Weaving a Formal Methods Education with Problem-Based Learning

J Paul Gibson

Le Département Logiciels-Réseaux, IT-SudParis,
9 rue Charles Fourier, 91011 Évry cedex, France
paul.gibson@it-sudparis.eu
http://www-public.it-sudparis.eu/~gibson/

Abstract. The idea of weaving formal methods through computing (or software engineering) degrees is not a new one. However, there has been little success in developing and implementing such a curriculum. Formal methods continue to be taught as stand-alone modules and students, in general, fail to see how fundamental these methods are to the engineering of software. A major problem is one of motivation — how can the students be expected to enthusiastically embrace a challenging subject when the learning benefits, beyond passing an exam and achieving curriculum credits, are not clear? Problem-based learning has gradually moved from being an innovative pedagogique technique, commonly used to better-motivate students, to being widely adopted in the teaching of many different disciplines, including computer science and software engineering. Our experience shows that a good problem can be re-used throughout a student's academic life. In fact, the best computing problems can be used with children (young and old), undergraduates and postgraduates. In this paper we present a process for weaving formal methods through a University curriculum that is founded on the application of problem-based learning and a library of good software engineering problems, where students learn about formal methods without sitting a traditional formal methods module. The process of constructing good problems and integrating them into the curriculum is shown to be analagous to the process of engineering software. This approach is not intended to replace more traditional formal methods modules: it will better prepare students for such specialised modules and ensure that all students have an understanding and appreciation for formal methods even if they do not go on to specialise in them.

Keywords: Teaching Formal Methods, Computing Curriculum, Mathematics of Computer Science, Science of Software Engineering.

1 Introduction

In this paper we consider the problem of teaching formal methods at 3rd level institutions (universities, colleges, institutes of technology, etc.). We support the view that the best way to teach formal methods is not to teach the subject as a stand-alone module or set of modules, but to try to integrate (weave) the formality throughout the whole curriculum. We propose a problem-based learning approach[1] (PBL)as the best way in which to weave the formal methods.

In the remainder of this introduction we review the notion of weaving formal methods, summarise the learning theory that lies behind problem-based learning (PBL), review PBL and introduce the core issue of finding good formal methods problems.

After the introduction, section 2 proposes some formal methods learning objectives, section 3 proposes a process — much like the software engineering process — by which formal methods can be woven through a curriculum, section 4 reviews some of the problems that we have found to be successful, and section 5 concludes with some brief observations.

We note that we try — where possible — to bring our own research back into all our teaching (undergraduate and postgraduate). Thus, much of our source material originates from our own research publications, which we cite. Of course, when teaching our students we ensure that they are also made aware of the work on which our own research is based. In the space of this paper we are unable to reference all this secondary material, although we do reference the research that is specific to teaching formal methods, where appropriate.

1.1 Weaving a Formal Methods Thread through a Curriculum: The Integration Problem

In 2000, Jeanette Wing wrote about weaving formal methods[2]:

> “Rather than treat formal methods solely as a separate subject to study, we should weave their use into the existing infrastructure of an undergraduate computer science curriculum. In so doing, we would be teaching formal methods alongside other mathematical, scientific, and engineering methods already taught. Formal methods would simply be additional weapons in a computer scientist’s arsenal of ways to think when attacking and solving problems.
> My ideal is to get to the point where computer scientists use formal methods without even thinking about it. Just as we use simple mathematics in our daily life, computer scientists would use formal methods routinely.”

She then goes on to identify the common core elements that need to be taught: state machines, invariants, abstract mappings, composition, specification, induction and verification. She states that tools are critical: model checkers, specification checkers and theorem provers. Specific courses where formal methods can be taught are identified as: introduction to programming, data structures and algorithms, programming principles, programming languages, compilers, software engineering, computer architecture, operating systems, networking, databases, and user interfaces. She concludes her paper by stating:

> “The biggest obstacle is getting “buy-in” from our colleagues: convincing co-instructors, curricula committees, and administrators that integrating

> formal methods unintru- sively is a good thing to do . . . The nitty-gritty hard future work is in thinking of the examples to use in lectures, in designing appropriate homework and exam problems, and in making learning these concepts and tools enjoyable."

Eight years have passed since this paper was published and not much has changed. Formal methods continue to be taught in a stand-alone fashion and little progress has been made in fully integrating them into the computer science and software engineering (CS&SE) curricula.

More recently, Kiniry and Zimmerman discuss the use of "secret ninja" techniques[3] "to integrate applied formal methods into software engineering courses." They demonstrate that formal methods can be taught through "stealth" (without calling them formal methods) in a number of different courses; but note that this success would not have been possible without good tool suppport. Their work is founded mostly on applying the design-by-contract paradigm. This demonstrates that formal methods can and should be used in the teaching of software design; and this view is supported by other research[4].

We believe that there are many other practitioners of teaching formal methods by stealth throughout the world. The problem is that each has their own technique for better integrating formal methods into the specific part of the curriculum in which they teach. There is no consistent approach to this integration. In this paper we propose that good problems can be used to weave (integrate) formal methods in a consistent manner. However, before we look at PBL we review the learning theory upon which our claims are based.

1.2 Learning Theory

There are numerous complementary, and competing, theories of learning. The review by Hilgard and Bower published over half a century ago[5] is a good introduction to the foundations of learning theory. In this paper, we review the work of the researchers that have had most influence on our own research into teaching formal methods: Piaget, Bruner, Guildford, Gardner, Papert, Schoenfeld and Bloom.

Cognitive structure is the concept central to Piaget's theory. (See the work by Brainerd[6] for a good overview and analysis of Piaget's seminal contribution.) These structures are used to identify patterns underlying certain acts of intelligence, and Paiget proposes that these correspond to stages of child development. Piaget's most interesting experiments focused on the development of mathematical and logical concepts. However, his work predates the development of software engineering as a discipline.

Piaget's theory is similar to other *constructivist* perspectives of learning (e.g., Bruner [7]), which model learning as an active process where learners construct new concepts upon their current knowledge and previous experience. As a result of following this theory, teachers encourage students to discover principles by themselves: this is the foundation upon which problem-based learning is built.

Similarites can be seen between the constructivist view and the *theories of intelligence* such as proposed by Guildford's *structure of intellect* (SI) theory [8] and Gardner's *multiple intelligences*[9]. Typically, these theories structure the learning space in terms of practical problem solving skills.

Piaget's ideas also influenced the seminal work by Seymour Papert in the specific domain of computers and education[10]. Papert argues that children can understand concepts best when they are able to explain them algorithmicaly through writing computer programs.

We were also influenced by the domain of teaching mathematics. In particular, Alan Schoenfeld argues that understanding and teaching mathematics should be treated as problem-solving [11]. He identifies four skills that are needed to be successful in mathematics: proposition and procedural knowledge, strategies and techniques for problem resolution, decisions about when and what knoweldge and strategies to use, and a logical *world view* that motivates an individual's approach to solving a particular problem.

To conclude our review we mention Blooms taxonomy[12] of educational objectives which is a fundamental model of learning, providing a well-accepted foundation for research and development into the preparation of learning evaluation materials. It structures understanding into 6 distinct levels: Knowledge, Comprehension, Application, Analysis, Synthesis and Evaluation.

1.3 Problem Based Learning

While there is no universal definition of PBL we present definitions from the last three decades. PBL was defined by Barrows and Tamblyn[13] as "the learning which results from the process of working towards the understanding of, or resolution of, a problem. The problem is encountered first in the learning process". Woods defined it[1] as "an approach to learning that uses a problem to drive the learning rather than a lecture with subject matter which is taught." Torp and Sage define it[14] as "Focused, experiential learning (minds-on, hands-on) organised around the investigation and resolution of messy, real-world problems."

Thus, the guiding principle behind PBL is that the problem is the driving force behind the learning. Within the PBL environment the problem acts as the catalyst that initiates the learning process. It is said that this way of learning encourages a deeper understanding of the material, rather than surface learning, because it is the students who are actively *doing*. As the problem is such a critical component of the learning process it is imperative that one uses *good* problems. In 2001, Duch identified five characteristics of what makes a PBL problem *good*[15]:

1. Effective problems should engage the students' interest and motivate them to probe for deeper understanding.
2. PBL problems should be designed with multiple stages.
3. The problems should be complex enough that cooperation within a group will be necessary in order for them to effectively work towards a solution.
4. The problem should be open-ended.
5. The content objectives of the course should be incorporated into the problems.

One of the major obstacles to the implementation of PBL, within any discipline, is the lack of a good set of problems. However, good PBL problems usually do not appear in textbooks[16]. Clearing houses provide an avenue to allow for the sharing of problems, but unfortunately there is a lack of CS&SE problems[1].

1.4 Good Formal Methods Problems

It is known that the students initial reactions to a subject or topic is critical to them gaining an interest. The choice of problem is therefore critical. A good formal methods problem is one in which students will have the computer science knowledge necessary to solve the problem but need software engineering knowledge to learn how to apply the science, or students will identify (through software engineering knowledge) a possible solution to the problem whose suitability depends on some core computer science that they do not currently have. The problem should lead both types of student to appreciate the need for formality and rigour. Consequently, they build their own formal methods bridges that link the science with the engineering (but just happen to start the construction on opposite sides of the academic divide).

The high-level objective of helping bridge the gap between the science and engineering is laudable; however, it is much too abstract. We must refine this high-level objective into more concrete objectives against which our problems can be verified.

2 Formal Methods: Learning Objectives

Through our formal methods problems we can verify our high level objective of helping students to bridge the gap between computer science and software engineering by checking that the students who work on the problems are then:

- able to build better software,
- better at reasoning about problems that are to be solved using a computer,
- motivated and able to work with abstract models and conceptual tools,
- able to classify, and motivated to use, software development tools,
- knowledgable about the scientific foundations upon which the tools are built, and
- comfortable working with mathematics such as logic and set theory.

2.1 Improve Software (Development)

Software engineering is all about going from *what* to *how*, moving from abstract problems to concrete solutions. This involves design steps: decisions that are made in order to move a model away from an open (usually non-deterministic)

[1] There is an abundance of CS programming problems available; however, the vast majority of these problems place emphasis on the learning of a particular programming concept rather than problem solving.

description of requirements to a closed (usually deterministic) description of the implementation. Software engineering cannot, in practice, be done prescriptively (otherwise we would automatically generate solutions from problems); and it cannot be done in a purely ad-hoc fashion (otherwise we would not need software processes to manage the complexity of the systems and behaviour being modelled). Software engineering is a unique mix of science, engineering and art: the best practitioners know that each new problem requires a different balance between the potential chaos of innovation and the constraints imposed by order and structure.

Thus, engineering is not about finding the correct solution to a problem; it is about understanding the engineering compromises involved in choosing a solution from a large number of possibilities. In such a situation can we judge how well a software engineer is working by judging the quality of a single project on which they have worked? In practice, it is very difficult, if not impossible, to fairly evaluate whether the objective of improving the students' software development skills is being met by our PBL approach to teaching formal methods. O'Kelly and Gibson have discussed the issues that arise when trying to validate PBL in the context of teaching programming[17], and many of the issues that they identify are relevant when analysing whether the formal methods problems are teaching the students how to be better software engineers (i.e. engineer better software).

Anecdotal evidence suggests that the better students adopt formal engineering practices (like the specification of invariants) in projects on other courses which follow their work on the formal methods problems (without being told to do so). Furthermore, the software that these students produce is better than that produced by the other students. However, that should be no surprise as these are the better students!

2.2 Thinking about (Computational/Algorithmic) Thinking

In 2006 Wing discusses the importance of computational thinking in education[18]:

> "Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science. Computational thinking includes a range of mental tools that reflect the breadth of the field of computer science."

She then goes on to discuss the characteristics of such thinking:

1. Conceptualizing, not programming;
2. Fundamental, not rote skill;
3. A way that humans, not computers, think;
4. Complements and combines mathematical and engineering thinking;
5. Ideas, not artifacts;
6. For everyone, everywhere.

Such computational thinking starts from a very early age[19] and should be exploited in the teaching of computer science (and formal methods) in schools[20].

Our experience shows that looking at simple formal methods problems with school children improves their ability to think computationally. Thus, we believe that this should also be true for university students.

2.3 Make Friends with Abstraction and Modelling: Conceptual Tools

One of the biggest problems in teaching software engineering is that students find it very difficult to work with models at different levels of abstraction. Through the formal methods problems, the students naturally discover that abstraction is a critical skill when searching the space of possible solutions. They then proceed to discover refinement — where they gradually add details to their abstract models in order to move them closer to a concrete solution. Students learn that nondeterminism is a very powerful mechanism. We first noticed this when we analysed how best to teach formal specification as part of requirements engineering[21].

2.4 Make Friends with Software for Software Engineering: Development Tools

The need for students to be able to use general software development tools is widely accepted by industry; but the importance of them being able to use formal methods tools is not. One used to be able to argue that formal methods were not used in industry because they were not mature enough — and therefore it would be difficult to motivate students to learn how to use them[22] — but this is no longer the case.

2.5 Understand the Scientific Foundations

In 2003, Curran discussed the balancing required between Computer Science (CS) and Software Education (SE) education[23]:

> "It is no longer clear whether SE topics reflect current industry needs or whether they are intended to lead and update industry practices. But regardless of who leads whom, without some sort of rapid, two-way communication, we run the risk of producing graduates who are out of touch, require much re-training, and have trouble competing. Industry might indicate that they need specific skills and knowledge from their CS employees, and that the special skills required of software engineers would be performed by software engineers, not by CS majors."

He concluded by stating:

> "... individual departmental goals for a degree in CS and the role of SE in the curriculum should be clearly understood so that a balance can be struck between academic topics and skills training."

We believe that PBL offers a natural solution to achieving this required balance and in using formal methods to bridge the gap between CS and SE.

More recently, Parnas and Soltys address the need for a "Basic Science for Software Developers"[24], stating:

> "The fundamental properties of computers are very important because they affect what we can and cannot do. Sometimes, an understanding of these properties is necessary to find the best solution to a problem. In most cases, those who understand computing fundamentals can anticipate problems and adjust their goals so that they can get the real job done. Those who do not understand these limitations, may waste their time attempting something impossible or, even worse, produce a product with poorly understood or not clearly stated capabilities. Further, those that understand the fundamental limitations are better equipped to clearly state the capabilities and limitations of a product that they produce. Finally, an understanding of these limitations, and the way that they are proved, often reveals practical solutions to practical problems. Consequently, "basic science" should be a required component of any accredited Software Engineering program."

They then suggest the curriculum for a theoretical computer science that would cover the required science of software engineering. The main topics proposed are:

1. Finite Automata (finite number of states, and no memory),
2. Regular Expressions,
3. Context-Free Grammars,
4. Pushdown Automata (like finite automata, except they have a stack, with no limit on how much can be stored in the stack),
5. Turing Machines (simplified model of a general computer, but equivalent to general computers)
6. Rudimentary Complexity.

We do not directly address the teaching of any of these theoretical computer science foundational topics in our problem based learning. Most of our problems can be (and are) extended to introduce the concepts of complexity, computability, correctness and common-sense — which we see as the fundamental computer science *boundaries* that all software engineers should know about.

2.6 Be Comfortable with Mathematics

Habrias has written about the problems of teaching formal methods when the students do not have a good understanding of foundational mathematics such as logic and set theory[25]. Much of the literature on teaching formal methods directly addresses the need for firm mathematical foundations. We believe that the problem-based learning approach helps students with the mathematics because they learn the mathematical concepts in the context of their practical application.

3 A Software Engineering Approach to Constructing a Formal Methods Curriculum

Parnas makes a strong case that "Software Engineering Programmes are not Computer Science Programmes"[26]. He discusses the differences between traditional computer science programmes and most engineering programmes and argues that we need software engineering programmes that follow the traditional engineering approach to professional education.

He summarises the issue as follows:

> "Just as the scientific basis of electrical engineering is primarily physics, the scientific basis of software engineering is primarily computer science. Attempts to distinguish two separate bodies of knowledge will lead to confusion. ...Recognising that the two programmes would share much of their core material will help us to understand the real differences."

Future scientists will add to our "knowledge base" while future engineers will design trustworthy products. His position is that: "engineers learn science plus the methods needed to apply science".

However, we must now ask where formal methods fit into this pedagogic structure and whether our approach to teaching formal methods should change depending on our target audience: computer scientists or software engineers.

In our approach we see formal methods as the main bridge between computer science and software engineering. Without formal methods software engineering is not a true engineering discipline; and without formal methods computer science remains a mainly theoretical subject. Thus, teaching formal methods should not be seen as a problem to be solved; but it should be viewed as the answer to the fundamental question of how we can better educate computer scientists *and* software engineers.

Our problem based learning approach helps us to better adapt our teaching to our target audience. Our experience suggests that good problems are not good for only one type of students (engineering or science, or even arts and humanities). The best problems can be introduced to any of these students and through interacting with the problem (and with the guidance of the lecturer) the problem will dynamically evolve in order for particular learning objectives to be met. In general, engineering students will learn by trying to build solutions to the problems whilst science students will learn through trying to analyse them. Of course, the lecturer will be responsible for making sure that the students learn that these are complementary approaches and for finding the right balance for the particular type of student that is being taught.

We propose that each problem should be set up to meet a specific curriculum objective. Each problem would then have a life-cycle similar to that seen for software and services, with key stages being specification, design, implementation, testing and maintenance. Once a problem is meeting a specific objective then it can be refined to incorporate other objectives. These objectives may be the responsibility of a single lecturer as part of a single module; but good problems will evolve to survive across different modules. In our experience this is most likely to happen when a single lecturer is responsible for multiple modules (where problems can be shared). However, in order to better weave our formal methods objectives through the curriculum we have to be able to also work with colleagues who do not teach formal methods but do teach other CS&SE modules.

We propose four complementary approaches to this weaving process. Firstly, look at the problems that are being used in other modules and incorporate them into a dedicated formal methods module. Secondly, offer to extend such problems (to meet the formal methods objectives) as part of the original modules in which the problems were taught. Thirdly, offer to extend your existing formal methods problems so that they incorporate learning objectives of colleagues teaching other modules. Fourthly, invite colleagues to participate in the PBL teaching in your own formal methods module(s).

We note that this integration should probably be done in an incremental fashion as we may end up replicating the feature interaction problem[27] at the level of the requirements (learning objectives). To extend our analogy of a problem as being a service, with additional learning objectives as features, we can consider the curriculum to be a system of collaborating services. As our curriculum evolves we maintain the system by updating our problem set: adding new problems, removing unsuccessful problems and evolving successful problems. As with large, complex, software systems the best way to manage this process is to have a clearly documented set of requirements and procedures in place to map these requirements through to the final implemented system (via the design).

We propose that the underlying architecture of the curriculum should be service-oriented in the sense that the main structure should support the evolution of the underlying objectives and the problems that are used to meet these objectives. In the next section we briefly review some of the problems that we have used to meet particular learning objectives. All these problems have been successfully shared across different modules (and different years) in the curriculum.

4 Weaving Formal Methods with Problem Based Learning

In each of the following subsections we briefly review a problem that we have used to teach formal methods within other parts of the CS&SE curriculum. In all cases, these problems have been used at different institutions and in different countries (Ireland, France and USA). Furthermore, they have all been used to teach students at different stages of their academic lives (school, undergraduate and postgraduate).

4.1 Example 1: Stacks and Queues

Stacks and queues are normally taught as part of an algorithms and data structures module. We have found them very useful as a design problem and have used them to teach formal design techniques[4]. We have also used them in teaching about testing, fault tolerance and dependable distributed systems.

4.2 Example 2: E-Voting

The e-voting case study has already been used in the teaching of formal methods [28] where they describe how they have developed a single teaching tutorial, making use of an electronic voting system (EVS), to complement an existing model checking course.

In our work we have used our published research on using formal methods in the verification of safety critical properties[29,30,31,32] as the starting point for problems presented to CS&SE undergraduates (from 1st year to 4th year), and to postgraduates specialising in software engineering. The problem has been re-used in teaching the following modules: introduction to programming, object oriented programming, data structures and algorithms, HCI, testing, requirements and design, rigorous software process, software process improvement.

4.3 Example 3: Sorting and Searching

Sorting and searching are such fundamental computations that it is difficult to imagine a CS&SE module in which they cannot be used as the foundation for a good problem. We have used sorting and searching problems to teach about refinement and correctness-by-construction[22,20] to school children and postgraduates. We also use them to teach about complexity and parallel programming. Finally, we use them in our teaching of models of computation where we consider non-standard computers such as optical computers.

4.4 Example 4: Games, Puzzles and Intelligence

Some of the games that we have used in our PBL teaching include TicTacToe[33], Connect4, and the 15-puzzle. TicTacToe is particularly good when trying to formalise the rules of the game and prove that a particular artificial player respects the rules. We also use it as a simple example of how easy it is for a model checker to automatically examine all possible states of the game. The 15-puzzle is very good when reasoning formally about optimizations as refinements. All these examples have been used in a software process course and all have been used in a module on object oriented design.

4.5 Example 5: Feature Interactions in Telephones

Everyone (who you could possibly wish to teach formal methods to) has an understanding of what you can do with a telephone; even if very few know how it works. It provides an excellent example of modelling and abstraction that can be used with all students. Even students who have never studied computer science usually end up drawing a state machine in order to explain their understanding. For CS&SE students this is a good point at which to introduce features and feature interactions[34]. With respect to formal methods, we use this problem to illustrate the limitations of model checking and the use of theorem provers. We also use it to let students discover the need for models for reasoning about temporal properties such as fairness. This problem has also been used in the teaching of object oriented programming and in the teaching of HCI.

5 Conclusions

We have had some success in weaving formal methods into our CS&SE curriculum by evolving problems that can be shared between modules and that meet multiple learning objectives. There is still much work to be done in adopting a problem-oriented approach to curriculum development. The hardest task is to convice certain colleagues that this approach works: many of them ask for strong evidence that is currently impossible to provide.

We believe that the future of software engineering as a discipline is dependent on students being exposed to formal methods throughout their academic lives. We propose that PBL is a good teaching approach for achieving this aim.

References

1. Woods, D.R.: Problem-based Learning: how to gain the most from PBL. Waterdown, Ontario (1996)
2. Wing, J.M.: Invited talk: Weaving formal methods into the undergraduate computer science curriculum. In: Rus, T. (ed.) AMAST 2000. LNCS, vol. 1816, pp. 2–9. Springer, Heidelberg (2000)
3. Kiniry, J.R., Zimmerman, D.M.: Secret ninja formal methods. In: Cuellar, J., Maibaum, T.S.E., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 214–228. Springer, Heidelberg (2008)
4. Gibson, J.P., Lallet, E., Raffy, J.L.: How do I know if my design is correct? In: Formal Methods in Computer Science Education (FORMED), pp. 59–69 (March 2008)
5. Hilgard, E.R., Bower, G.H.: Theories of Learning. Prentice Hall, Englewood Cliffs (1956)
6. Brainerd, C.: Piaget's Theory of Intelligence. Prentice Hall, Englewood Cliffs (1978)
7. Bruner, J.S.: Toward a theory of instruction. Belknap Press of Harvard University, Cambridge (1966)
8. Guilford, J.P.: The Nature of Human Intelligence. McGraw-Hill, New York (1967)
9. Gardner, H.: Frames of mind: the theory of multiple intelligence. Basic Books, New York (1983)
10. Papert, S., Sculley, J.: Mindstorms: children, computers, and powerful ideas. Basic Books, New York (1980)
11. Schoenfeld, A.H.: Mathematical Problem Solving. Academic Press, Orlando (1985)
12. Bloom, B.S., Engelhart, M.D., Furst, E.J., Hill, W.H., Krathwohl, D.R.: Taxonomy of educational objectives Handbook 1: cognitive domain. Longman Group Ltd., London (1956)
13. Barrows, H., Tamblyn, R.: Problem-Based Learning: An Approach to Medical Education. Springer Publishing Company, New York (1980)
14. Torp, L., Sage, S.: Problems as Possibilities: Problem-Based Learning for K16 Education. Association for Supervision and Curriculum Development (ASCD), Alexandria (2002)

15. Duch, B.: Writing Problems for Deeper Understanding, pp. 47–53. Stylus Publishing, Sterling (2001)
16. Tien, C., Chu, S., Lin, Y.: Four phases to construct problem-based learning instruction materials. In: PBL In Context Bridging work and Education, pp. 117–133. Tampere University Press (2005)
17. O'Kelly, J., Gibson, J.P.: PBL: Year one analysis — interpretation and validation. In: PBL In Context — Bridging Work and Education (2005)
18. Wing, J.M.: Computational thinking. Commun. ACM 49(3), 33–35 (2006)
19. Gibson, J.P., O'Kelly, J.: Software engineering as a model of understanding for learning and problem solving. In: ICER 2005: Proceedings of the 2005 international workshop on Computing education research, pp. 87–97. ACM, New York (2005)
20. Gibson, J.P.: Formal methods - never too young to start. In: Formal Methods in Computer Science Education (FORMED), pp. 149–159 (March 2008)
21. Gibson, J.P.: Formal requirements engineering: Learning from the students. In: Australian Software Engineering Conference, pp. 171–180. IEEE Computer Society, Los Alamitos (2000)
22. Gibson, J.P., Méry, D.: Teaching formal methods: Lessons to learn. In: Flynn, S., Butterfield, A. (eds.) IWFM. Workshops in Computing, BCS (1998)
23. Curran, W.S.: Teaching software engineering in the computer science curriculum. SIGCSE Bull. 35(4), 72–75 (2003)
24. Parnas, D.L., Soltys, M.: Basic science for software developers. In: Workshop on Formal Methods in the Teaching Lab (FM-Ed 2006), pp. 9–14 (August 2006)
25. Habrias, H.: Teaching specifications, hands on. In: Formal Methods in Computer Science Education (FORMED), pp. 5–15 (March 2008)
26. Parnas, D.L.: Software engineering programmes are not computer science programmes. Ann. Software Eng. 6, 19–37 (1998)
27. Gibson, J.P.: Feature requirements models: Understanding interactions. In: Dini, P., Boutaba, R., Logrippo, L. (eds.) Feature Interactions in Telecommunications Networks IV (FIW 1997), pp. 46–60. IOS Press, Amsterdam (1997)
28. Miller, A., Cutts, Q.: The use of an electronic voting system in a formal methods course. In: Workshop on Formal Methods in the Teaching Lab (FM-Ed 2006), pp. 3–8 (August 2006)
29. Gibson, J.P., McGaley, M.: Verification and maintenance of e-voting systems and standards. In: 8th European Conference on e-Government, pp. 283–290 (July 2008)
30. Cansell, D., Gibson, J.P., Méry, D.: Refinement: A constructive approach to formal software design for a secure e-voting interface. Electr. Notes Theor. Comput. Sci. 183, 39–55 (2007)
31. Cansell, D., Gibson, J.P., Méry, D.: Formal verification of tamper-evident storage for e-voting. In: Software Engineering and Formal Methods (SEFM 2007), pp. 329–338. IEEE Computer Society, Los Alamitos (2007)
32. Gibson, J.P.: E-voting and the need for rigorous software engineering — the past, present and future. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, p. 1. Springer, Heidelberg (2006)
33. Gibson, J.P.: A noughts and crosses java applet to teach programming to primary school children. In: PPPJ 2003: Proceedings of the 2nd international conference on Principles and practice of programming in Java, pp. 85–88. Computer Science Press, Inc., New York (2003)
34. Gibson, D.J.P.: Méry,: Formal modelling of services for getting a better understanding of the feature interaction problem. In: Bjorner, D., Broy, M., Zamulin, A.V. (eds.) PSI 1999. LNCS, vol. 1755, pp. 155–179. Springer, Heidelberg (2000)