation.

Each course consists of: a period of preparatory study (involving, for example, the reading of one or more research papers or book chapters and / or a small exercise); an intensive teaching week; and a written assignment. Each teaching week involves some combination of lectures and exercise / practical sessions. The relatively small class sizes of up to 18 students typically lead to much interaction between students and instructors. The take-home assignments — which are undertaken over a period of six weeks — allow students to reflect upon and apply the techniques taught during the week.

There are good reasons for this choice of mode of assessment. First, our students often travel from all over the world to attend our courses; to expect them to travel back to sit examinations would be impractical. More importantly, a six-week period in which to undertake an assignment provides students with an opportunity to properly reflect upon the material that was delivered during the one-week course.

3.3 The Software Engineering Mathematics course

The course under consideration — *Software Engineering Mathematics* — attempts to do two things. First, it attempts to teach students key aspects of discrete mathematics — with the mathematical language of Z being the vehicle of delivery. Second, it aims to show how formal models can be used to aid comprehension and communication.

A 'light touch approach' (as per, for example, the philosophy of [15]) is advocated[3], and a realistic view of the success of the impact of formal methods in general (as reflected by, for example, [30] and [24]) is presented. The course text is *Using Z* [41] by Woodcock and Davies; *Discrete Mathematics By Example* [32] is used as a supplementary text for additional examples and exercises.

Anecdotal evidence suggests that the course is seen as 'difficult' by many students: the combination of new concepts and techniques, an unfamiliar language, and the intense pace of a week-long course makes for a challenging experience for some students. In addition, this course is a particular victim of the disconnect between theory and practice — while the techniques taught (thinking abstractly and precisely) are clearly beneficial in the long term, this is not always immediately obvious to the students.

Students gain a passing grade in this subject (50%+) if they can demonstrate that they can use the mathematical language of Z to build simple models; they

[3] See [43] for a useful classification of 'lightweight formal methods'. While Z does not appear in the discussion, we would argue that it's ideally suited to be used as a 'lightweight' method.

gain a grade in the distinction range (70%+) if they can demonstrate that they can convincingly undertake deductive and inductive proofs.

4 The approach

We now give consideration to the changes in our approach to delivering the Software Engineering Mathematics course.

4.1 The motivation for change

As discussed in Section 3, the course text is *Using Z* by Woodcock and Davies [41]. Prior to the change in emphasis, the course's timetable followed faithfully the first 10 chapters of the book:

1. Introduction (Monday AM)
2. Propositional logic (Monday AM)
3. Predicate logic (Monday AM–Tuesday AM)
4. Equality and definite description (Tuesday PM)
5. Sets (Wednesday AM)
6. Definitions (Wednesday PM)
7. Relations (Thursday AM)
8. Functions (Thursday PM)
9. Sequences (Thursday PM–Friday AM)
10. Free types (Friday AM)

The timetable (and, relatedly, the textbook) gave rise to two main challenges in delivering the content. First, natural deduction is at the forefront of *Using Z*: natural deduction rules for conjunction, disjunction, etc. are presented at the point at which the core logical concepts are introduced. To some students, this presents a barrier to learning as the pace at which they learn the notions of propositional and predicate logic is slowed due to a need to appreciate the intricacies of natural deduction rules (and tactics). An additional consequence is that natural deduction assumes greater importance in the minds of the students than it perhaps deserves.

Second, while the timetable allowed for exercises that reinforced learning of individual concepts, there wasn't the scope to allow students to leverage the techniques taught to actually build models: exercises simply reinforced the concepts taught in the previous hour or two. Subsequently, there was evidence that, when it came to the assignment, some students — having not had the experience of building models during the week — had difficulty making the transition from theory to practice.

4.2 A change in emphasis

The substantial change made was to compress and redistribute material to ensure that all of the material required to utilise the taught techniques in a meaningful

way and build models was taught by the end of Wednesday — leaving Thursday clear for a whole day of case studies. (Friday morning was thereafter dedicated to free types and structural induction).

The other important change (although less important in the context of this paper) was to divorce the introduction to propositional and predicate logic from the introduction to natural deduction. The resulting compressed timetable looked as follows:

1. Introduction and propositional logic (Monday AM): Chapters 1 and 2 (minus natural deduction)
2. Predicate logic, equality and definite description (Monday PM): Chapter 3 (minus natural deduction) and Chapter 4
3. Natural deduction (Tuesday AM): the remainder of Chapters 2 and 3
4. Sets and definitions (Tuesday PM–Wednesday AM): Chapters 5 and 6
5. Relations (Wednesday AM–Wednesday PM): Chapter 7
6. Functions and sequences (Wednesday PM): Chapters 8 and 9 (minus structural induction on sequences)
7. Modelling case studies (Thursday AM and PM)
8. Free types and structural induction (Friday AM): the remainder of Chapter 9 and Chapter 10

4.3 Benefits and challenges

The change gave rise to two significant benefits. First, the new structure has a clear delineation between modelling and proof: proof techniques no longer 'get in the way' when introducing new techniques. Second, the conceptually familiar topic of sets appear significantly earlier in the week: on Tuesday afternoon, rather than on Wednesday morning.

As well as benefits, the change gave rise to some challenges. The most significant challenge was that, in order to create the space to spend a whole day on modelling exercises, the pace of the first three days necessarily had to be swift in order to cover the material. Second, as there was deviation from the 'natural order' of the course text, there had to be a degree of trust from the students that the 'postponed' material would be covered in due course.

4.4 An example

Dedicating a whole day to modelling case studies allows students to apply the techniques that they have been taught. The students tackle the exercises in groups of three or four, using whiteboards. If time allows, the students utilise LaTeX and the Fuzz type-checker.

An example case study is reproduced below.

> A TV recording system records television programmes to a hard-drive. The hard-drive has the capacity to store up to 200 hours of programming; each programme may be at most 6 hours in length.

When the viewer accesses the hard-drive, they are presented with a menu presenting all of the shows currently stored. The details are:

- title;
- programme length; and
- whether or not the programme has been viewed.

You may assume the following types and abbreviations.

$$[Title]$$
$$Length == \mathbb{N}$$
$$Viewed ::= yes \mid no$$

(We shall assume that the length of recordings is represented in terms of **minutes**.)

(a) Complete the following axiomatic definition with appropriate constraint information ("the hard-drive has the capacity to store up to 200 hours of programming; each programme may be at most 6 hours in length"):

$$hd : \operatorname{seq}(Title \times Length \times Viewed)$$
$$\vdots$$

The sequence hd captures information pertaining to programmes stored on the hard-drive. (You should assume, for now, the existence of a function $cumulative\_total \in \operatorname{seq}(Title \times Length \times Viewed) \to Length$. This function will be defined in part (d).)

(b) Define, via set comprehension, the collection of titles of programmes (which appear in hd) that are over two hours in length.

(c) Define functions *viewed* and *not_viewed* that take sequences of type $Title \times Length \times Viewed$, and return the sequences with the not viewed and viewed programmes removed respectively. So,

$$viewed\ \langle (t_1, 3, yes), (t_2, 4, yes), (t_1, 5, no) \rangle = \langle (t_1, 3, yes), (t_2, 4, yes) \rangle$$
$$not\_viewed\ \langle (t_1, 3, yes), (t_2, 4, yes), (t_1, 5, no) \rangle = \langle (t_1, 5, no) \rangle$$

(d) Define a **recursive** function, *cumulative_total*, that takes sequences of type $Title \times Length \times Viewed$ and returns the cumulative length of the programmes recorded. So,

$$cumulative\_total\ \langle (t_1, 3, yes), (t_2, 4, yes), (t_1, 5, no) \rangle = 12$$

(e) Give a μ-expression for the title of the longest programme that appears in hd.

(f) Define a function that maps programme titles (which appear in hd) to cumulative lengths, i.e.,

$$f\ \langle (t_1, 3, yes), (t_2, 4, yes), (t_1, 5, no) \rangle = \{t_1 \mapsto 8, t_2 \mapsto 4\}$$

(g) Define a function that takes a sequence of programmes and removes the longest viewed one, i.e.,

$$g\,\langle (t_1, 3, yes), (t_2, 4, yes), (t_1, 5, no)\rangle = \langle (t_1, 3, yes), (t_1, 5, no)\rangle$$

(h) Define a function that takes an element of $\mathrm{seq}\,(\mathit{Title} \times \mathit{Length} \times \mathit{Viewed})$ and sorts that sequence in terms of programme length—with the longest programme appearing first. So,

$$s\,\langle (t_1, 3, yes), (t_2, 4, yes), (t_1, 5, no)\rangle = \langle (t_1, 5, no), (t_2, 4, yes), (t_1, 3, yes)\rangle$$

Typically, such an exercise — which would take take an expert no more than 20 minutes or so to complete — will take groups of three or four between two and three hours.

5 Indicative results

We now consider some indicative results regarding the success of the initiative. We recognise that there are caveats here: the class sizes are small; the groups are heterogeneous in their make up; there is an element of subjectivity in any assessment process. However, we are able to leverage data spanning several years.

There have been 10 instances of the course using the approach described in this paper; to compare, we also consider the final 10 instances of the course using the former approach. We consider first student performance (in terms of examination results) and then consider student feedback.

5.1 Student performance

As already discussed, students are assessed by way of a take-home assignment that they have six weeks to complete.

The assessment criteria for the course are given below.

1. Propositional and predicate logic: have you understood the syntax and semantics of propositional and predicate logic? can you write logical statements? can you interpret logical statements? can you reason about logical statements?
2. Equality and definitions: do you understand the notion of a type? do you understand the different ways of introducing types, sets, and identifiers into a formal document? do you understand the notion of equality and its associated properties?
3. Sets, relations, functions, and sequences: do you understand the formal representations of these structures? can you define such structures according to some property? can you apply the operators associated with these structures? can you interpret a statement defined in terms of these operators? can you reason about such statements? can you use these structures to describe systems and properties?

Table 1. Student performance

Date	Submissions	Min.	Max.	Mean	Median	% 50+	% 70+
2010 (iteration 1)	16	10	85	59	62.5	81.25	37.5
2010 (iteration 2)	13	35	95	61	55	84.6	23.1
2011 (iteration 1)	17	20	85	56	55	64.7	29.4
2011 (iteration 2)	12	35	80	58	57.5	83.3	16.7
2012 (iteration 1)	15	30	85	64	60	86.7	40
2012 (iteration 2)	10	20	90	61	60	90.0	30.0
2012 (iteration 3)	10	30	80	55	56.5	60.0	30.0
2013 (iteration 1)	13	20	90	63	65	84.6	38.5
2013 (iteration 2)	11	50	80	62	58	100.0	36.4
2014 (iteration 1)	13	33	94	65	60	92.3	15.4
2014 (iteration 2)	14	40	95	67	67	78.6	42.9
2015 (iteration 1)	3	55	63	60	63	100.0	0.0
2015 (iteration 2)	19	10	88	61	64	89.5	31.6
2015 (iteration 3)	16	42	80	63	66.5	81.3	43.8
2016 (iteration 1)	5	35	74	55	62	60.0	20.0
2016 (iteration 2)	15	40	88	67	67	86.7	46.7
2017 (iteration 1)	12	35	83	63	65	83.3	41.7
2017 (iteration 2)	16	35	73	58	61	81.3	12.5
2018 (iteration 1)	14	45	95	72	69	92.9	50.0
2018 (iteration 2)	14	55	90	72	71.5	100.0	50.0
Pre-change	**130**	**10**	**95**	**60**	**60**	**82.3**	**35.4**
Post-change	**128**	**10**	**95**	**64**	**65**	**85.9**	**37.5**

4. Free types: can you define a free type? have you understood the principle of recursion? can you construct an inductive proof?

Assignments in this subject typically consist of 10 questions and follow a similar structure each time: Question 1 is typically concerned with truth tables; Question 2 is typically concerned with equivalence proofs; Question 3 typically pertains to proof trees; Questions 4–8 leverage a scenario, asking the students to write definitions and constraints, and then define sets, relations and functions. Such questions are on a par (in terms of style and difficulty) with the case studies discussed in Section 4. Questions 9 and 10 typically involve free type definitions and proof by induction. The structure of the assignments was consistent across the course instances considered in this paper.

Recall from Section 3 that students pass an assignment in this subject (scoring 50%+) if they can use the mathematical language of Z to build simple models; they gain a grade in the distinction range (70%) if they can demonstrate that they can convincingly undertake deductive and inductive proofs.

Table 1 illustrates examination results for 20 iterations of the course: the first 10 were delivered 'pre-change'; the last 10 were delivered 'post-change'. The number of submissions, lowest score, highest score, mean score and median score are given for each instance. The percentage of submissions scoring 50+ and 70+ respectively are also given.

Table 2. Student feedback

	All courses	Pre-change	Post-change	**Difference**
Statement 1	4.6	4.73	4.77	*0.85%*
Statement 2	4.66	4.68	4.82	*2.99%*
Statement 3	4.41	4.38	4.59	*4.79%*
Statement 4	4.77	4.83	4.94	*2.28%*
Statement 5	4.46	4.64	4.7	*1.29%*
Statement 6	4.77	4.89	4.91	*0.41%*
Statement 7	4.75	4.84	4.85	*0.21%*
Statement 8	4.55	4.58	4.75	*3.71%*
Statement 9	4.43	4.13	4.18	*1.21%*
Statement 10	4.42	4.45	4.64	*4.27%*
Statement 11	4.44	4.38	4.61	*5.25%*
Statement 12	4.58	4.56	4.7	*3.07%*
Overall	**4.57**	**4.59**	**4.71**	**2.61%**

The bottom rows aggregate the respective scores. Curiously, the lowest and highest grades do not differ at all. However, the mean and median scores have improved significantly; there are slight increases in the percentages scoring 50+ and 70+.

There is one final measure that can be utilised: non-submission of assignments by students who have attended the course. This rate has decreased slightly: from 21.2% (pre-change) to 20.0% (post-change).

There is, beyond the raw facts, little that we can conclude here. However, the overall increase in grades is clearly a pleasing result and perhaps indicates that, even if practice does not 'make perfect', it does 'make better'.

5.2 Feedback

Following each course, students are invited to complete (anonymously) a questionnaire. The statements (scored in the range 1–5) are as follows.

1. The lectures added significant value to the course material
2. The lecturer took the time needed to explain the key concepts
3. The lectures included valuable contributions from the other students in the class
4. The lecturer was helpful and ready to answer questions
5. The exercises helped me to understand the topics covered in the lectures
6. The lecturer or tutor was knowledgeable and encouraging
7. Help was available — from the lecturer or tutor — when I needed it
8. Issues raised were adequately addressed — through model solutions or discussion
9. I think that the techniques taught during the course will be valuable to me in the future
10. The course was well constructed: the various components worked well together

11. The course material was appropriate, and of good quality
12. The course administration was efficient and effective

While the final question is not of direct relevance to this paper, we include it here for the sake of completeness.

In Table 2, we compare the scores per-question for pre-change and post-change iterations. We also compare the scores with the overall scores across all courses between mid-February 2010 (when data was first collected in this fashion) and mid-February 2019 — giving rise to a total of 6264 completed questionnaires.

When comparing pre-change and post-change courses, there is a positive difference in feedback in all questions. The most significant differences can be seen for Statements 3 ("The lectures included valuable contributions from the other students in the class"), 8 ("Issues raised were adequately addressed — through model solutions or discussion"), 10 ("The course was well constructed: the various components worked well together") and 11 ("The course material was appropriate, and of good quality").

Post-change, the course outperforms the average feedback with respect to all statements, with one exception. (Pre-change, it was below the average on three others: "The lectures included valuable contributions from the other students in the class", "The course material was appropriate, and of good quality" and "The course administration was efficient and effective".) And it is this question — "I think that the techniques taught during the course will be valuable to me in the future" — which, after all, motivated the changes (and, indeed, this contribution). While the slight increase is pleasing, the feedback does, perhaps, show that there is still some way to go in terms of demonstrating relevance to practitioners.

6 Conclusions

In this paper we have described how we have changed the emphasis of a course that introduces part-time students typically employed in the software engineering industry to introductory topics from discrete mathematics and formal methods. While our arguments for such an emphasis are not new (see, for example, [2], in which Barr advocates helping the situation by requiring students to model real-world implementations), we have been able to demonstrate how, via close to a decade's worth of data, the changes have given rise to some pleasing results.

We recognise that our experiences are somewhat unique: the nature of the Software Engineering Programme (being targeted at professional software engineers) is very different to an undergraduate programme in Computer Science; the make-up of the class is more heterogeneous; the nature of the teaching (in intensive one-week blocks) is different from the typical mode of delivery; and the nature of assessment differs from what most full-time students will be used to. However, we do think that some of the challenges faced will be familiar to many, and, indeed, are part of the ongoing discourse with respect to the value and applicability of modelling techniques.

One clear trend over the 25+ years of the Programme's existence is the shift from companies funding their employees' professional development to few employers now being prepared to provide such support. As it is now typical for students to 'pay their own way', there is an increasing need to provide evidence of practical value. To this end, and reflecting upon the results of Section 5, future changes will be driven by the statement "I think that the techniques taught during the course will be valuable to me in the future".

Acknowledgements

The author would like to thank the anonymous reviewers for their helpful and constructive comments.

References

1. Almi, N.E.A.M., Rahman, N.A., Purusothaman, D., Sulaiman, S.: Software engineering education: The gap between industry's requirements and graduates' readiness. In: Proceedings of the IEEE Symposium on Computers and Informatics (ISCI 2011). pp. 542–547 (2011)
2. Barr, T.: Improving software engineering education by modeling real-world implementations. In: Proceedings of the 8th edition of the Educators' Symposium (EduSymp 2012). pp. 36–39. ACM (2012)
3. Cowling, A.J.: The role of modelling in teaching formal methods for software engineering. In: Bollin, A., Margaria, T., Perseil, I. (eds.) Proceedings of the 1st Workshop on Formal Methods in Software Engineering Education and Training (FMSEE&T 2015). CEUR Workshop Proceedings, vol. 1385 (2015)
4. Cristiá, M.: Why, how and what should be taught about formal methods? In: Bollin, A., Margaria, T., Perseil, I. (eds.) Proceedings of the 1st Workshop on Formal Methods in Software Engineering Education and Training (FMSEE&T 2015). CEUR Workshop Proceedings, vol. 1385 (2015)
5. Davies, J.W.M., Gibbons, J., Welch, J., Crichton, E.: Model-driven engineering of information systems: 10 years and 1000 versions. Science of Computer Programming **89**, 88–104 (2014)
6. Davies, J.W.M., Simpson, A.C., Martin, A.P.: Teaching formal methods in context. In: Dean, C.N., Boute, R.F. (eds.) Proceedings of CoLogNet / Formal Methods Europe Symposium on Teaching Formal Methods 2004. Lecture Notes in Computer Science, vol. 3294, pp. 186–202. Springer (2004)
7. Devlin, K.: Why universities require computer science students to take math. Communications of the ACM **46**(9), 37–39 (2003)
8. Dijkstra, E.W.: The humble programmer. Communications of the ACM **15**(10), 859–866 (1972)
9. Fincher, S., Utting, I.: Pedagogical patterns: their place in the genre. In: Caspersen, M.E., Joyce, D.T., Goelman, D., Utting, I. (eds.) Proceedings of the 7th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, (ITiCSE 2002). pp. 199–202. ACM (June 2002)
10. Finkelstein, A.: Software engineering education: A place in the sun? In: Proceedings of the 16th International Conference on Software Engineering (ICSE 1994). pp. 358–359. IEEE Computer Society Press (1994)

11. Fraser, S., Bareiss, R., Boehm, B., Hayes, M., Hill, L., Silberman, G., Thomas, D.: Meeting the challenge of software engineering education for working professionals in the 21st century. In: Proceedings of the 18th Annual SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003). pp. 262–264 (2003)
12. Hayes, I.J.: Specification Case Studies. Prentice-Hall, second edn. (1992)
13. Honiden, S., Tahara, Y., Yoshioka, N., Taguchi, K., Washizaki, H.: Top SE: Educating superarchitects who can apply software engineering tools to practical development in Japan. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007). pp. 708–718. IEEE Computer Society Press (2007)
14. Ishikawa, F., Taguchi, K., Nobukazu, Y., Honiden, S.: What top-level software engineers tackle after learning formal methods: Experiences from the Top SE project. In: Gibbons, J., Oliveira, J.N. (eds.) Proceedings of the 2nd International Conference on Teaching Formal Methods (TFM 2009). Lecture Notes in Computer Science, vol. 5846, pp. 57–71. Springer (2009)
15. Jackson, D.: Lightweight formal methods. In: Oliveira, J.N., Zave, P. (eds.) Proceedings of the 2001 International Symposium of Formal Methods Europe (FME 2001). Lecture Notes in Computer Science, vol. 2021, p. 1. Springer (2001)
16. Jackson, M.: Aspects of abstraction in software development. Software and Systems Modeling **11**(4), 495–511 (2012)
17. Jacky, J.: The Way of Z: Practical Programming With Formal Methods. Cambridge University Press (1997)
18. Jaume, M., Laurent, T.: Teaching formal methods and discrete mathematics. In: Dubois, C., Giannakopoulou, D., Méry (eds.) Proceedings of the 1st Workshop on Formal Integrated Development Environment (F-IDE 2014). pp. 30–43 (2014)
19. Kiniry, J.R., Zimmerman, D.M.: Secret ninja formal methods. In: Cuellar, J., Maibaum, T.S.E., Sere, K. (eds.) Proceedings of the 15th International Symposium on Formal Methods (FM 2008). Lecture Notes in Computer Science, vol. 5104, pp. 214–228. Springer (2008)
20. Kramer, J.: Is abstraction the key to computing? Communications of the ACM **50**(4), 36–42 (2007)
21. Kramer, J.: Abstraction and modelling — a complementary partnership. In: Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M. (eds.) Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008). Lecture Notes in Computer Science, vol. 5301. Springer (2008)
22. Kramer, J., Hazzan, O.: The role of abstraction in software engineering. ACM SIGSOFT Software Engineering Notes **31**(6), 38–39 (2006)
23. Larsen, P.G., Fitzgerald, J., Riddle, S.: Learning by doing: Practical courses in lightweight formal methods using VDM+. Tech. Rep. CS-TR-992, University of Newcastle upon Tyne (2006)
24. Mandrioli, D.: On the heroism of really pursuing formal methods: title inspired by Dijkstra's "On the Cruelty of Really Teaching Computing Science". In: Proceedings of the Third FME Workshop on Formal Methods in Software Engineering (Formalise 2015). pp. 1–5. IEEE Computer Society Press (2015)
25. Martin, A.P., Simpson, A.C.: Generalizing the Z schema calculus: Database schemas and beyond. In: Proceedings of APSEC, 2003. pp. 28–37 (2003)
26. Mead, N.R., Ellis, H.J.C., Moreno, A., MacNeil, P.: Can industry and academia collaborate to meet the need for software engineers? Cutter IT Journal **14**(6), 32–39 (2001)
27. Morgan, C.C., Sufrin, B.A.: Specification of the UNIX filing system. IEEE Transactions on Software Engineering **10**(2), 128–142 (1984)

28. Nishihara, H., Shinozaki, K., Hayamizu, K., Aoki, T., Taguchi, K., Kumeno, F.: Model checking education for software engineers in Japan. SIGCSE Bulletin **41**(2), 45–50 (2009)
29. Reed, J.N., Sinclair, J.E.: Motivating study of formal methods in the classroom. In: Dean, C.N., Boute, R.T. (eds.) Proceedings of Teaching Formal Methods 2004 (TFM 2004), Lecture Notes in Computer Science, vol. 3294, pp. 32–46. Springer (2004)
30. Robinson, K.: Embedding formal development in software engineering. In: Dean, C.N., Boute, R.F. (eds.) Proceedings of CoLogNet / Formal Methods Europe Symposium on Teaching Formal Methods 2004. Lecture Notes in Computer Science, vol. 3294, pp. 203–213. Springer (2004)
31. Roscoe, A.W.: Understanding Concurrent Systems. Springer (2010)
32. Simpson, A.C.: Discrete Mathematics by Example. McGraw-Hill (2002)
33. Simpson, A.C., Martin, A.P., Cremers, C., Flechais, I., Martinovic, I., Rasmussen, K.: Experiences in developing and delivering a programme of part-time education in software and systems security. In: Proceedings of the 37th International Conference on Software Engineering (ICSE 2015) — Volume 2. pp. 435–444. IEEE Computer Society Press (2015)
34. Simpson, A.C., Martin, A.P., Gibbons, J., Davies, J.W.M., McKeever, S.W.: On the supervision and assessment of part-time postgraduate software engineering projects. In: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003). pp. 628–633. IEEE Computer Society Press (2003)
35. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall International, second edn. (1992)
36. Subrahmanyam, G.V.B.: A dynamic framework for software engineering education curriculum to reduce the gap between the software organizations and software educational institutions. In: Proceedings of the 22nd IEEE International Conference on Software Engineering Education and Training (CSEET 2009). pp. 248–254 (2009)
37. Tarkan, S., Sazawl, V.: Chief chefs of Z to Alloy: Using a kitchen example to teach Alloy with Z. In: Gibbons, J., Oliveira, J.N. (eds.) Proceedings of the 2nd International Conference on Teaching Formal Methods (TFM 2009). Lecture Notes in Computer Science, vol. 5846, pp. 72–91. Springer (2009)
38. Vaughn, R.B., Carver, J.: Position paper: The importance of experience with industry in software engineering education. In: Proceedings of the 19th IEEE International Conference on Software Engineering Education and Training (CSEET 2006). pp. 19–19 (2006)
39. Warford, J.S.: An experience teaching formal methods in discrete mathematics. ACM SIGCSE **27**(3), 60–64 (1995)
40. Wing, J.M.: Computational thinking. Communications of the ACM **49**(3), 33–35 (2006)
41. Woodcock, J.C.P., Davies, J.W.M.: Using Z: Specification, Refinement, and Proof. Prentice Hall (1996)
42. Woodcock, J.C.P., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM Computing Surveys **41**(4), Article number 19 (2009)
43. Zamansky, A., Spichkova, M., Rodriguez-Navas, G., Herrmann, P., Blech, J.O.: Towards classification of lightweight formal methods. In: Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2013) (2018)