Formal Methods in the Teaching Lab

Examples, Cases, Assignments and Projects Enhancing Formal Methods Education

A Workshop at the *Formal Methods 2006* Symposium

Organized by the *Formal Methods Europe Subgroup on Education*
August 26, 2006
McMaster University, Hamilton, Ontario, Canada

Workshop preprints

Edited by
Raymond T. Boute and José N. Oliveira

II

Preface

Formal methods are reputed to be difficult, and their acceptance in industry proceeds slowly. Some might wonder whether this is a serious problem. After all, information technology seems to be doing fine regardless of this situation.

Actually, the rapid growth in this area creates a huge quantity of design and implementation tasks that can be done and, more importantly, are being done with little or no scientific, professional or educational background. This makes it difficult to convincingly advocate true engineering —by definition involving the use of *"scientific and mathematical principles"*[1]— on the basis of direct everyday necessities. Hence one might think: if things work anyway, so much the better. However, from a longer-term viewpoint, the same observations indicate that, intellectually, information technology is a victim of its own economic success.

Indeed, industries dealing with the design of complex and critical systems have an increasing need for methods that provide reasonable confidence in the result, and are often looking for external assistance in the area of formal methods from consulting firms and academia. Arguably, the professionalism that is a *necessity* for complex and critical systems is an *opportunity* for run-of-the-mill systems in terms of design quality (the process as well as the product). The short-term view accepting the status quo lets this opportunity go to waste.

One may also hope that, utilitarian trends notwithstanding, intellectual curiosity and professional pride have not become outdated. Holloway [2] is sufficiently positive in this respect to base his arguments advocating formal methods on the premiss that *"software engineers strive to be true engineers"*. As Parnas [4] points out, it is the ability to use mathematical models that distinguishes professional engineers from other designers. In the same spirit, a significant number of university staff enjoys the intellectual challenge of research in formal techniques and of teaching them to students.

Unfortunately, however, an increasing number of students de-select formal methods in the curriculum, due to various causes and trends.

One such cause is a general mathphobic tendency in society and in education. Even at world-class universities, colleagues observe that students are increasingly unprepared to invest hard work in their study[2], especially mathematics, because of the demand for *immediate gratification* that is instilled in them from various sides, especially via the media. Curricula that yield to this pressure increase the gap between the excellent and the average students, because the latter are less able to patch deficiencies in their education on their own.

Another cause is that, in many EE/CS departments, all too few lecturers have a background in the mathematical foundations of formal methods, while the aforementioned opportunities for getting along without such background limits the incentive of

[1] This is the common element in most definitions on the web. The "and" assumedly serves emphasis only!

[2] Some reports indicate a difference between Western and Oriental, and between male and female students.

their colleagues to develop it. Some lecturers teaching CS students are even mathphobic themselves [1], and in extreme cases lecturers unwilling to proceed beyond traditional mathematics (such as analysis) openly or covertly resist curriculum changes that might give the students a fresh, broader perspective. In brief, sometimes the staff may constitute a larger obstacle than the students to the acceptance of formal methods because of the need to update their courses (Page [3]) or just the intellectual challenge (Gries [1], Wing [5]).

These trends are so pervasive that the small minority of FM educators has little hope to curb them in the near future. More effective in the long term is instilling a higher degree of professionalism in the next generation. This requires in particular a directed, positive action towards helping all students in becoming more motivated towards and proficient in the use of formal methods.

This workshop is intended as an opportunity for exchanging educational experiences and all other kinds of information that may be useful to educators in Formal Methods and related fields for making such action more effective.

Emphasis of this workshop This workshop welcomes short papers, presentations, demonstrations and evaluations describing sharp classroom or lab experiments which have proved particularly beneficial to the students' understanding and motivation for formal methods.

The emphasis is not on (new) theories or methods but on specific illustrations and exercises that can be used by colleagues in their own courses, perhaps applying their own formalisms. The main goals are:

- to share knowledge and experience on the practicalities of teaching and learning formal methods;
- to build a collection of interesting cases, examples, assignments and projects that FM teachers can use in educational activities.

Format The workshop is organized as a forum-like event, with short presentations, demos and informal discussion slots.

After the workshop, provided the evaluation committee decides that there is a sufficient number of high-quality contributions, an agreement will be sought with Springer Lecture Notes in Computer Science about publishing a special volume. In case of a positive outcome, authors will be invited to submit a revised version of their contribution for refereeing.

The following preprints are derived from the initial versions, which have been subject to light reviewing and corrected by the authors based on the reviews.

August 2006

Raymond Boute[3]
José N. Oliveira[4]

[3] Formal Methods Group, INTEC, Universiteit Gent, Belgium
boute@intec.UGent.be
[4] Departamento de Informática, Universidade do Minho, Braga, Portugal
jno@di.uminho.pt

References

1. David Gries, "The need for education in useful formal logic", *IEEE Computer 29*, 4, pp. 29–30 (Apr. 1996)
2. C. Michael Holloway, "Why Engineers should Consider Formal Methods", *Proc. 16th Digital Avionics Systems Conference* (Oct. 1997)
   Web: `http://shemesh.larc.nasa.gov/people/cmh/DASC97`
3. Rex Page, *BESEME: Better Software Engineering through Mathematics Education*, project presentation `http://www.cs.ou.edu/~beseme/besemePres.pdf`
4. David L. Parnas, "Predicate Logic for Software Engineering", *IEEE Trans. SWE 19*, 9, pp. 856–862 (Sept. 1993)
5. Jeannette M. Wing, "Weaving Formal Methods into the Undergraduate Curriculum", *Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST)* pp. 2–7 (May 2000); file `amast00.html` in `www-2.cs.cmu.edu/afs/cs.cmu.edu/project/calder/www/`

Acknowledgments

This workshop was organized by the Formal Methods Europe Subgroup on Education. All submitted contributions were reviewed by the following members of the subgroup and by some external referees whom we'd like to thank for their collaboration:

Dino Mandrioli
Eerke Boiten
Izzat M. Alsmadi
John Fitzgerald
José Carlos Almeida
José N. Oliveira
Kees Pronk
Luís S. Barbosa
Peter Lucas
Raymond Boute
Simão M. Sousa

The preprints of the workshop were type-set in LaTeX using Springer-Verlag's class package `llncs.cls`. The **rtf2LaTeX 2ε** RTF→LaTeX converter was found useful in processing some contributions.

Table of Contents

Contributions

The Use of an Electronic Voting System in a Formal Methods Course

Alice Miller and Quintin Cutts

Department of Computing Science
University of Glasgow
Glasgow, Scotland.
{alice,quintin}@dcs.gla.ac.uk

Abstract. We describe the design of a discussion-based tutorial within an honours "Modelling Reactive Systems" course facilitated with an electronic voting system. The approach combats confidence and breadth issues, and we report on its effectiveness.

1 Introduction

In industry, formal methods are traditionally viewed as obscure, unscalable and lacking in sufficient tool support [4]. In the lecture theatre too, formal methods can seem too mathematical [8] and of little practical use in the development of complex systems [7].

In order to gain any insight into the benefits of formal methods, it is important that students are allowed to develop confidence with a specific formal notation, and use it to solve a wide range of problems [7]. This is usually achieved via practical lab sessions in which students are encouraged to complete a set of problems for practice and assessment, typically using only one formalism, due to the limited time available. We have observed that, with such a restriction, students are unlikely to reflect upon their practical experience and relate it to the other formalisms introduced in the course only theoretically.

In this paper we describe how we have developed a single teaching tutorial, making use of an electronic voting system (EVS), to complement an existing model checking course. The tutorial is designed specifically to consider the problem described above incorporating small group teaching techniques, such as *buzz groups* and *brainstorming* [2], along with the use of an EVS.

The authors are both lecturers in Computing Science at Glasgow University. Quintin Cutts' recent research focuses on the use of an EVS within the context of higher education, and computing science education in particular [6,10]. His focus is on engaging the students with the material either by problem solving or by reflecting deeply on their own understanding or misconceptions.

Alice Miller's research is in the area of formal methods. Specifically her interests are in the use of advanced mathematical techniques, like induction and symmetry reduction, for model checking [11,3]. She is director of the "Modelling Reactive systems course" discussed in this paper.

2 Modelling Reactive Systems Course (MRS4)

"Modelling Reactive Systems" (MRS4) is a fourth year honours course provided by the department of Computing Science at Glasgow University [5], which has now been running for 5 years. Designed to introduce and explore a variety of formal process description and analysis techniques used in the design of reactive systems, the course consists of 20 hours of lectures and 8 of lab sessions. It is divided into two parts. The first part of the course contains an introduction to reactive systems and basic (graphical) modelling formalisms, and focuses largely on the use of the model checker SPIN [9]. The second part (consisting of 6 lectures) is concerned with tools designed for real time systems development (e.g. SDL 2000). This paper relates to the first part of the course. There are, on average, 35 students in the class, with a wide range of ability. The mathematics in the course is kept to a minimum, although some students still struggle with some of the more theoretical concepts – the definition of Büchi automata etc. This is, of course, a well known phenomenon with formal methods courses [8]. Since the students perceive the course to be difficult, they choose not to participate when prompted for a response by the lecturer, in case they get an answer wrong. In addition, the mathematical nature of the course leads the students to assume that every problem has a right or wrong answer, although in many cases (for example, the most appropriate formalism to use in a particular situation) there is more than one possible solution.

Because of the steep learning curve and lack of time available, in the labs the students only gain experience from one formalism (SPIN). There is no scope within these sessions for wider discussion of the issues involved and to generalise experience across the breadth of the subject.

3 Electronic Voting System (EVS)

A typical EVS consists of a set of keypads, one per student, and a receiver connected to a PC. Multiple choice questions can be presented verbally, on the board, by overhead projector (OHP), or using a PC; students then submit their answer using the keypad. Finally, a bar chart showing the collated responses is displayed by the PC at the front of the class. The lecturer can then use the answers, and the increased knowledge about the students' level of understanding provided by the answers, to guide class-wide or buzz group discussions designed to encourage the students to explore the topics more deeply. Students typically report the anonymity of answering and their increased attentiveness to the subject matter at hand as the key benefits of the system; lecturers the feedback on student understanding, which they use to guide their ongoing course delivery.

4 Using EVS in MRS4

In order to address the problems posed by strict lecture plus lab teaching methods described above, we introduced a tutorial session to the course which exploited the EVS.

4.1 Objectives

The major objectives here are:

1. to encourage participation in lab sessions prior to the tutorial
2. to affect self-learning via reading of a prescribed text
3. to promote reflection, an appreciation that issues are not always black and white, and a deeper knowledge of formal methods.

4.2 Methods

The task took the form of a set of 8 multiple choice questions (see below for a set of sample questions). The first 5 questions were assessed (in total they contributed up to 2% of the final mark for this course). The reason for making these questions assessed was to encourage the students to prepare for, and turn up to the session. The percentage was kept small, as it was based on only 1 of 28 teaching hours related to the course. The remaining questions were designed to promote debate within the class. The structure of the questions (and the mark obtainable) was discussed with the class prior to the session.

In order to achieve objective (1) above, the first three questions were based on issues that arose from discussions in the lab sessions related to the example sheets. In order to achieve objective (2), questions $3-5$ were based on the prescribed text, namely "The Great Debates", a discussion of some of the deeper questions relating to formal verification (see [9]). Finally, to achieve objective (3), some more open-ended questions were provided. We will discuss how these questions were used below.

The multiple choice questions We give samples from the set of questions only, due to lack of space.

1. In SPIN, what can be passed as parameters to processes?
(a) global variables and constants
(b) constants and channel names
(c) channel names and global variables

4. One of the following statements about the temporal logics CTL and LTL is true. Which is it?
(a) CTL is much more expressive
(b) the expressiveness of the two logics do not overlap
(c) LTL is more suitable for "on the fly" verification

6. What do you think of the statement "SPIN allows us to accurately model synchronous communication"?
(a) the statement is true
(b) the statement is false
(c) The statement is not exactly true, but it is close enough.

7. Some real world examples of protocols (e.g. IEEE 802.11, FireWire, bluetooth device discovery) include some notion of randomness and probability. However Gerard

Holzmann does not believe that the addition of probabilities to SPIN is necessary. Do you
(a) agree
(b) disagree
(c) neither agree nor disagree (i.e. you have a better solution..)

8. What do you think is the most important thing that is lacking in SPIN?
(a) a reliable, informative type-checker
(b) the ability to model real time
(c) something else (that you can describe)

Responses and Discussion Using EVS the students became increasingly confident in choosing their response and became far more engaged in this session than in other classes. (Until this session there had been almost *no* student participation in whole group sessions.)

Question 1 (and two further questions based on lecture/lab material): Over 80% of students answered these questions correctly (b for question 1). This was gratifying as it showed that discussions within lectures arising from problems in the labs had brought the relevant issues home. Answering these initial questions correctly also encouraged the students to be bold in answering the subsequent questions.

Question 4. 75% of students got this right (c). This demonstrated that they had read and understood the text. As this was the only exposure to CTL model checking that the students had on the course, their success was surprising.

Question 6. This prompted a lively discussion lasting several minutes. Most students went for (c), (which is the answer suggested in the prescribed text). Students were split into buzz groups and asked to come up with examples of synchronous/asynchronous systems. Some good examples arose (email, slot machine, cash machine, ADSL, token ring). The class discovered that it is generally very difficult to decide which systems are synchronous (in terms of communication) and which are not – it depends very much on the actual implementation. Some students expressed surprise (and, in some cases, relief) that an issue such as this was not completely black and white.

Question 7. The responses were approximately half (a) and half (b). The students were asked to discuss the problem for 5 minutes within buzz groups and then two students were selected, one to advocate (a) and one to advocate (b) (each justifying their answer). The main argument for (a) was that one can use non-determinism instead of probabilistic choice to some extent (when considering discrete time probabilistic systems, this point was perhaps too sophisticated for the class). The main argument for (b) was that we can only allow for a limited number of choices using non-determinism. The issues that arose in the discussion were: probability is not suitable for adding to SPIN, but there are other more suitable tools: probabilistic model checkers like PRISM for example; and that probability is a very important aspect of some protocols.

Question 8. Almost all of the students went for (c). This led to some very lively and noisy debate. The students had strong opinions on this issue as they had spent several hours in the lab working with SPIN, and were keen to express their opinions on its perceived shortcomings. We did not attempt to divide the class into small groups in this case as time was running short. Students were invited to volunteer their suggestions to the rest of the class. There followed enthusiastic criticism of the error messages given by SPIN, the poor editor etc.

4.3 Outcomes

- Reassurance that students had read and understood material that could then comfortably be referred to later in the course. The first 3 (of the original 8) questions probed their understanding of the course material, and the remaining questions their ability to self-teach. They could not have understood the later questions if they had not read the text provided.
- Noticeable increase in confidence with students volunteering to answer questions within the class in subsequent lectures. Previous to the EVS session there had been no response from most students to questions asked in the class. After the session there were responses to questions, interruptions from the class and (well-meaning) contradictions from other students.
- The use of EVS was very popular with the students as a means to communicate anonymously. A typical comment in the course feedback was:
  "we enjoyed the session with the handsets more than we thought we would. It was good to argue with each other and made a change from lectures which can be rather boring".
  EVS was also invaluable for providing an immediate summary of responses, enabling further discussion.
- The debate-style questions prompted a great deal of discussion and gave rise to unexpected responses with a high degree of ingenuity. An observer (a lecturer from the Electronic Engineering department at the University of Glasgow) commented thus:
  "The groups behaved like groups in a pub quiz. Correct answers even caused cheering and strengthened the spirit of the group ... Besides the fun aspect the students found it much easier to speak to the whole class after having discussed the issues with their peers – possibly because they had the approval from their peer group."

Note that the major benefit of using EVS was that students had the means to anonymously express their doubts and opinions. This could, to some extent, be achieved by other means, e.g. via a *Moodle* forum [1]. However, we believe that the immediate feedback, within a controlled environment could not have been reproduced in that context. We also point out that our outcomes are based on our observations rather than from the result of a rigorously controlled experiment. We had no control group, like that used in [12] for example. However, we *could* compare the behaviour of students before and after the intervention, and with students taking the course in previous years.

5 Conclusion

We have identified two major problems with the traditional lecture plus lab teaching of Formal Methods, namely low class participation in discussions, and lack of breadth in acquired learning. We have implemented a tutorial session using EVS which has provided an environment in which these problems have successfully been addressed. We are planning to extend the concept to a greater number of sessions in the next academic year.

References

1. K. Brandl. Are you ready to "Moodle"? *Language, Learning and Technology*, 9(2):16–23, 2005.
2. G. Brown and M. Atkins. Studies of student learning. In *Effective Teaching in Higher Education*, pages 150–158, London, 1988. Metheun.
3. M. Calder and A. Miller. Detecting feature interactions: how many components do we need? In M. Ryan, D. Ehrich, and J.-J. Meyer, eds, *Objects, agents and features*, Lecture Notes in Computing Science, pages 45–66. Springer, 2004.
4. E. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996. Report by the Working Group on Formal Methods for the ACM Workshop on Strategic Directions in Computing Research.
5. University of Glasgow Computing Science. Modelling reactive systems course description: `http://www.dcs.gla.ac.uk/courses/teaching/level4/modules/MRS4.html/`.
6. Q. Cutts. Practical lessons from four years of using an ARS in every lecture of a large class. In D. Banks, editor, *Audience response systems in Higher Education*. Information Science Publishers, 2006.
7. C. Dean and M. Hinchey. Formal methods and modeling in context. In C. Dean and M. Hinchey, editors, *Teaching and Learning Formal Methods*, pages 99–112. Academic Press, 1996.
8. K. Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.
9. G. Holzmann. *The SPIN model checker: primer and reference manual*. Addison Wesley, Boston, 2003.
10. G. Kennedy and Q. Cutts. The association between students' use of an electronic voting system and their learning outcomes. *Journal of Computer Assisted Learning*, 21(4):260–268, 2005.
11. A. Miller, A. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *Computing Surveys*, 2006. To appear.
12. A. Sobel and M. Clarkson. Formal methods application: An empirical tale of software development. *IEEE Transactions on Software Engineering*, 28(3):308–320, March 2002.

Introducing Model Checking to Undergraduates

Abhik Roychoudhury

Department of Computer Science, National University of Singapore
abhik@comp.nus.edu.sg

Abstract. Introducing temporal logics and model checking to undergraduate students is usually an involved activity. The difficulty stems from the students' lack of exposure to logics, unfamiliarity with reactive systems and lack of conviction that model checking search can lead to anything practical. Here, I narrate some experiences in attempting to overcome these stereotypes over a period of five years at the National University of Singapore.

1 Introduction

Teaching of formal techniques has always been a topic of discussion and debate in Computer Science (CS) education. CS academics have underlined the importance of encouraging formal system development practices by trying to incorporate them into the CS curriculum (*e.g.* see [2]).

However, in reality the task of convincing students of the value of formal methods could be a formidable one. This is typically because of the paucity of formal methods courses in the CS curriculum which results in students' inherent lack of exposure to formal techniques. Often times, we face the following arguments from our students (or even our colleagues).

- CS students (particularly undergraduates) are not strong enough to learn formal methods, or
- It is difficult to get CS students used to the rigor of formal methods (even if they are capable), or
- Formal methods is mostly about mathematics which is of limited value for building (computer) systems.

As a formal methods educator, I feel that all of these arguments are false. The question is how to do we get past such stereotype arguments which may reside in the minds of our students (who might think that formal methods courses are to be avoided) or even our colleagues (who might perceive formal methods to be of little value).

In this year's International Conference on Software Engineering (ICSE), there was a panel on Formal Methods where the panelists were asked — *if you have $10 million for promoting formal methods how would you invest it ?* Several panelists underscored the value of education for such investment. More importantly, an interesting analogy [1] was drawn with engineering undergraduates who reguarly learn complex mathematical concepts (such as differential equations) thereby strongly alluding to the incorrectness of the usual argument that Computer science undergraduates are not strong enough to learn formal methods.

In this article, I describe some experiences in teaching formal methods to undergraduate students over five years. Primarily, I deal with issues arising from exposing undergraduate students in Computer Engineering to a course on model checking. *All course materials* have been made freely available from

`http://www.comp.nus.edu.sg/~abhik/CS4271`

Before proceeding to elaborate on the teaching methods, I give some background information about the course which may be helpful in judging the applicability of my teaching methods.

2 Background Information about the Course

The course in question was offered as an *elective module* once a year for five years at the National University of Singapore. It was part of the Computer Engineering curriculum, that is, it was offered to our Computer Science students specializing in Computer Engineering. These students take a wide variety of Computer Science courses with a concentration of courses on embedded system design. Consequently, they are required to take four electives on system design from various courses such as:

- *Critical Systems and their Verification*
- *Hardware Software Codesign*
- *Mobile Computing*
- *Performance Analysis of Embedded Systems*
- *Embedded Software Design*
- *An advanced course on Computer Networks*
- *An advanced course on Programming Language Design and Implementation*
- ...

The course we are discussing here is the first one in the above list – *Critical Systems and their Verification*. Note that the other courses in the list are more focused on (embedded) system performance rather than formal techniques. So, our course (which was offered as an elective) is inherently somewhat different from the other elective modules.

Our course on formal verification is taken by third or fourth year Computer Engineering undergraduates with most of the students coming from the fourth year. The total number of students in the Computer Engineering programme is approximately 75 out of which approximately 45 students have opted for the formal verification module in the last three offerings of the module. The exact enrolment numbers over the five years are as follows.

2001-02	2002-03	2003-04	2004-05	2005-06
3	32	34	45	54

Since our verification course was offered as part of the Computer Engineering curriculum, we could not require a course in Logic to be a pre-requisite (since a Logic course in not mandatory in our Computer Engineering curriculum). This in fact made the offering of the course more challenging. The only pre-requisites of our course on formal verification were:

- a first year undergraduate course on Discrete Mathematics (which gives the students brief exposure to propositional and predicate logic), and
- a first year undergraduate course on Computer Organization (which gives the students some exposure to combinational/sequential circuits, buses etc).

Note that most of our students were in their fourth year and they only had a brief introduction to logic in their first year of undergraduate study. Hence it was necessary to communicate to them that the course goes beyond logic/discrete maths. On the other hand, it was also important to refresh their background on logics while introducing temporal logics. In the next section, I proceed to outline the main strategies that were adopted as an attempt to enhance the students' learning experience. Some of these strategies are standard ones, while some were learnt gradually by offering the course multiple times.

3 Strategies to Enhance Students' Learning Experience

For enhancing the students' learning experience, we need to elicit more student interest and participation by relating the techniques (in this case model checking) to real-life. However, this is often done in a rather extreme way by mentioning dramatic historical disasters which happened due to lack of formal verification. Too often we motivate a formal verification technique by mentioning the Arianne space shuttle disaster, or the Therac-25 accidents. If we (the formal methods educators) decide to be a bit more down-to-earth while motivating our techniques, at most we refer to the Intel Pentium floating-point error from 1994 (which resulted in substantial financial loss for Intel). Clearly, mentioning these historical incidents to the students serve an important purpose— they get the students' initial attention/interest. However, from my experience, this interest is often *difficult to retain* — possibly because many of these historical disasters seem to be far removed to the students. Emphasizing these historical incidents also serves to emphasize the students' perception that formal methods is something "exotic" — a perception we as educators should fight against.

Students need to understand, they do not need to be surprised As a first step, I have avoided mentioning historical disasters in my lectures for the purpose of motivating formal verification. Instead, in the first lecture, I try to refer to existing industry practice in "verification and validation" — why these practices do not amount to formal verification, and what needs to be done to achieve formal verification. Since my course is part of a programme with Embedded System focus, I refer to some existing Electronic Design Automation industry practices in this regard (methods like in-circuit emulation). I try to explain how the existing methods are intrusive to the design process and how a model-based technique can help the design cycle. This results in a rather different pedagogical style, where the aim is to discuss the system design cycle with the students rather than impressing/surprising the students with the power of formal verification.

Presenting formal verification as a tool to improve the system design cycle helps. It removes the misconception that formal methods are required only for very very safety-critical systems which normal engineers need not be bothered with. However, until and unless the students can get some amount of gratification from using formal methods,

they are quite likely to forget about it once the semester ends. Often, we (as formal methods educators) take a view that the theory should be taught prior to the tool. In a course focusing on model checking this would mean that the students need to learn about Kripke Structures, Temporal Logics, Explicit-state checking, Binary Decision Diagrams (BDD) and Symbolic Checking — even before they can write a single line of code in a model checker! Clearly, such an approach is unlikely to evoke student interest. We could try to improve the state of affairs by teaching only Kripke Structures, Temporal Logics and explicit-state checking prior to discussing model checkers. However, from my experience, a significant fraction of the students still feel lost by the time the model checkers are introduced. To effectively teach model checking, it is important to discuss system modeling (from requirements) as early as possible.

Discuss System Modeling as early as possible To address this issue, I try to familiarize the students with (at least) the input language of a model checker even before they learn temporal logics and model checking. So, the rough flow of my course (which focuses on model checking) is

- Transition Systems and Kripke Structures
- SMV model checker and case studies
- Temporal Logics
- Explicit-state Model Checking
- Binary Decision Diagrams (BDDs)
- Symbolic Model Checking using BDDs

A few points need to be emphasized at this stage. When we discuss SMV and its case studies, I try to pick moderate sized but real-life case studies. These examples serve an important purpose — they are not toy examples, but they are not so large that their modeling cannot be discussed in details. I believe this is more effective than mentioning some very large case studies, where the students may be more surprised/impressed but they may not understand the intricacies of modeling a real-life protocol. Also, note that when we discuss the SMV model checker and case studies, the students have not yet been introduced to temporal logics. Hence, the properties being verified in the case studies are mentioned informally at this stage and they are formalized in subsequent lectures. The detailed flow of the course is available (in the form of a **Lesson Plan**) from

`http://www.comp.nus.edu.sg/~abhik/CS4271/lesson-plan.html`

Unfamiliarity (with temporal logics & reactive systems) breeds contempt From my experience, students are often uncomfortable with one of the following.

- connection between program behaviors and transition systems,
- understanding reactive systems which have execution traces of infinite length
- interpreting temporal logic formulae over infinite execution traces.

The first hurdle is relatively easy to overcome. A refresher revision hour on operational semantics might help in this regard. However, since the students are typically

familiar with transformational systems it takes them substantial time to make the conceptual switch to systems with execution traces of infinite length. This can be aided by presenting (successively more complex) example transition systems in class and telling the students to list out the infinite execution traces of the given transition system. Getting familiar with reactive systems (and infinite length execution traces) is often the primary hurdle in the minds of students. Once this barrier is overcome, they can (relatively) easily adapt to the concept of Linear-time temporal logic (LTL) and its operators. Branching-time temporal logics are then covered by building on Linear-time logics.

Finally, keep it project-based The final point that I want to discuss here is a lesson that was learnt the hard way. In retrospect, it is probably an obvious lesson but it was not obvious (to me) when I started teaching the course. To give the students hands-on experience with the model checking tools, I had the option of designing a series of assignments or allowing them to choose term-projects. From a pragmatic point of view, managing an assignment-based course is easier (for grading and other purposes). I ran the course in two successive years in two different modes (project based and assignment-based). The student response was overwhelmingly in favor of the project-based version. In retrospect, this was so for more reasons than one.

- A term project allows the student some choice and encourages some independent exploration for fixing the project as well as during modeling/validation.
- A term project gradually builds on itself during the entire semester and is more substantial. This way the students can see the benefit of using model checking on some substantial-sized examples.

I feel that engaging the students in a medium-sized term project might be the best way to convince them of the applicability of formal techniques. However, if the entire class does the same term-project it becomes a bit like an extended assignment depriving the students of a sense of independent exploration. For this reason, it might be important to allow students (individually or in groups of 2-3) choose different term projects even if the module administration becomes difficult.

An initial list of possible project ideas that I gave out to students in my course is available from

`http://www.comp.nus.edu.sg/~abhik/CS4271/proj-ideas.html`

Needless to say, students did projects outside this list as well. We should note that for a course based on independent term projects there are several adminstration issues involved such as counseling the students on deciding their project (particularly this needs to be done at the beginning of the semester when the students are not yet familiar with formal tools/techniques). If the course is offered multiple times, there is the additional issue of modifying/upgrading the list of project ideas in subsequent years.

One should emphasize here that *prior to actually doing their term projects, the students go through substantial experience in modeling and analysis of several case studies*, particularly when I introduce the SMV tool. These include

- medium-sized examples which are fleshed out in full details for the students to grasp intricacies of system modeling (such as the examples in [4]), and

– larger scale real-life protocol verification examples (such as lessons learnt from model checking the AMBA system-on-chip bus protocol running on ARM processors [3]) which lets the students appreciate the value of modeling and model checking.

4 Discussion

The lessons mentioned in this paper should be (at least partially) applicable to various formal methods courses — even those not covering model checking. The generic versions of these lessons are as follows.

– Do *not* rely on historical incidents to motivate formal methods.
– Emphasize system modeling (from requirements) rather than focusing only on verification techniques.
– Introduce verification tools prior to techniques as far as practicable.
– Allow students freedom in doing term projects (rather than assignments or fixed projects) even if module administration becomes difficult.

I sincerely hope that these general issues (which I learnt gradually over a period of five years) and the course materials (which I have made available through the Internet) will be useful to fellow formal methods educators in other universities and institutes.

References

1. John C. Knight. Position statement in Panel "Formal Methods: Too little or too much?". In *ACM Intl. Conf. on Software Engineering (ICSE)*, 2006.
2. Peter J. Denning and others. A debate on teaching Computing Science. Essays in response to Edsger W. Dijkstra's lecture "On the cruelty of really teaching Computer Science". *Communications of the ACM*, 32(12), pages 1397-1414, 1989.
3. Abhik Roychoudhury, Tulika Mitra and S.R. Karri. Using formal techniques to debug the AMBA system-on-chip bus protocol. *Design Automation and Test in Europe (DATE) 2003.*
4. J.M. Wing and M. Vaziri-Farhani. A case study in model checking software systems. *Science of Computer Programming, 28, 1997.*

Basic Science for Software Developers

David Lorge Parnas and Michael Soltys

Department of Computing and Software
McMaster University
1280 Main Street West
Hamilton, ON., Canada L8S 4K1

1 Introduction

Every Engineer must understand the properties of the materials that they use. Whether it be concrete, steel, or electronic components, the materials available are limited in their capabilities and an Engineer cannot be sure that a product is "fit for use" unless those limitations are known and have been taken into consideration. The properties of physical products can be divided into two classes: (1) *technological properties*, such as rigidity, which apply to specific products and will change with new developments, (2) *fundamental properties*, such as Maxwell's laws or Newton's laws, which will not change with improved technology.

In many cases technological properties are expressed in terms of numerical parameters and the parameter values appear in product descriptions. This makes these limitations concrete and meaningful to pragmatic developers. It is the responsibility of engineering educators to make sure that students understand the technological properties, know how to express them, know how to determine them for any specific product, and know how to take them into account when designing or evaluating a product.

However, it is also the responsibility of educators to make sure that students understand the fundamental limitations of the materials that they use. It is for this reason, that accredited engineering programs are required to include a specified amount of basic science (see [9]). Explaining the relevance of basic science to Engineers is a difficult job; technological limitations are used to compare products; in contrast, fundamental limitations are never mentioned in comparisons because they apply to all competing products. As a result, the technological limitations seem more real and students do not perceive fundamental limitations as relevant.

For Software Engineers, the materials used for construction are computers and software. In this area too, the limitations can also be divided into two classes: (1) *technological limitations*, such as memory capacity, processor speed, word length, types of bus connections, precision obtained by a specific program, availability of specific software packages, etc., (2) *fundamental limitations*, such as limits on computability, complexity of problems, and the inevitability of noise in data.

Computer Scientists have developed a variety of useful models that allow us to classify problems and determine which problems can be solved by specific classes of computing devices.

The most limited class of machine is the finite state machine. Finite state machines can be enhanced by adding a "last-in-first-out" memory known as a stack. Adding

an infinitely extensible tape that can move both forwards and backwards through the reader/writer makes the machine more powerful (in an important sense) than any computer that can actually be built. Practicing software developers can use these models to determine how to approach a problem. For example, there are many problems that can be solved completely with the simplest model, but others must be restricted before they can be solved. Many people know these things in theory, but most do not understand how to use the theory in practice.

Like the students in other engineering disciplines, software engineering students must be able to understand and deal with technological limitations. Even the youngest have seen rapid improvements in technology and most of them easily understand the practical implications of those differences.

It is not useful to spend a lot of time on the technological limitations of specific current products. Much of what students learn about products will be irrelevant before they graduate. However, it is very important to teach the full meaning of technological parameters and how to determine which products will be appropriate for a given application.

Nonetheless, 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.

In the next section, we will give a few illustrations to make these points clearer.

2 A few anecdotes

2.1 What can be said with grammars

Many years ago, Robert Floyd encountered a graduate student who was trying to find a complete context-free grammar for Algol-60, one that specified that all variables must be declared before use. The student's plan was to use the grammar as input to a compiler generator. Floyd's understanding of CS fundamentals allowed him to prove that no such grammar could exist. The graduate student was saved months, perhaps years, of futile effort. With this information he understood that he would have to find another way to express those restrictions as input to his compiler generator. [6]

This anecdote makes it clear that it is very important to be able to decide whether or not a task is impossible. Some people spend their lives trying to solve impossible problems.

2.2 The meaning of computational complexity

Computer Scientists have developed ways to classify the complexity of algorithms and to classify problems in terms of the complexity-class of the best solution to those problems. This allows them to determine whether or not an algorithm is as good as it can get (optimal). However, strange as it may sound, sometimes an "optimal" algorithm is not the best choice for a practical application.

In the 70's Fred Brooks, working on visualization tools for chemists, announced that he wanted an optimal algorithm for a well defined problem. A very bright graduate student proposed such an algorithm and submitted a short paper proving that it was optimal. Brooks insisted that the algorithm be implemented and its performance compared with the performance of the method that they had been using; the performance of the "optimal" algorithm was worse than the old one. Computer Science complexity methods refer to asymptotic performance, that is, performance for very large problems. Algorithms that are not optimal may actually be faster than the "optimal" ones for certain values of the key parameters. Since a developer may find an "optimal" algorithm in a textbook, she must be aware of what "optimal" means and check to see that the performance is actually better in practice than other algorithms. Moreover, a developer who knows the asymptotically optimal algorithm can often modify it to produce an algorithm that will be fast for the application at hand.

Another such example is Linear Programming. The widely used Simplex algorithm is known to be exponential in the worst case. However, the Simplex has superb performance in practice (in fact, it's expected performance is provably polynomial). On the other hand, the ("worst-case") polynomial algorithm for Linear Programming, known as the Ellipsoid Algorithm, appears to be impractically slow. [10]

2.3 The practicality of a "bad" solution to the "Knapsack Problem"

In the "Knapsack Problem" the input is a set of weights $w_1, w_2, \ldots, w_d$, and the capacity, C, of the knapsack. We want to pack as much weight into the knapsack, without going over the capacity. The most obvious approach, starting with the largest weights, does not work, because if we have three weights $w_1 = 51, w_2 = 50, w_3 = 50$, and $C = 100$, and our strategy is to pack as much as possible at each step, we would put 51 in, and we would have no more space left for w_2 and w_3. The optimal solution in this case is of course $w_2 + w_3 = 100$.

The "Knapsack Problem" can be solved with Dynamic Programming, where we construct a table with dimensions $d \times C$ (d = number of weights, C = capacity), and fill it out using simple recursion. A classical worst-case running time analysis of the dynamic programming algorithm shows that it requires exponential time. The reason is that the algorithm builds a $d \times C$ table, so if C is given in binary, the size of the table is exponential in the size of the capacity (i.e., exponential in the size of the input). Therefore, the dynamic programming solution to the Knapsack Problem runs in exponential time in the size of the capacity of the knapsack, and hence it is asymptotically infeasible.

In fact, the dynamic programming solution to the "Knapsack Problem" is widely used in Computer Science. In applications such as Compilers and Optimization prob-

lems, equivalent problems arise frequently, and they are solved using dynamic programming. The method is practical, even with many weights, for reasonable C.

One should not interpret this as meaning that the theoretical complexity is useless; *au contraire*, it demonstrates why even practitioners who think that they are not interested in "theory" should understand computational complexity when developing algorithms for difficult problems.

2.4 Maximum size for a halting problem

Two software developers were asked to produce a tool to check for termination of programs in a special purpose language used for building discrete event control systems. One refused the job claiming that it was impossible because we cannot solve the halting problem. A second, who understood the proof of the impossibility of the halting problem, realized that the language in question was so limited that a check would be possible if a few restrictions were added to the language. The resulting tool was very useful. Here again, an understanding of the nature of this "very theoretical" result was helpful in developing a practical tool with precisely defined limitations.

2.5 Can we prove that loops terminate

Dr. Robert Baber, a software engineering educator who has long advocated more rigorous software development [1,2,3,4] was giving a seminar in which he stated that it was the responsibility of programmers to determine whether or not loops they have written will terminate. He was interrupted by a young faculty member who asserted that this was impossible, "the halting problem says that we cannot do that." In fact, the halting problem limits our ability to write a program that will test *all* programs for termination, not about our ability to check *a given* program for termination. This incident shows that a superficial understanding of computer science theory can lead people astray and cause them to be negligent.

Clearly, we must teach fundamentals in such a way that the student knows how to translate theoretical results into practical knowledge. For example, when teaching about the general undecidability of halting problems, one can accompany the proof with an assignment to determine the conditions under which a particular machine or program is sure to terminate. Comparing the general result with the specific example helps the student to understand the real meaning of the general result.

2.6 The implications of finite word length

In 1969, some software developers became enthusiastic about a plan to store 6 bytes in a 4 byte word. They proposed computing the product of 6 bytes and converting the result to a 4 byte floating-point number. Sadly, none of the programmers in the organization understood the impossibility of this scheme and they invested a lot of time discussing it. Luckily, an academic visitor[1] who did understand basic information theory, could convince them of its applicability by providing a counter-example, i.e., an example

[1] Dave Parnas.

where the same output would be obtained for two different inputs. It was quite possible that even extensive testing would not have revealed the error, but it would cause "bugs" in practice.

2.7 The limitations on push-down automata

Recently one of us had occasion to talk to some people who were familiar with the standard results about push-down automata, i.e., that the class of problems that they could solve was smaller than that for Turing machines. He reminded them that in today's market, one can buy an auxiliary disk and attach it to a laptop or other personal computer. He asked if this changed the fundamental properties of the machine (it does not). He then asked what would happen if we could buy an auxiliary push-down stack and attach it as a separate device on a push-down automata that already had one stack. All claimed that the result would still be a push-down automata, i.e., they did not recognize that having a second (independently accessible) stack changed the fundamental capabilities of the machine. The same group included many who did not realize that placing limits on the depth and item size of the stack in a push-down automaton made it no more powerful than any other finite state machine. This meant that they did not understand that there would be an upper limit in the number of nested parenthesis in an expression that would be parsed by any realizable push-down automaton, or that a twin-stack push-down automaton (with infinite stacks) was as powerful as a Turing machine and more powerful than any realizable computer.

2.8 The practical limitations of open questions

There are number of problems in computability and complexity theory that remain open. Many practitioners and students believe that these problems are of interest only to theoreticians. In fact, they have very practical implications. Probably the most dramatic of these is the "$\mathbf{P} = \mathbf{NP}$" question [5] for which a prize of $1,000,000 has been offered. This does not interest most students who realize that they will not win the prize. However, the question has very important implications in cryptography. Some very widely used encoding algorithms are only "safe" if the answer is that $\mathbf{P} \neq \mathbf{NP}$. If it is not, it might be possible to find ways to crack codes quickly (see [5,7]). (The reason is that if $\mathbf{P} = \mathbf{NP}$, then that would imply the existence of a polytime algorithm for factoring, and such an algorithm would render the RSA encryption scheme insecure.)

3 A course in basic science for Software Engineers

McMaster Universities CEAB accredited Software Engineering Program includes a course designed to teach its students both the parts of "theoretical computer science" that they ought to know and how to use them. For a complete outline of the course see [12]; here we mention the main topics: (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.

For four years we used [8] as the textbook, but last year (2005-06) we used [11]. The former was perhaps better suited for engineers, but the latter has a better complexity section and de-emphasizes push-down automata which have lost ground in the last years as a theoretical constuction.

Deeper discussions of the basic subject matter can be found in [7,8,10]. However, these references do not discuss educational motivations as we do.

4 Conclusions

Established engineering accreditation rules require that each engineering student have a minimum exposure to basic science. As accredited software engineering programs are relatively new, there is no clear understanding what constitutes appropriate basic science. Although we believe that every Engineer should have been taught basic physical science, we believe that those who will specialize in software require a thorough exposure to the topics discussed above. This paper has illustrated why, a course on these topics should be required as part of the basic science component of a program for engineers specializing in software intensive products.

References

1. R.L. Baber. *Software Reflected: the Socially Responsible Programming of Our Computers*. North-Holland Publishing Co., 1982. German translation: Softwarereflexionen: Ideen und Konzepte für die Praxis, Springer-Verlag, 1986.
2. R.L. Baber. *The Spine of Software: Designing Provably Correct Software—Theory and Practice*. John Wiley & Sons, 1987.
3. R.L. Baber. *Error Free Software: Know-How and Know-Why of Program Correctness*. John Wiley & Sons, 1991. German original: Fehlerfreie Programmierung für den Software-Zauberlehrling, R. Oldenbourg Verlag, München, 1990.
4. R.L. Baber. *Praktische Anwendbarkeit mathematisch rigoroser Methoden zum Sicherstellen der Programmkorrektheit*. Walter de Gruyter, 1995.
5. Stephen Cook. The P versus NP problem. `www.claymath.org/prizeproblems/p_vs_np.pdf`.
6. Robert Floyd. On the nonexistence of a phrase structure grammar for ALGOL 60. *Communications of the ACM*, 5(9):483–484, 1962.
7. Michael R. Garey and David S. Johnson. *Computers and Intractability*. Bell Telephone Laboratories, 1979.
8. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2000. We used the 2nd edition, but the 3rd edition is now available.
9. Canadian Council of Professional Engineers. Accreditation criteria and procedures, 2005. `http://www.ccpe.ca/e/files/report_ceab.pdf`.
10. Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
11. Michael Sipser. *Introduction to the Theory of Computation*. Thomson, 2006.
12. Michael Soltys. `http://www.cas.mcmaster.ca/~soltys/se4i03-f02/course-outline.txt`.

Two courses on VDM++ for Embedded Systems: Learning by Doing

Peter Gorm Larsen

Engineering College of Aarhus, Dalgas Avenue 2, DK-8000 Århus C, DK
http://www.iha.dk email: pgl@iha.dk

Abstract. This small paper presents two new courses that aim to improve the students' ability to aquire the basic skills underlying formal methods and motivate them to use principles from here in their subsequent professional life. The two courses are combined by a project that must be carried out by the students in order to learn to apply this kind of technology in practice. The project is carried out first at a high abstraction level and afterwards including concurrency and real-time aspects. The main rationale behind these two courses is using formal methods in a light-weight fashion on examples inspired by industrial usage using appropriate tool support. We believe that such a pragmatic approach using abstract modelling and test of such models rather than formal verification is a route which is more likely to succeed from a teaching perspective at university level. We believe that students with this approach obtain a higher level of appreciation and skills for thinking abstractly and precisely.

1 Introduction

Having recently returned to academia from a long period in industry we think that it is paramount to be able to motivate students with realistic examples. This gives the students confidence in the ability to apply formal techniques industrially. Seeing a formal technique applied to trivial examples such as stacks and dictionaries does not provide the students with an appreciation of the ability to apply the technique for real system development.

The two 5 ECTS point MSc level courses that is the focus of this short paper are based on the new VDM++ book [1]. In addition, practical projects are arranged for the students. Here they need to learn abstraction "by doing" it themselves on their projects. Note that although the projects carried out by the students are not large they are all inspired from real systems and thus have more of a flavor of realism. The target for these courses are primarily students studying the development of embedded software-based systems. In this context the objective of the courses is to equip the students with skills regarding abstraction and rigour. Thus the examples used are on purpose selected as reactive systems.

Each course lasts seven weeks each of which consists of one lesson lecturing theory from the book and one lesson where each group presents the status of their project (http://kurser.iha.dk/eit/tivdm1/). The division of the material between the courses is made such that in the first course focus is on teaching the basic modelling principles using a combination of UML and VDM++. In the second course concurrency

and real-time aspects are then introduced and more attention is paid to alternative validation techniques. In this way the students are forced to start with abstract modelling and the subsequently move downwards on the abstraction ladder gradually.

In both courses the students have a substantial project to carry out. The intent is that in the first course a more abstract model of the project is made. A more elaborate model of the same project is then made in the second course. The projects form the basis for an oral examination used to assess the abilities of the students after each of the courses.

2 Structure of the first course

The lessons in the first course are structured as:

1. Introduction and development process (chapter 1+2 from [1])
2. VDMTOOLS and logic (chapter 3 from [1])
3. Defining data and functionality (chapter 4 + 5 from [1])
4. Modeling using unordered collections (chapter 6 from [1])
5. Modeling using ordered collections (chapter 7 from [1])
6. Modeling relationships (chapter 8 from [1])
7. Course evaluation and repetition

All these lectures are presented with a high level of student interaction. Typically, practical questions are presented on the slides right after a subject has been introduced. The students know that in effect they take turns in answering these questions. As a consequence they need to stay alert and active to be able to answer the next question they get. This has the consequence that the students learn the new concepts over a longer period than if they just prepare for an oral exam. We believe that learning the topics in this fashion increase the retention rate. The subsequent evaluation by the students also showed that they really appreciated this approach (see Section 3).

The suggested projects for the students to work on initially was:

1. SAFER (Simplified Aid For EVA (Extra Vehicular Activity) Rescue) from [2]
2. Production Cell from [3]
3. Cash Dispenser from [4]
4. CyberRail from [5]
5. Conveyor belt from “Automation BSc course”
6. Projects from “Distributed Real-Time Systems” MSc course
7. Projects from “Specification of IT Systems” MSc course
8. Suggest your own project

For some of these projects substantial VDM modelling has already been taking place whereas for others only ideas were present and/or models are provided in different formalisms. However, the students that selected projects where substantial VDM modelling had already been done, naturally had to read and understand the existing models first. Afterwards they would suggest what kind of alternations they would like to make to the model. In this way they have been trained in the ability to comprehend a model made by someone else. For the students selecting projects without any existing VDM modelling more freedom was present. On the other hand this meant that their main challenge was to decide upon the right level of abstraction.

The projects selected by the different groups of students were:

1. SAFER
2. Production Cell
3. Cash Dispenser
4. CyberRail
5. Car radio navigation system
6. Self navigating vehicle
7. Personal Medical Unit

The three latter projects are all projects suggested by the students themselves. All groups had to use both VDMTOOLS and Rational Rose for their modelling. No specific training in the use of these tools where provided, but it turned out that a little more practical introduction to VDMTOOLS would have been an advantage[1]. The overall objective for the projects was to let the students learn to think in terms of abstract models by doing it themselves. They should all end up with a consistent and abstract VDM++ model that could also be viewed as a UML class diagram. In addition the models should be validated using traditional test techniques with the VDMTOOLS interpreter.

During the course there was a strict number of milestones with one presentation about progress made by each group per week. In this way the internal communication between the different groups was also facilitated because they had to give each other feedback. Each group of students have handed in a report about their project and these will be made publically available. Unfortunately some of the reports are in Danish so it will only be partly useful for international teachers. We will consider in the future to make it mandatory to all students writing their reports in English.

3 Course Evaluation

After each course a standard course evaluation is carried out at the Engineering College of Aarhus. The questions used in this questionaire are:

1. What is your judgement of the course relevance in your study?
2. What is your judgement of the difficulty of the course?
3. What is your judgement of the course material used?
4. What is your judgement of the teacher as a whole?
5. How do you like the education form?
6. How has the internal relationship been between the students?

All 15 students attending this course returned the questionaire with the following results to the different questions (in the same order):

1. 6 major relevance and 9 suitable relevance (none for small or very small)
2. 2 easy, 8 suitable and 5 difficult (none for very difficult)
3. 5 very good and 10 good (none for less good or bad)
4. 12 very good and 3 good (none for less good or bad)

[1] Our plans for the next academic year include conduction a few small practical exercises with VDMTOOLS together with the students during the first week of the course to accomodate for this.

5. 4 very good, 10 good and 1 less good (none for bad)
6. 6 very good and 9 good (none for less good or bad).

Thus, although this course was very different from other courses that the students have had during their studies the overall impression was that this was worthwhile. We believe that even if not all students use formal methods in their subsequent career at least they will have obtained a much higher appreciation for performing abstract considerations and will be dedicated to a higher level of professionalism. Certainly the additional comments made by the students indicated that they now have become motivated to look more into this direction.

With respect to the projects carried out by the students it turned out that all the projects were appropriate for the students to be able to acquire the necessary experience in trying on their own to produce and analyse abstract VDM++ models. However, projects stemming from a pre-existing VDM model were less challenging for the students because an appropriate level of abstraction was already found. Thus, that kind of projects are most appropriate for the weaker students. On the other hand the projects where no model exists prior to the students undertaking this job there is potential to explore and form the project to more easily ensure as much coverage of the curricium for the course in their models.

4 Performance at the Exam

At the final exam for the course the students had been instructed to prepare an oral presentation of 10 minutes (maximum) without the use of multimedia support. They had also been instructed that in order to examine their skills they should not expect to be allowed to complete the presentation without interruption. Each student was examined for 15 minutes and 5 additional minutes were used to decide upon the grade with the external examiner. The interruptions was made harder, the better the report handed in was judged in advance, and the better the student performed in the oral presentation. The examination attempted to cover as much of the curriculum as possible both in depth and in breath.

In Denmark the grading system goes from 0 to 13 and the grade 6 is needed to pass an exam. The normal average for courses is 8 which is the stable average performance by a student. In this case the students performed outstandingly well and demonstrated great insight into the curriculum. An overview of the grades granted is given in Table 1.

Grade	number of students
8	1
9	2
10	5
11	6
13	1

Table 1. Exam results from the first VDM++ course

I have not experienced any exams in the past at the MSc level where the grades have been this high (an average above 10 where it would normally be around 8). Thus, there are reasons to believe that the teaching approach chosen in this course actually has a positive effect on the learning ability of the students and their appreciation of the subject.

5 Structure of the second course

This course is only planned and it will be starting October 2006 so it is naturally still to early to evaluate its value. The lectures in this course are structured as:

1. Model Structuring and Combining Views (chapter 9 and 10 from [1])
2. Concurrency in VDM++ (chapter 12 from [1])
3. Real-time modelling in VDM++ over two weeks
4. Distributed systems in VDM++
5. Model Quality (chapter 13 from [1])
6. Course evaluation and repetition

In [1] there is no coverage of real-time or distribution. Thus, the real-time and distribution subjects will be taught using a combination of [6] and [7]. Both of these poblications naturally lean on from [1] so the main difference for the students here is the kind of primitives used to describe models. This will be experimental in the sense that this is very new research that recently has been published. From a practical point of view the main risk here is that the tool support for these extensions are under development at the moment. It will be interesting to see how this develops, since the students will most likely be the first real users of the new tool support.

6 Concluding Remarks

We believe that a pragmatic approach to introducing light-weight use of formal techniques with lots of hands-on experience for the students is a very efficient way to get them interested in using this kind of technology once they have graduated and started working in industry.

We also think that the principles for modelling systems at different abstraction levels without having to do formal refinement justifications is a sound one, because it is our experience that if developers are able to master where to put the appropriate level of abstraction the likelihood of successful development is increased significantly.

Acknowledgments

The author wish to thank Erik Ernst, John Fitzgerald, Jozef Hooman, Troels Fedder Jensen, Marcel Verhoef, Stefan Wagner and the anonymous referees for their valuable comments and support when writing this paper.

References

1. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005)
2. NASA: Formal Methods, Specification and Verification Guidebook for Verification of Software and Computer Systems. Vol 2: A Practitioner's Companion. Technical Report NASA-GB-001-97, Washington, DC 20546, USA (1997) Available from http://eis.jpl.nasa.gov/quality/Formal_Methods/.
3. Lewerentz, C., Lindner, T., eds.: Formal development of reactive systems: case study production cell. Volume 891 of LNCS. Springer-Verlag, New York (1995)
4. Group, T.V.T.: A "Cash-point" Service Example. Technical report, IFAD (2000) ftp://ftp.ifad.dk/pub/vdmtools/doc /cashdispenser.zip.
5. : The Concept of CyberRail. http://cyberrail.rtri.or.jp/english/ (2006)
6. Group, T.V.T.: Development Guidelines for Real Time Systems using VDMTools. Technical report, CSK (2006)
7. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and validating distributed embedded real-time systems with VDM++. In Misra, J., Nipkow, T., eds.: FM 2006, Formal Methods Europe, Springer (2006) to appear.

Comments on several years of teaching of modelling programming language concepts

J.W. Coleman, N.P. Jefferson and C.B. Jones

School of Computing Science
Newcastle University
NE1 7RU, UK
e-mail: {j.w.coleman,n.p.jefferson,cliff.jones}@ncl.ac.uk

Abstract. This paper describes an undergraduate course taught at the University of Newcastle upon Tyne titled *Understanding Programming Languages*. The main thrust of the course is to understand how language concepts can be modelled and explored using semantics. Specifically, structural operational semantics (SOS) is taught as a convenient and light-weight way of recording and experimenting with features of procedural programming languages. We outline the content, discuss the contentious issue of tool support and relate experiences.

1 Introduction

The course discussed in this paper[1] is entitled "Understanding Programming Languages".[2] (For brevity, the course is referred to below by its number "CSC334".) It teaches the modelling of *concepts* from programming languages. Formally, it covers operational semantics using parts of VDM for the formal notation (including an unconventional emphasis on "abstract syntax" (see Section 3)) but tries not to labour the formalism itself. The teaching objectives are about the student being able to read a formal (operational) semantics and to experiment with language ideas by sketching a model.

The School of Computing Science at University of Newcastle Upon Tyne offers several undergraduate "degree programmes" each of which includes CSC334 as an optional final year course. The course is taught to a wide variety of students with varying degrees of experience with formal methods.

There are no prerequisite courses for the CSC334 course, but students from most degree programmes take compulsory 2nd year courses that teach VDM as an introduction to formal methods (the textbook used for this is [2]). Interestingly, the School of Computing Science does not offer a course on compilers and this has to be taken into account in the delivery of CSC334.

2 Instructor's Motivation

There are several reasons for teaching formal semantics at undergraduate level. Probably the strongest can be motivated from the "half life of knowledge" that can be imparted: programming languages come and go — over previous ten year periods, there

[1] A longer version of this paper can be found in [1].
[2] A book with the same title is being written.

have been complete changes in the fortunes of one or another language (e.g. Pascal, Modula-n, Ada, C, C++ and Java just within main line procedural languages). Any language that we teach in a university course today might be added to the list of faintly remembered languages in a decade's time. It therefore behoves academics to try to teach something which will last longer and give students a way to look at future languages. There are of course very good books on comparative languages (a recent example is [3]). The fact that so many of the programming languages –even those which are widely used– exhibit really bad design decisions is also worrying and indicates that there is a need to give future computer scientists ways to explore ideas more economically than by building a compiler.

The idea of teaching students a way to *model* concepts in programming languages is attractive in itself but it also provides an opportunity to say things about the fundamental nature of Informatics. Computing science is not a natural science in which one is stuck with modelling the universe as it exists; neither is it usefully viewed as a branch of mathematics as one cannot ignore what can be realized in an engineering sense. This tension is nowhere more clearly seen than in the design of programming languages. Language designers must find a compromise between clarity of expression of programs written *in* the language and reasonable performance of implementations *of* the language. Of course, this list could be extended to include all sorts of issues like the ability to diagnose programmers' errors but the essential tension is that indicated above.

To actually model these concepts we use operational semantics. The essence of operational semantics is that it provides what John McCarthy called an "abstract interpreter" for the language under study. Both words are important. An interpreter makes clear how programs are executed; for an imperative language, it shows how statements cause changes to the state of the computation. The importance of this being described "abstractly" cannot be overemphasized: the interpretation can be understood and reasoned about because it is presented in terms of abstract objects.

This interpretative framework allows the students to reason about the language in terms of the overall system. This includes the intermediate configurations generated during a system's operations, and is in sharp contrast to semantic definitions that are defined only in terms of a mapping from inputs to outputs.

The course then explores language definition questions. Concerns about the separation of syntactic and semantic issues, and the problems inherent in extending a language with new features are handled; practical coursework is used to illustrate how interactions between language features can have deep structural implications for the language. The nature of procedural languages is covered followed later in the course by object-oriented languages, and the two styles are contrasted by examining the roles procedures and objects play. Typing of values and variables in a language is handled mostly at the "static-checker" level, unfortunately the scope of the course does not permit a thorough coverage of the error-handling techniques required for dynamic typing. The topic has, however, come up during the practical sessions with some frequency. Lastly, the link between programs and data in the overall system has been covered as time permitted.[3]

[3] Though we do not go into the LISP-like notions of programs as data, some students have made the connection independently.

3 Technical material covered

Because the interest is in *modelling* rather than the meta-theory of semantics, the course teaches by example. A series of three language definitions are tackled: *Base*, *Blocks*, and *COOL*.

- *Base* introduces the basic idea of states and abstract interpretation; after beginning with a simple deterministic language, concurrency is used to explain the need to cope with non-determinism; a trivial (and rather dangerous) form of threads with sequences of unguarded assignments is modeled using "Plotkin rules" (see Section 4)
- *Blocks* includes Algol-like blocks and procedures; it is used to show how the key idea of an "environment" can be employed to model sharing and the normal range of parameter passing mechanisms are discussed
- *COOL* is a concurrent object-based language; this is where the rule form of description really pays off. The language is rich enough to explore many alternatives.

The natural division of discussing syntax and semantics (and the difficult to place issue of context dependencies) is used. Before addressing the semantics of a language, it is necessary to delimit the language to be described. A traditional concrete syntax defines the strings of a language and suggests a parsing of any valid string. The publication of ALGOL-60 [4] solved the problem of documenting the syntax of programming languages: "(E)BNF" offers an adequate notation for defining the set of strings of a language. Most texts on semantics are content to write semantic rules in terms of concrete syntax. Although this is convenient for small definitions, it really does not scale up to larger languages. We therefore base everything on *Abstract Syntax* descriptions, and in particular, we use VDM-style records to define the structure of the language.

Using abstract syntax has the advantage of immediately getting the students to think about the information content of a program rather than bothering about the marks inserted just as parsing aids. There is an additional bonus that pattern matching with abstract objects gives a nice way of defining functions and rules by separating the definitions into cases.

The class of *Program*s defined by any context free syntax (concrete or abstract) is too large in the sense that things like type constraints are not required to hold. There are many ways of describing *Context Conditions* but we prefer to write straightforward recursive predicates over abstract programs and static environments rather than, for example, use type theory as in [5].

So, given a class of "well formed" abstract programs, how do we give the semantics? McCarthy's formal description of "micro-ALGOL" [6] defines an "abstract interpreter" which takes a *Program* and a starting state and delivers the final state. This "abstract interpreter" is defined in terms of recursive functions over statements and expressions.

We have taught both recursive functions and Plotkin rules in the course as means of defining the semantics of the language. However, as of this past year we have dropped them in favour of focusing solely on Plotkin rules.

4 Plotkin rules

Non-determinism arises in many ways in programming languages. Certainly the most interesting cause is concurrency but it is also possible to illustrate via non-deterministic constructs like Dijkstra's "guarded commands". Unfortunately, McCarthy's idea to present an abstract interpreter by recursive functions does not easily cope with non-determinacy. Defining the recursive functions so that they produce a set of states is not convenient because of the bookkeeping requirements.

In 1981, Gordon Plotkin produced the technical report on "Structural Operational Semantics"[4] [9]. This widely photo-copied contribution revived interest in operational semantics.

The advantage of the move to such a rule presentation is the natural way of presenting non-determinacy. Many features of programming languages give rise to non-determinacy in the sense that more than one state can result from a given (program and) starting state. This natural expression extends well to concurrent languages. The advantage of the rule format appears to be that the non-determinacy has been factored out to a "meta-level" at which the choice of order of rule application has been separated from the link between text and states. For this reason, the complications of writing a function which directly defines the set of possible final states are avoided. Here is a case where the notation used to express the concept of relations (on states) is crucial.

5 Tool support

A key question for teaching CSC334 has been the use of tool support. Tool support has only been used in the teaching of CSC334 during some of the years it has been offered, and is actually an addition of the second author. The inclusion of tool support has both deepened the understanding of some students, as well as increased the confusion for others. There is the extra burden of learning to use the tool as well as the differences between the tool's ASCII syntax and the classroom syntax. Because of this, the tool has been an optional but fully supported part of the course, and the choice of use was entirely left to the student.

The tool used is the CSK VDMTools® [10,11] which many of the CSC334 students have experience of from other courses. It provides an environment in which a VDM specification may be syntax- and type-checked and explicit functions may be executed via an interpreter. The students are provided with language specifications translated into ASCII VDM-SL, notably with the semantic rules translated into functions so that they can be executed in the Toolbox interpreter. This translation, in some cases, produces functions that are significantly different than the original semantic rules that are taught in class.

Beyond the syntactic differences between the original semantic rules and their encoding in the tool, there are often cases where the two versions of a semantic rule or function are wildly different. This is most evident when translating an implicit definition: the tool cannot directly encode implicit definitions, so an explicit equivalent must

[4] This material, together with a companion note on its origins [7], has finally been published in a journal [8].

be created. Unfortunately, the process of doing this often results in a large, ugly and confusing specification. It is of no practical benefit for the students to study and comprehend these explicit definitions; it is important that they focus on the meaning of the semantics and not the implementation issues. Because of this the students are shielded from much of the underlying explicit implementation by separating it from the main language specification through the use of mechanisms made available by the tool.

It is our belief that for some students at least, the benefits of using the tool outweigh the negatives. Through use of the tool, the students can easily identify bugs in their VDM syntax; quickly spot type errors in their specifications; and execute test programs to test and improve their understanding.

The vast majority of mistakes made are errors in the semantic definitions and therein lies the major benefit of using the tool. The execution of test programs highlights such semantic slips and allows greater understanding by directly showing students the consequences of their design decisions.

6 Pedagogic experience

This course is evaluated positively by the students who take it. As an optional course, they obviously tend to self-select and about one quarter of the potential cohort choose to pursue it. The limited number (approximately 20–40 students) makes it possible to adopt a reactive learning environment experimenting with ideas from the students.

The practical work of CSC334 is based heavily on problem solving. Threading through the semester is a large project to make non-trivial extensions to the provided language definitions, and the lecturer tries to keep things timed so that he is introducing concepts just before they are needed. The format of the course's final exam stresses problem solving: it is an open-book exam, and is based on a language specification included from the lectures.

One of the course's final events has evolved over the past few years. As initially run, a few of the students were chosen to study one of the language specifications used in the lectures[5], and they would have the chance to grill the lecturer on the choices made in the design of that language. Their role included gathering comments and questions from their classmates, though the unchosen students also had the opportunity to ask questions through the session. This walkthrough of the language had the side-effect of debugging the language design; the design errors found by the were very instructive.

This exercise transformed first into the lecturer redeveloping a portion of one of the specifications during the lectures, then in the following year, a larger portion of the specification was redeveloped. These lectures had several aims: eliciting direct student participation in the writing of the specification[6]; showing how errors are made during specification and how to both discover and correct them; and to give a real demonstration of the kind of thinking that is needed to do this kind of development — teaching directly by example. While the lecturer did have the language specification to hand, its use was kept to a minimum: mainly to keep the names of the variables synchronized with their notes.

[5] The same specification that would be used in that year's exam.

[6] Mainly by continually asking the class what else was needed for a given rule.

There is, of course, much related material that could usefully be taught on semantics. Textbooks such as [12] and [13] provide excellent introductions to the basic notions of semantics, but –to our taste– do so without a practical context. Their concern with meta-properties of the language would motivate our students less well than experiments with modelling a range of programming language issues.

A preliminary analysis of the feedback from the students suggests that they consider the practical portion of the course to be the most effective in gaining an understanding of the core course ideas. We would conjecture that this arises from the problem-oriented nature of the course: the students are warned at the start of the course that the exam is a problem-based, and to pass the exam they will have to apply the course material.

From both the feedback as well as from discussions with the small groups during practical sessions it appears that the students agree with the notion that the course content has a longer "half-life" than language-specific details. Part of this, we believe, is the realization that design decisions made in languages can be quite arbitrary when there are several ways to model a given feature.

Tool support for this course is explored in greater depth in [1]; analysis of the related effects was omitted from this version as the last run of the course did not use the tool.

Acknowledgments All of the authors acknowledge the support of the EPSRC *DIRC* project. The first and third authors also acknowledge support from the EU IST-6 programme project *RODIN*, and the ESPRC project *"Splitting (Software) Atoms Safely"*. In addition the second author is grateful to the EPSRC funded *Diversity with Off-The-Shelf* (DOTS) project for providing his studentship.

References

1. Coleman, J.W., Jefferson, N.P., Jones, C.B.: Black tie optional: Modelling programming language concepts. Technical Report Series CS-TR-844, School of Computing Science, University of Newcastle Upon Tyne (2004)
2. Fitzgerald, J., Larsen, P.G.: Modelling systems: practical tools and techniques in software development. Cambridge University Press (1998)
3. Watt, D.A.: Programming Language Design Concepts. John Wiley (2004)
4. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised report on the algorithmic language Algol 60. Communications of the ACM **6**(1) (1963) 1–17
5. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
6. McCarthy, J.: A formal description of a subset of ALGOL. In: [14]. (1966) 1–12
7. Plotkin, G.D.: The origins of structural operational semantics. Journal of Logic and Algebraic Programming **60–61** (2004) 3–15
8. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming **60–61** (2004) 17–139
9. Plotkin, G.D.: A structural approach to operational semantics. Technical report, Aarhus University (1981)
10. CSK: VDMTools®: VDM-SL Toolbox Manual, www.vdmbook.com/tools.php. (2006)
11. CSK: VDMTools®: The CSK VDM-SL Language, www.vdmbook.com/tools.php. (2006)
12. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. Wiley (1992) Available at http://www.daimi.au.dk/∼bra8130/Wiley_book/wiley.html.

13. Winskel, G.: The Formal Semantics of Programming Languages. The MIT Press (1993) ISBN 0-262-23169-7.
14. Steel, T.B.: Formal Language Description Languages for Computer Programming. North-Holland (1966)

A Graduate Seminar in Tools and Techniques

Patrick J. Graydon, Elisabeth A. Strunk, M. Anthony Aiello, and John C. Knight

Department of Computer Science
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740, USA
{graydon, strunk, aiello, knight}@cs.virginia.edu

Introduction

In the spring of 2006 we offered a graduate seminar designed to introduce graduate students to the tools and techniques that are being used to construct dependable systems. Our target audience was a group of graduate students who are or will be conducting research in software engineering, particularly in the dependable systems arena. We had several goals with this course. First, we wanted to familiarize these students with the state of the art and practice so that they would be better able to make research contributions. Second, we wanted to help the students learn about scientific critique by asking them to assess the strengths and weaknesses of the tools, both through review of the literature written about the tools and through hands-on experience. Finally, we wanted them to compare some of the tools within this class as an initial effort in helping both researchers and practitioners choose which tools are most suited for the kinds of problems encountered in our profession.

We chose to focus on two distinct topic areas: model checking and model-based development. In either area the students had much to learn, and so either could have been the focus of an entire course. We chose to cover both primarily because these are both important topic areas and secondarily to determine the feasibility of adequately covering both areas in one course. We believe this was a good approach to take, because there is some overlap between the two areas, and because the difference between them underlines the need to assess the suitability of particular analysis techniques when solving problems.

We did not intend to make the students expert, or indeed proficient, in the use of any one model checking or model-based development tool. Rather, we aimed to make students familiar with the technical concepts behind the tools we covered, the kinds of problems to which each tool is applicable, and the power, capability, and limitations of each tool. Each student chose one particular tool to study in more depth than the others, providing the class with insight into the tool and providing general knowledge about the motivation behind that tool. Then the class as a whole learned what the student experts taught, and discussed each student's tool in isolation and with respect to the others. In this way, we wanted to provoke thought and discussion about how and when the tools should be used, the ways in which each tool succeeded or failed, and how the tools compared to each other.

Course outline

At the beginning of the course, each student selected a model checker or model-based development tool. He or she then became solely responsible for presenting it to the other students throughout the course. Students chose the following tools:

- **SLAM** A model checker for device drivers developed by Microsoft [1].
- **BLAST** A model checker for code-level properties developed at the University of California, Berkeley [2].
- **Kronos** A model checker for complex real-time systems developed at Verimag [3].
- **Spin** A model checker for concurrent systems originally developed at Bell Labs [4].
- **SCRtool** A specification tool developed at the Naval Research Laboratory [5].
- **Perfect Developer** A design-by-contract development tool from Escher Technologies [6].
- **SCADE** A model-based development tool produced by Esterel Technologies [7].
- **Simulink** A model-based development tool from The MathWorks [8].

We divided the course into three phases: (1) presentations; (2) laboratory exercises; and (3) comparison discussions. Our course was scheduled for two 75-minute meetings per week over a 15-week semester: the presentation phase occupied the first 9 weeks; the laboratory exercise phase occupied the next 4 weeks, and the comparison presentation phase occupied the final 2 weeks of the course.

In the presentation phase, each student gave a 75-minute presentation in class to familiarize the other students with his or her chosen tool. These initial presentations were intended to convey the overall purpose, organization, and capability of each tool. During each presentation, we encouraged all students to propose questions that would help to evaluate the purpose, capability, and limitation of the tool being presented. In order to prepare students in the class for a detailed presentation and to ask probing questions, the presenting student selected and assigned pre-reading material from the available literature.

In the laboratory phase, each student prepared and directed a 75-minute laboratory session designed to give the other students the opportunity to get hands-on experience with his or her chosen tool. Students were instructed to prepare laboratory exercises that familiarized other students with the tool's user interface, the process of using the tool, and the tool's merits.

The comparison phase was split into two parts: model checking and model-based development. For each part, the students who had chosen that type of tool collectively prepared a presentation comparing and contrasting their tools, which was followed by general discussion lasting the remainder of the 75-minute course period.

The presentations, laboratory exercises, and comparison discussions did not occupy all of the scheduled class meetings. In order to fill the remaining meetings, we scheduled discussions of related papers and guest lectures on related topics. During the presenta-

tion phase in particular, we used this technique to give students time to acquire and study their tools.

Students in the course were asked to prepare both a laboratory write-up and a paper describing and analyzing their chosen tool. The laboratory write-ups included a description of the laboratory and all of the materials used in it. The papers were required to be at least 7 pages in length and to include: (1) a description of the tool and its developers; (2) a summary of the problem it was created to address; (3) a discussion of its capabilities; (4) an analysis of the problems to which it is suited and unsuited; and (5) a comparison with the other tools studied in class. Course grades were based upon participation (5%), the presentation (10%), laboratory draft (10%), final laboratory write-up (30%), draft project report (10%), and final project report (35%).

Outside of class, each student investigated his or her chosen tool, prepared a presentation on it, read assigned reading, prepared a laboratory session, and wrote a project report. Of the students responding to a questionnaire distributed at the end of the course, 2 reported spending 1-3 hours per week on these tasks outside of class, 1 reported spending 4-6 hours, and 2 reported spending 7-9 hours.

Some of the tools we examined were not freely available, but in most cases we were able to get the tool's manufacturer to allow us to use the tool in the course without cost. In the remaining cases we located a similar, freely available tool and asked the student present that instead.

An example course unit: SCRtool

As an example of the approach that we used, we describe one student's efforts to present SCRtool, the prototype tool built by the Naval Research Laboratory (NRL) to support the Software Cost Reduction (SCR) tabular notation. Note that the preparation of the material described in this section was part of the class itself. The student who presented the SCRtool (one of us, Graydon) developed the complete presentation and laboratory, and participated in the preparation of the discussion.

A product of research at the NRL by Heitmeyer et. al., SCRtool is a prototype tool intended to allow software engineers to create concise, unambiguous requirements specifications for embedded systems. The tool is designed around the tabular specification notation developed by Heninger and Parnas as part of the SCR efforts.

The presentation

Before the presenting this work, we asked students to read Heitmeyer's "Managing Complexity in Software Development with Formally Based Tools" [1] and Heninger's "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications" [10]. We selected the former paper because it provides a short, accessible description of the SCRtool team's vision for tools. We chose the latter because it describes both a forebear of the tabular notation used by SCRtool, and because it illustrates the thinking that underlies that tabular notation. There are numerous papers on the SCRtool and the theorem-proving technology of which it makes use, but we felt that the papers we selected provided a good introduction to the kinds of problems the

SCRtool is intended for use with and the overall specification approach advocated by its authors.

Because the tool's authors were in the process of preparing a new version of SCRtool for release at the time of the presentation and suggested that we wait for this version rather than use an older one, we elected to focus the presentation on the tabular SCR notation rather than the tool itself. We were able to obtain the new version after the presentation.

The laboratory exercise

We wanted to expose each student to the main features of the SCRtool. To accomplish this, we designed the laboratory activity in which each student would be required to complete and validate a simple specification using those features. The example specification was for an embedded controller for an automatic Japanese-style bath tub with built-in heating and automatic filling and draining. The laboratory handout given to students described the system's purpose and intended behavior, enumerated the system's sensors and actuators, and then guided the students through the process of using SCRtool to complete and validate the specification. In order to give students examples to work from and to limit the amount of work needed to complete the activity, the laboratory materials included a partially-completed specification.

Monitored and controlled variables. SCR specifications model the world in terms of monitored variables, which represent input from sensors, and controlled variables, which represent output to actuators. Software is modeled as a collection of continuous and demand functions that take input from monitored variables and determine the value of controlled variables. In order to expose students to SCRtool's specification editor we asked them to add variable declarations to the partially-completed specification as needed to complete it. Since the laboratory handout listed all of the system's sensors and actuators, and since we included all of the needed variable types in the partially-complete specification, students needed only to add variables representing a small number of actuators.

Modes and mode transition tables. In order to simplify function description, SCR specifications declare mode classes, each of which divides system state into a number of modes. As monitored variables change, mode transition tables dictate the resulting effect upon the system's modes. The partially-complete specification included a mode class with all of the modes needed to describe the system and the incomplete mode transition table shown in Table 1. In the lab handout, we explained that in the SCR event notation the expression `@T(x)` specifies the moment when `x` becomes true, introduced the optional `WHEN y` guard clause that specifies the conditions under which the event causes the mode transition, and asked the students to use their knowledge of the system's intended functionality to complete the table.

Condition and event tables. SCR specifications model continuous and demand functions using condition and event tables, respectively. Each kind of table specifies the values taken on by one controlled variable under different modes. Rows in both tables

Table 1: Mode transition table provided to students

Source Mode	Events	Destination Mode
`Off`	`@T(PowerButton = Depressed)`	`Filling`
`Filling`	`@T(WaterLevel = Full) WHEN (Temperature < SetTemperature)`	`Heating`
`Filling`	`@T(WaterLevel = Full) WHEN (Temperature >= SetTemperature)`	`Ready`
`Heating`	*Left for students to complete.*	`Ready`
`Heating, Ready`	*Left for students to complete.*	`Filling`
`Ready`	*Left for students to complete.*	`ShuttingDown`
`Ready`	*Left for students to complete.*	`Heating`
`ShuttingDown`	*Left for students to complete.*	`Off`

represent modes, and columns in both table represent values. In condition tables cells contain logical tests of monitored variable values such that if a cell in the row representing the current mode evaluates to true, the controlled variable will have the value represented by that cell's column. In event tables cells contain event expressions, so that when the event in the row representing the current mode occurs, the controlled variable is assigned the value corresponding to the cell's column.

In the partially-complete specification, we included an example condition table and an example event table. Students were directed to examine these and then create tables defining the value of the remaining controlled variables.

Table 2: Sample event table provided for students

Modes	Conditions	
`Off, Filling, Heating, Ready, ShuttingDown`	`@T(UpButton = Depressed) WHEN (SetTemperature < 50)`	`@T(DownButton = Depressed) WHEN (SetTemperature > 25)`
`SetTemperature =`	**`SetTemperature + 1`**	**`SetTemperature - 1`**

Checking. SCRtool provides built-in checking for disjointness, coverage, and type correctness. Disjointness and coverage testing ensure that condition and event tables specify exactly one value for the variables they define under all circumstances. Type

checking catches syntax errors in table cells, initial conditions that are inconsistent with function definitions, uses of a variable inconsistent with its type, and the like. These checks are fully automatic, take only seconds, and can be started by clicking one tool-bar icon. We wanted to demonstrate the power and simplicity of these checks, so we asked students to check their specifications after they deemed them complete. We did not introduce any deliberate errors into the provided specification, but, since students made mistakes when completing the specification, they were able to see the checker's power and error-reporting mechanism in action.

Assertions. SCRtool permits specification developers to declare assertions representing properties that must hold at all times and provides an automatic theorem prover for proving these assertions. We asked students to write an assertion representing a safety property for the system and use the tool's built-in theorem prover to prove it. Theorem proving in SCRtool is a deliberately push-button activity, and we wanted to demonstrate how accessible it is to people unfamiliar with the underlying prover and its strategies.

Unfortunately the standard built-in prover was unable to prove the property we specified quickly enough for all students to be able to complete this part of the laboratory exercise. Because the prover took fifteen minutes or more, depending upon the exact formulation of the property and the speed of the computer, only one out of six groups managed to prove the property during the 75-minute laboratory period. Although our example's illustration of the runtime this prover requires is arguably instructive, were we to repeat this laboratory activity, we would chose an example safety property that could be proved by the default prover in seconds so that we could use our limited lab time to expose the students to as many aspects of the tool as possible.

Simulation. While students were waiting for the theorem prover, we demonstrated the use of SCRtool's simulator using an instructor's machine. Again, we wanted to demonstrate that the simulator is both automatic and simple. Clicking one tool-bar icon causes the system to generate and launch a basic simulator that allows the user to input monitored variable values and changes using standard dialog-box controls and observe the system's reactions.

Results of comparison discussion

After the laboratory activities, the two groups of students-those who chose model checkers and those who chose model-based development tools-each prepared a short presentation comparing and contrasting the tools. Each presentation was followed by a discussion involving the whole class. The presentation on model-based development tools raised the observations summarized below. We include this list of observations to illustrate the level of comprehension of the technical area, the tools, and their capabilities achieved by the students. The details of the observations are not especially significant.

The kind of software the tool is intended for. SCRtool is intended for the specification of embedded control systems. Perfect Developer and SCADE deliberately target safety-related systems with stringent dependability requirements. Simulink focuses on dynamic systems such as signal processing and communications systems.

The kind of developer the tool is aimed at. SCRtool is intended for use by software engineers with no particular discrete math or formal methods skills. Perfect Developer, in contrast, is aimed at software engineers with strong discrete math skills. Simulink and SCADE are intended for use by control engineers, not software engineers, and so describe systems using signal diagrams.

The tool's limitations. The present-generation SCRtool does not support any collection type such as an array, list, set, or map, and so cannot be used to specify systems whose state includes such a collection. (This capability may be added to a future release.) Perfect Developer requires the developer to write procedural code but offers no way to express concurrency or synchronization. Simulink lacks both a precisely defined semantics and theorem-proving capability. SCADE, like Perfect Developer, offers no way to express concurrency. None of these tools offers the ability to elegantly specify or analyze real-time behavior.

The guarantees made by the tool. SCRtool can guarantee that specifications are complete (in the sense that all tables are fully specified) and unambiguous, and can prove assertions. Perfect Developer and SCADE can guarantee that the implementation matches the specification. Simulink offers only simulation capabilities.

The V&V activities supported by the tool. All of the tools we studied support simulation. SCRtool and SCADE also offer proof of assertions. Simulink checks assertions during simulation.

The tool's code-generation capabilities. SCRtool generates Java code, but this code is intended for simulation rather than production use. Perfect Developer can generate C, Java or Ada code. Simulink can generate C or Ada. SCADE can generate C, Ada, and SPARK Ada.

The tool's usability. SCRtool's GUI needs refinement, but the specification model is simple. Perfect Developer has a much steeper learning curve and its documentation could be enhanced by a context-sensitive lookup feature. Simulink has an intuitive interface, good documentation, and the solid support of its makers. SCADE likewise has an intuitive user, although some students complained that the GUI was so cluttered as to leave little room for diagrams and that the program did not draw and flow wires cleanly.

The tool's scalability. Citing literature, students pointed out that SCR has been used on sizeable projects including thousands of tables. Perfect Developer can handle as much code as developers are willing to write. Simulink and SCADE allow functional blocks to be composed so that the size of a project is limited only by the system's available memory.

Conclusion

We have presented details of a course taught at the University of Virginia designed to introduce students to a variety of elements of formal methods by examination of

state-of-the-art tools. The approach used was to have each student in the class study a single tool, and for that student to: (1) present a lecture on the tool; (2) to conduct a laboratory exercise in the use of the tool; and (3) to create a final report analyzing the tool's strengths and weaknesses. The goal was to familiarize the students in the class with a variety of tools and to encourage them in critical thinking in the application of formal methods by asking them to assess the suitability of particular techniques.

While we have no sure way of telling to what extent the course met its goals, we feel that it was a success. The comments and course reviews from the members of the class were positive. The final reports were overall very high quality, and the comparisons of the different tools were particularly impressive. The classification scheme for dimensions of applicability of model-based development tools are described above; the classification scheme for the model checking tools was equally impressive. All of the courseware that was developed (presentations, laboratories, summary discussions, tool reports) are available from the authors.

Finally, we were pleased with the interest the students showed in teaching others. We presented the option to emphasize the laboratory activities to the students as a choice that they could make, telling them that we would reduce the work required for the final report somewhat if they chose to put effort into developing laboratory exercises. The feedback for this option was very strong, with no one objecting to it (other than expressing concerns about workload) and several students enthusiastically supporting it. We feel that there is significant potential for combining results from future similar courses, at the University of Virginia and perhaps at other universities, to start to develop a more comprehensive body of knowledge on how to teach the state of the art to senior undergraduates as an overall technology transfer initiative.

Acknowledgements

We thank Ralph Jeffords of the Naval Research Laboratory for his assistance in obtaining and installing SCRtool at the University of Virginia; and the other tool vendors—Escher Technologies, Esterel Technologies, iLogix, and the MathWorks—for allowing us to use their tools. We also thank all of the students who took this class for their efforts and support for the idea of trying this approach to introducing formal methods. We thank Kendra Schmid, Michael Spiegel, and Benjamin Taitelbaum in particular for their comments on SCADE, Perfect Developer, and Simulink, respectively.

References

1. "SLAM Project." Internet: http://research.microsoft.com/slam, [21 July 2006].
2. "BLAST." Internet: http://embedded.eecs.berkeley.edu/blast, [21 July 2006].
3. "Kronos Home Page." Internet: http://www-verimag.imag.fr/TEMPORISE/kronos, [21 July 2006].
4. "Spin - Formal Verification." Internet: http://spinroot.com/spin/whatispin.html, [14 July 2006].
5. "Automatic Construction of High Assurance Systems from Requirements Specifications." Internet: http://chacs.nrl.navy.mil/personnel/heitmeyer.html, [21 July 2006].

6. “Escher Technologies - Products.” Internet: http://www.eschertech.com/products/ index.php, [21 July 2006].
7. “SCADE Suite :: Products.” Internet: http://www.esterel-technologies.com/products/scade-suite/overview.html, [21 July 2006].
8. “The MathWorks - Simulink® - Simulation and Model-Based Design.” Internet: http://www.mathworks.com/products/simulink, [21 July 2006].
9. C. Heitmeyer. “Managing Complexity in Software Development with Formally Based Tools.” Electronic Notes in Theoretical Computer Science, vol. 108, 2004.
10. K. Heninger. “Specifying Software Requirements for Complex Systems: New Techniques and Their Application.” IEEE Transactions on Software Engineering, vol. SE-6, no. 1, January 1980.

A Playful Approach to Formal Models — A field report on teaching modeling fundamentals at middle school

Katharina Spies, Bernhard Schätz

Technical University Munich
Department of Computer Science IV
Boltzmannstr.3, D-85748 Garching (Munich)
Tel. +49/89 289-17800/17826
Fax +49/89 289-17307
spiesk | schaetz@in.tum.de

Abstract. Formal methods – besides supplying a set of generally mathematics-oriented formalisms – provide a collection of elementary concepts capturing the essential aspects of (software) systems independent of the chosen formalism. Getting a good intuition of these elementary concepts (e.g., variable, value, state, event) is a prime requisite to education in formal methods. The introduction of Informatics in German middle schools offers an opportunity to introduce these modeling concepts in a non-mathematical fashion to foster their early and intuitive understanding as a basis for further education.

Keywords: model-oriented approaches description techniques, modeling, teaching CS, specification, formal methods

1 Teaching CS Fundamentals at Middle School

Models have always been the prime tools of computer science. Traditionally, there have been two fractions in computer science dealing with them. On the one hand, software engineers have applied them due to their possibility to extract essential aspects from otherwise too complex systems. Modern modeling techniques cover these points in different flavours, especially with the UML in combination with colloquial descriptions; the fast adoption of these approaches is most likely due to their (assumed) understandability, often achieved at the price of impreciseness and informality. On the other hand, formal methods have traditionally applied models to obtain precise, analyzable and even verifiable descriptions of a system. Approaches like Z, CSP, or ASM address these aspects, together with increasing tool support for analysis; the reluctance to use these approaches is most likely due to a lacking familiarity of typical computer scientist with the underlying base concepts combined with the – nowadays decreasing – lack of convenient notations or editors.

To further acceptance of formal/precise models and make them accessible – in whatever lightweight form – the basic concepts of these models must be commonplace to computer scientists; understanding their advantages must be second nature to them. In the following we show how some basic concepts common to formal models can be

taught in middle school, establishing an early and intuitive understanding of these concepts prior to learning a mathematical apparatus to formalize these concepts. We also address illustrating the benefits of using these modeling concepts to 7^{th} graders.

To ensure sustainability, basic CS courses taught at schools should provide the pupils with universally valid as well as applicable CS knowledge. Pupils/students must rather acquire knowledge of typical CS models and methods than of specific tools, languages, or notations. Applied tools and software should be only used as means of demonstration; especially in middle school context, they should be used in a playful manner. Teaching typical concepts of a research field enables the children to transfer the acquired concepts and detect their counterparts in possible fields of application.

We present a field report on two half-year elective courses with 13-years old children held by the authors at the Werner-Heisenberg-Gymnasium (see [9] and [11]) at Garching (Munich, Germany). The approach was applied in four different groups of about 15 pupils (both male and female), over a course of two years. Classes were held over a period of half a year with about 12 sessions of 2 hours each. Classes were focused on immediate experience of the pupils, thus with more than 50% of the time spent on practicing the introduced concepts.

Although no formal evaluation and assessment of the effect of the taught classes was performed, some observations seem to be plausible and hold for all the classes performed: The principle of playful discovery of the introduced concepts, immediate feedback loops, and as little as possible up-front teaching proved a fruitful method of knowledge transfer, leading to a maximum of participation by the pupils (seemingly independent of the academic background of their parents) with little variation of the performed classes. Up to out experience, no significant difference could be found concerning speed of learning and depth of understanding between female and male pupils.

The construction of models proved to be a good method to let children experience typical basic concepts of CS. The curriculum was based on fundamental concepts like objects, states, events, or actions. In the following these concepts and their application in the class material is demonstrated. Modeling –combined with basic algorithmic concepts – helps setting a good foundation of CS methodology. Modeling was learned like a child's play.

The paper is organized as follows: first, we discuss the importance of modeling as core technique of CS and the possible advantages of understanding the basics of modeling in CS. Based on this, we use three examples based on the course material to explain the objective of the curriculum: learning *fundamental modeling concepts*, understanding the impact of using *models as abstraction and generalisation,* and learning to appraise advantage of *working with precise descriptions*. By applying these concepts in the context of object-oriented models, behavioural models, and algorithmic models, the exercises establish the children's ability to intuitively apply these concepts without requiring the mathematical apparatus to precisely define these concepts.

2 Models as the Language of Computer Science

On the one hand models form the product and the result of CS typical system development. On the other hand, applying models for the representation and construction of

systems and considers the activities, all transactions and the process of system development. By achieving a playful approach to (formal) models, an intuitive understanding of the possibilities of these models can be supplied at an early age:

1. Software development to a large extent is about the use of models. Each step is based on working with explicit or implicit models of the system under development or its environment. Learning to apply models on an intuitive level independently of more or less formal techniques supplies the necessary basics for a 'toolbox' of concepts needed in software engineering, which are often experienced to be quite challenging in a formal setting.
2. Even without supplying a formal definition, models are easily understood as having a precise meaning, allowing to assess whether a specification is valid. Furthermore, the formality of the language encourages greater rigor in the system specification. If 7th grade CS education gets across the understanding that exact and precise working leads to better results, an important step is done towards better CS products and processes, see e.g. [2] or [6].
3. Using models trains thinking in structures and hierarchies as well as understanding modularisation and the composition of systems, thus supporting the stepwise description of a complex system starting with a simplified model, going via several enhanced and more detailed models to the final models. By learning that models support a view-based description of a system ([6], [7] and [8]), children can experience the use of (formal) models to support the purpose-oriented definition of correct abstractions and descriptions of a system, allowing to focus on the necessary aspects rather than an formalism.

Therefore, the central objective of the approach presented here is:

1. The development of products and/or applications requires appropriate concepts depending on the intended purpose, covering all details and criteria relevant to the problem. (**Abstraction**)
2. Based on such models supply suitable and unambiguous descriptions. (**Specification**)
3. Descriptions and presentations created using those concepts support a systematic development process. (**Application**).

3 Course material: Some concrete exercises

The main topic of the course is teaching modeling approaches in general, and especially core concepts like object-orientation, behavioural modeling, and the description of algorithms. Based on simple but not over-simplified exercises children achieve an intuitive approach of thinking in models, using standard tools and techniques to present and describe their own ideas.

Office software suites are generally available at schools equipped with computers, supplying a simple tool set for constructing system descriptions. As a second advantage, children nowadays already generally are experienced with the use of office tools. Other similar programs can also be used for the following examples if they support graphical descriptions.

3.1 Specification by classes and objects

Using the geometric *auto forms* provided by office applications, the concept of specifications (and their models) as abstractions is introduced, as well as the concept of an attribute (or property) of a model. By introducing graphical models composed of geometrical modules (like triangle, rectangle or circle with attributes for each module (like fill or line color, position), the usefulness of specifications to precisely describe a specific model as well as a range of models is demonstrated.

By constructing graphical models (e.g., a TV set or a telephone) from those basic shapes, the pupils intuitively experience the concept of underspecified descriptions – corresponding to those basic shapes – defining classes allowing to generate possible instances – with specific colour, position, etc. – in a stereotypic fashion by fixing the attributes of those instances. By using a textual language representing classes, objects and attributes to describe instances of these basic shapes, and letting the pupils reconstructs a complex model according to a given textual description, the children experience the use of specifications to describe a specific model.

The acquired knowledge of structured/formalized specifications is used in a second step to clarify the benefit of precise and clear descriptions. This is done by playing a group game with switched roles of specifier and implementer. After designing a graphical model and specifying the model using the textual syntax in each group, the specifications are swapped between groups, and group is requested to redesign the picture by using the given specification. As in most cases, the original and the redesigned model do not complete correspond, this makes the pupils realize the impact of precise and clear. Having experienced the discrepancies with their own specification – without any influence of the instructor – the pupils gat a good understanding of the consequences of underspecified models.

3.2 Automata to describe components behavior

All kind of graphical models designed until this point give a good background to discuss several main aspects in modeling and specification like detecting and construction an abstraction of an actual system, the impact of precise descriptions, the use of given tools and modules, the structuring of systems, and the implications of using standard (CASE) tools. But the presented models focus primarily on static aspects and properties of the systems, like the arrangement of the components. This is certainly an important aspect in CS modeling because of the necessity to describe and specify e.g. the architecture of a system.

Obviously, however, the dynamic aspects like the behaviour are not covered by these means and must be modeled, described and specified otherwise. The limitations of the techniques introduced so far in the curriculum concerning the description of additional aspects a system can be experienced when discussing the functionality of a system specified only in terms of its (static) interface. The pupils are able to design the layout of the mobile phone with *auto forms* supplied by the office suite. But the question if this picture models a mobile phone or only the receiver of a conventional telephone can't be answered. The children learn that the picture is enough to represent a telephone but not

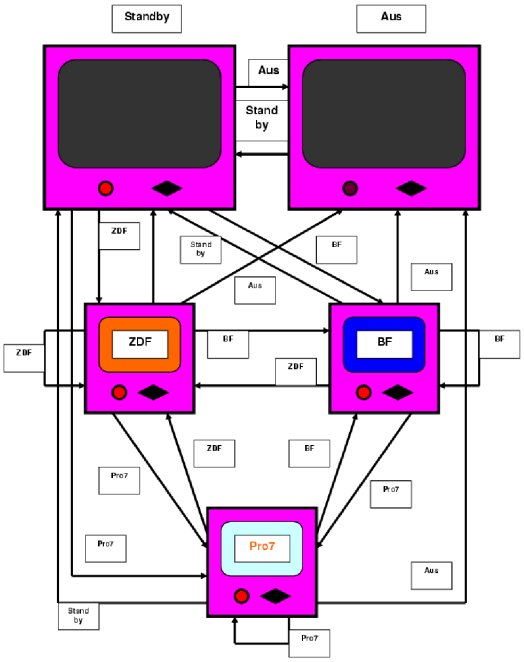


Fig. 1. A “Formal” Model of Behavior

enough to model the main characteristics of a mobile phone that is in fact given by its behavior.

To deepen the dichotomy of structural and behavioural description of a system, a second exercise is given by modeling a TV with 3 main stations, a standby and the out functionality, as shown in Figure 1. At first children are able to model the layout of the TV. Using this picture, the next step is designing five different TV to show its five states. Only some instructions allow the design of an automaton with five different TV pictures as states, connected via arrows, if there it's possible to reach that state from the other. The knowledge of the remote control and the appropriate switches the design with labelled edges is simple.

3.3 Algorithmic thinking

After working with the above topics, children are able to use objects and classes to model architectural structures and to give precise descriptions of systems, including its interactions performed via its interface. However, to describe behaviour on a detailed level, algorithmic description must be introduced as a third aspect. Although the topic in this course is modeling, the presented concepts allow the teacher to play a game that gives a first impression in giving instructions in a regular and goal-oriented way: jigsaw reading with a mouse that runs through a labyrinth. The children put characters in a wrong order into the given path. The mouse is able to run over the path in separate steps that are given by e.g. by the small boxes of the picture. The mouse is only able to run straightforward. If she should turn around or run the right or left way, the mouse must make a right or left 90° turn. If the mouse trespasses a character the character is read and no longer written at the path. As a first *program* children are now asked to describe mouse's run through the labyrinth to read the sentence or word and to give the teacher the task to recognize the intended word using children's algorithm. As an additional level of complexity, control structures – like conditional commands or repetitive execution – can be introduced to demonstrate the possibilities of parametric behaviour.

4 Experiences

Modeling enforces the ability to goal-oriented design, to find adequate structures, to work with formal and precise descriptions and to communicate by the use of models (to understand the background and more school environment, see e.g. [1], [3], [4], [5] or [10]). Children easily pick up the main activities involved in modeling: real objects must be represented by virtual objects, requiring to abstract from unnecessary details and finding suitable "formal" concepts to express the intended properties. Simulating the application of the real world problem means working with the model by an engine. Precise description of a model allows to examine properties of the model and to validate its usefulness.

In addition to the challenge to teach a common approved topic – CS – all facts and learned concepts should be recognized later on in further courses or even a CS study. All contents must be presented

- in a simple and comprehensible but not simplified way;

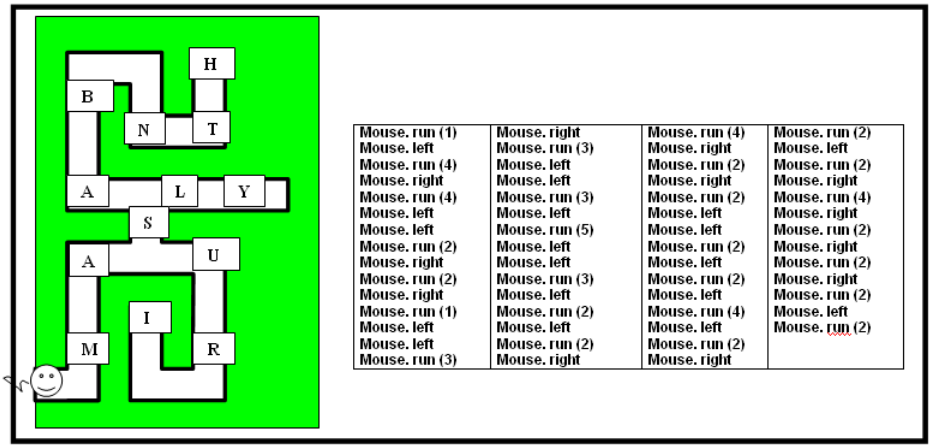


Fig. 2. Algorithms as a Game

- to obtain correctness based on typical CS concepts in general and to avoid falsification;
- to cover main aspects of the learned concepts and to understand their value based on CS;
- to help children discover the fundamental concepts found in formal approaches in a playful manner.

The presented course was taught in two half-year elective courses in the 7^{th} grade, without 15 pupils (both male and female, ages 12 to 13) per course. The experiences have shown that children in this age group easily understand the basic concepts presented in the sections above. With little training by smaller examples, they generally manage to apply the introduced techniques (e.g., specification by automaton-like notations) to new problems (e.g., the TV-control) within a session. Furthermore, by supplying simulation for the generated "specifications", they quickly embrace the possibility to use those concepts using a non-code oriented modeling paradigm. Finally, by exchanging "specifications" between different teams, children have a first-hand experience of the need to precise specifications during the validation of second-hand models.

While in this field-study office tools have been used as the means to convey the concepts, special open-source tools like LTSA [12] may even prove more useful if applied in the direction of putting the intuition above the formalization. In future classes, the use of such tools will be investigated more closely. Currently, additionally some simple tools are developed supporting specific aspects like the specification and simulation of simple programs (e.g., controlling a mouse like in Section 3.3).

Currently, in German curricula for middle schools computer science education is optional and therefore there is little tendency to install state-wide class material. Therefore, the class material presented here – including special adaptations of the applied

tools – is only updated and extended on a on-demand basis when holding those optional classes.

References

1. R. Baumann. *Wann sind zwei Objekte gleich?* LOG IN 124. pp 43-52. 2003
2. M. Broy. *Unifying Engineering Models and Theories of Distributed Software Systems*. Working Material Summer School Marktoberdorf. August 2002.
3. M. Broy, P. Hubwieser. *Ein neuer Ansatz in der Schulinformatik*. LOG IN 17. Heft 3/4. pp 42-44. 1997
4. W. Hartmann, J. Nievergelt. *Informatik und Bildung zwischen Wandel und Beständigkeit.* Informatik Spektrum 25. Heft 6. pp 465-476. Dezember 2002
5. P. Hubwieser. *Modellierung in der Schulinformatik*. LOG IN 19. Heft 1. pp 24-29. 1999
6. B. Schätz, K. Spies. *Modellierung im Software-Engineering*; *Grundlagen und Anwendung*. Lecture Notes. Lecture in Summer 2001. Technische Universität München. 2001.
7. B. Schätz, K. Spies. *Model-based Software Engineering.* ICSSEA 2002.
8. B. Schätz, A. Pretschner, F. Huber., J. Phillips. *Model based Development*. Technical Report. TUMI-0402. Technische Universität München. 2002.
9. K. Spies. *Informatik Kinder-leicht?! 1 Jahr Wahlkurs Informatik in der 7ten Klasse.* Jahresbericht Schuljahr 2002/2003. Werner-Heisenberg-Gymnasium Garching. Juli 2003.
10. A. Schwill *Computer Education Based on Fundamental Ideas*.
11. *www.whg-garching.de* Homepage Werner-Heisenberg-Gymnasium. Garching bei München.
12. J. Magee. J. Kramer. *Concurrency: State Models & Java Programs.* Wiley, 1999.

Teaching the Mathematics of Software Design

Emil Sekerinski

McMaster University, Hamilton, Ontario, Canada
emil@mcmaster.ca

Abstract. This note summarizes the experience and philosophy of teaching two one-semester courses, Software Design 1, a second year course, and Software Design 2, a third year course, repeatedly in the period from 1999/2000 to 2005/06. These courses had a peak enrollment of 190 students. Many students perceive these two courses as the core courses for their career in software development. The same material was presented in a condensed form in a graduate course in 2005/06. The courses taught students the mathematics of software design, rather than a particular "formal method" tool or language.

1 The Need for Software Design

Whereas in the design of a mechanical device breaking design rules would quickly lead to recognizable failure, one can very well break the rules of software design and still get a "sufficiently functional" and marketable product. Qualities of software are not as evident as qualities of physical products; *design qualities* are even harder to judge than the qualities evident from using a product. Students follow the rules of software design because they are told so and not because they would experience the consequences of not doing so. Students grow up with unreliable software to the extent that they consider such poorly working software to be normal or unavoidable. There is a widespread belief that programming skills are sufficient to write software. All this makes is difficult to convince students that software can be better designed, that it is worth doing so, and that it is worth learning the mathematics for doing so. We report on our experience teaching software design with its mathematical foundation.

2 Uniform Design Notation and Uniform Mathematical Basis

Our approach is to integrate mathematics in the presentation of software design, rather than to contrast formal and informal approaches; to teach all core topics in software design rather than a specific topic for which a dedicated formalism or tool exist; and to present the material with minimal notational burden.

A uniform textual design notation is used, in order to emphasize the similarities among the concepts and help students interconnect these concepts, rather than making students switch to a new mindset due to the notational differences. Graphical notations like flowcharts, class diagrams, and statecharts are presented as appropriate and defined by translation to the textual notation.

A mathematical basis for all design constructs is given. A typed logic using the same type system as a programming language is used. Equational reasoning is used for all proofs because of the familiarity from calculus.

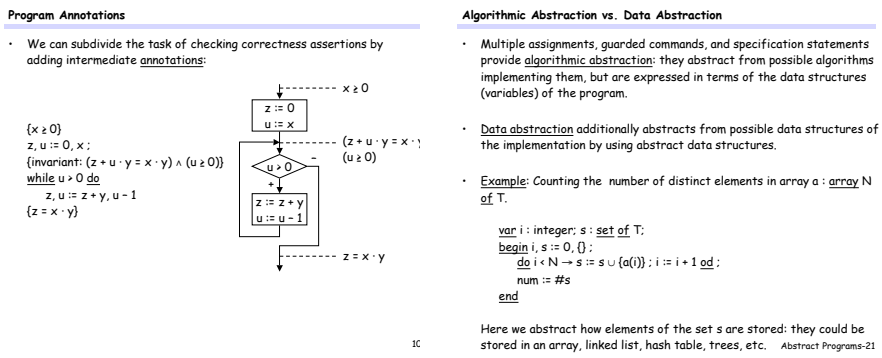


Fig. 1. Excerpts from *Elements of Programming* and *Abstract Programs*

3 Middle-out Sequencing of Topics

Courses on software design, or software engineering as they are called elsewhere, are typically structured according to the phases in which software is developed. However, students who have only written small programs so far, do not see the need for, say, elaborate requirements. Instead, we have started with writing and analyzing small programs and gradually moving to topics to which students become motivated, approximately in middle-out order of a normal software development, spread over two semesters:

1. *Elements of Programming* The course starts with an introduction to basic control structures in both textual and graphical notation, a formalization of syntax and typing, annotations, proofs of correctness using wp, machine limitations and partial expressions, principle of stepwise refinement, and further control structures. Searching and sorting examples are used throughout. Stepwise refinement is illustrated with the example of printing images [1]. The exercises are with paper-and-pencil only and force students to argue about programs without running and testing them.
2. *Program Modularization* The goals and principles of modularization are discussed and modules are introduced as a language construct. Module invariants and dependencies between modules are discussed formally. The KWIC example is used for illustrating the difference in qualities that arise from different modularizations [4]. The need for robustness of modules is discussed and defined formally. Exercises continue to practice formally reasoning about programs, but also show how to map modules to the constructs found in programming languages, in particular how encapsulation is enforced in common languages.
3. *Abstract Programs* Four means of abstraction are presented: multiple assignments, guarded commands, specification statements, and abstract data types. Guarded commands and specification statements are illustrated with common algorithms. The use of abstract data types is illustrated with algorithms familiar from other courses and with modeling information systems. The techniques are then used for the specification of modules. Assignments help to enforce the understanding of the concepts and practice the use of abstract data types.

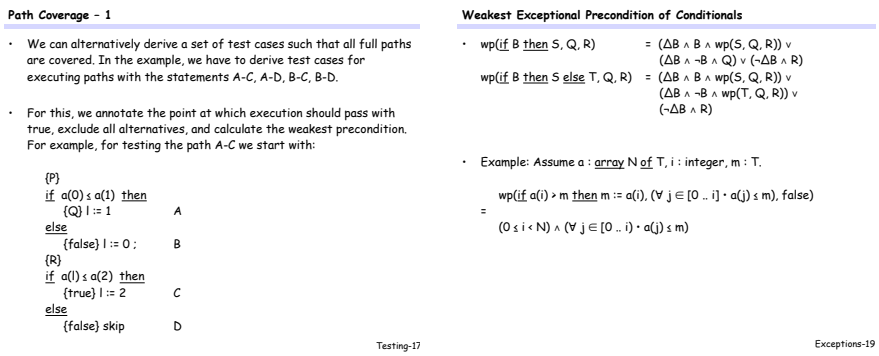


Fig. 2. Excerpts from *Testing* and *Exception Handling*

4. *Testing* The role and need for testing is discussed. Testing the internal consistency of modules is illustrated with checking module invariants. Specification based testing is used for both black box and white box testing. The *wp* calculus is used for deriving test cases to achieve various types of coverage. Test strategies are discussed. The assignments practice writing both implementations of modules and test suites according to formal specifications; in one assignment both implementations and test suites are run against (faulty) ones from other students: students experience the need for precise specifications.
5. *Exception Handling* Failures, the need for exception handling, and ways of reacting to exceptions are discussed. The raise and the try-catch statements, as the dominant exception mechanism, are introduced textually and graphically, and then defined formally through *wp*. Correct design of exception handlers is discussed and practiced with small examples.
6. *Functional Specifications* A formal model of programs in terms of relations is given and connected to *wp*. An alternative way of specifying programs by tabular relations is used to discuss the concepts of completeness of consistency of specifications [5]. Algorithmic refinement and data refinement are formally introduced.
7. *Object-Oriented Programs* Classes and inheritance are introduced textually and graphically. The object-oriented style is contrasted with the traditional style. A formal model is given in which class invariants and class refinement are studied. Class refinement is then used to discuss "good" and "bad" used of inheritance. Small exercises enforce the understanding of the formalism. Programming assignments then practice class design without formal proofs.
8. *Object-Oriented Modeling* Object-oriented models are presented as an alternative, graphical way of specifying data structures. Constraints like the multiplicity of associations are defined graphically and textually. The transition from an object-oriented model to an object-oriented implementation is illustrated.
9. *Requirements Analysis* The need for formulating requirements in the "user's world" is discussed. The step of delineating the context of a software system is discussed with use cases and use case diagrams. The notion of the interaction of a software system with its environment is motivated with sequence diagrams. Proving the con-

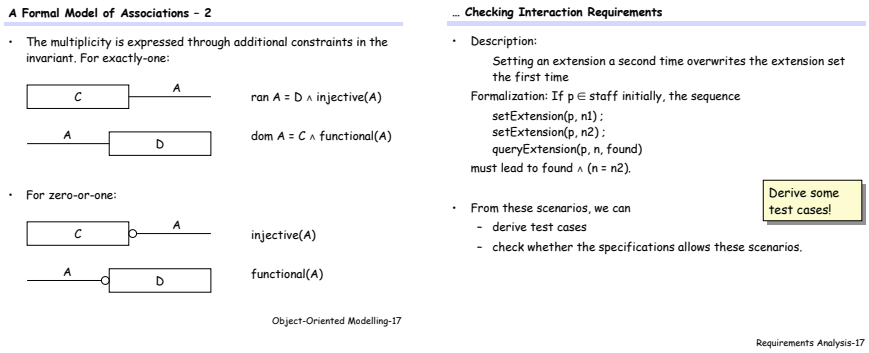


Fig. 3. Excerpts from *Object-Oriented Modeling* and *Requirements Analysis*

sistency of a specification with respect to sequence diagrams is discussed using *wp*, together with techniques for deriving test cases from sequence diagrams.

10. Object-Oriented Design Object-oriented techniques (idioms), design patterns, and frameworks are discussed, referring to class invariants and class refinement. Ten of the common 24 design patterns are selected. Assignments use the java.util framework for illustrating the use of design patterns and frameworks.

11. Reactive Programs The characteristics of reactive programs are contrasted with those of transformational programs. Statecharts are introduced as a dedicated formalism for reactive programs. A definition of statecharts in terms of guarded commands is given. Assignments practice the use of statecharts with a tool that allows statecharts to be animated and executed.

12. Software Development Process Different software development processes are mentioned, without going into detail (for time).

Additionally, the topic of *Configuration Management* was included at the beginning of Software Design 2 and subversion was used from that point on for all assignment submissions. Except for the topics of Configuration Management and Software Development Process, all other topics used a coherent notation and coherent mathematical basis. The assignments used Pascal, Java, jUnit (for testing), and iState (for statecharts); introduction to these was provided in optional tutorials.

4 Evaluation

We did not observe that students are in any sense math-phobic: they are sceptic towards the use of logic in software design as much as they are sceptic toward design patterns and configuration management systems; they haven't seen the need for any of these. In a series of assignments, the use of each concept of the course is practiced. At the end of the second course, students take the use of logic for granted. The topic that caused the most difficulties was object-oriented modeling; it was dropped in later editions of Software Design 2.

Requiring students to take a course in logic and discrete math before Software Design 1 had only a moderate effect on their ability to use logic and abstract data types for the description of problems. Our explanation is that logic and discrete math courses traditionally teach a body of knowledge, and do not practice the use for description and do not practice proving. Additionally, the difference in notation, as for implication and for quantification, prevents students from seeing the connection even if there is an obvious one; many operators, like relational overwrite, that are useful in software design are not taught in discrete math courses; usually (untyped) first-order logic is taught, rarely equational proofs. A good portion of Software Design 1 is spent—or rather wasted—with introducing notation for typed logic, data types, and equational proofs. One would wish that the field would have matured by now to standardize them.

Students are required to complete a two-semester design project in their fourth year. Software Design 2 was consistently ranked as the most useful course in a questionnaire at the end of the project, and Software Design 1 as the third most useful course. While that may sound encouraging, the projects rarely show a sufficiently systematic application of the techniques. That may be partly due to the explorative nature of these projects. However, the author believes that this is mainly due to these concepts not being repeated and practiced elsewhere. In courses on databases, operating systems, compilers, user interfaces, networks, real-time and algorithms terms like invariants and robustness are not used, giving the impression that these notions are not universally relevant. To give evidence to this claim, we refer to the analysis of the five most popular algorithm textbooks in [7]: four books, with 550 to 770 pages, devote zero pages on correctness and one book with 790 pages devotes eight on correctness. One would wish that textbooks and instructors would acknowledge the usefulness of these notions more widely.

Over the years, in the course evaluations 30%–65% of the students report that 81%–100% of the course material seems valuable and 35%–50% report that 61%–80% seems valuable. The numbers were on the higher end in later years and for Software Design 2 (not all students continue with Software Design 2). The use of independent critical judgement was rated high, particularly in later years. The overall delivery of the course received mixed evaluations, because the material was not fully developed in earlier years, the material was not motivated well in earlier years, most teaching assistants were of little help to the students, and because students felt overloaded. Except in the first year, there were no complaints that the contents is overly mathematical.

5 Discussion

We believe that we have successfully integrated the mathematics of software design into a two-semester courses in software design. The material is covered in 710 pages of lectures notes by the author plus a couple of original articles and book chapters (a course pack with the material is printed for students on demand). The mixture of mathematical and less mathematical topics gives students confidence that the use of mathematics is justified. We could not have done this with a single one-semester course.

If teaching the mathematics of software design is to be useful, it has to be taught as early as possible, before students acquire “bad habits,” a point that has been repeatedly

made; we wish we could have started even earlier in the curriculum. We have deliberately not used a specific formal tool; we find those more appropriate for upper level courses. Gordon [3] also offers a two-semester course, also using higher order logic as a unifying framework, but covers the logical aspects in more detail, and includes hardware verification. We have not tried to use any "light" method that makes formal techniques "invisible"; in our experience students appreciate being taught the theory in an isolated, minimal way, before seeing it applied with constraints.

Compared to the inverted curriculum, or outside-in order by Pedroni and Meyer [6], we do not teach class design before control structures, but we do teach programming before requirements analysis. Our way of introducing formal techniques is less gentle than theirs, but our reason is the same as for their outside-in order: to put all students on the same level and keep them motivated. We have succeeded with the first one, even if it comes by shocking the students in the first classes with mathematics, for which they were not prepared; we were less successful in keeping them motivated during Software Design 1.

While we believe that the courses influenced the way how students think about programs, the main obstacle for having a profound influence on their practice of programming is that concepts are not being repeated and practiced in other courses. We agree with Dijkstra's observation on computing science [2]:

> ... providing symbolic calculation as an alternative to human reasoning ... is sometimes met with opposition from all sorts of directions: ... 6. the educational business that feels that if it has to teach formal mathematics to CS students, it may as well close its schools.

If anything, with low enrollment numbers after the dot-com bubble burst, the pressure to eliminate mathematics has increased.

Acknowledgement. The author would like to thank David Parnas, Michael Soltys, and the two reviewers for their careful reading and thoughtful suggestions.

References

1. Edsger W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.
2. Edsger W. Dijkstra. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1398–1404, 1989.
3. Mike Gordon. *Specification and Verification, Parts I and II*. http://www.cl.cam.ac.uk/~mjcg/, 2006.
4. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
5. David L. Parnas. Tabular representation of relations. CRL Report 260, McMaster University, October 1992.
6. Michela Pedroni and Bertrand Meyer. The inverted curriculum in practice. In *SIGCSE Technical Symposium on Computer Science Education*, Houston, Texas, USA, 2006. ACM Press.
7. Allen B. Tucker, Charles F. Kelemen, and Kim B. Bruce. Our curriculum has become mathphobic! In *SIGCSE Technical Symposium on Computer Science Education*, Charlotte, North Carolina, USA, 2001. ACM Press.

Supporting Formal Method Teaching with Real-Life Protocols

Hugo Brakman, Vincent Driessen, Joseph Kavuma, Laura Nij Bijvank and Sander Vermolen

Radboud University Nijmegen

Abstract. In the world of computer science, formal methods play a primary role in the development of student minds and ability of abstract problem solving. Formal methods form the foremost technique that enables students to be trained in breaking complex questions up into abstract, manageable, pieces and to solve them using models that they themselves constructed. While the advantages of formal methods seem clear, students *en masse* tend not to attend the courses. This short paper analyses the core problems underlying this phenomenon and shows by example why the practical assignment that has been carried out by the five of us during a course in embedded systems, has contributed to the goal of formal method teaching.

1 Educational aspects

There is considerable evidence that the teaching of formal methods contributes largely to the student's understanding of problems, abstract reasoning skills, and their ability of elegant problem solving. However, these skills are not taught at once, but are part of a large educational programme, much like how reasoning skills are taught in high school. The consequence is that the yield of this kind of teaching is settled in the long term.

Students on the other hand, tend to have a more narrow focus, valuing their study progress by looking at short term efforts and results. Important roles in their course selection are the so called "fun-factor" and the amount of (applicable) skills or knowledge that will be gained by following the course. As a typical example, a student would preferrably follow a web programming course than a course in formal methods.

Among the problems that formal method teaching faces currently, we may distinguish the following:

"What's the use?" Students simply do not see the practical use of "all this math". This is the main problem for formal method teachers to face. This problem incorporates a few consequences.

First and foremost, motivation for students drops significantly if they must do things when they have no clue about what they will gain from doing it. Telling them that they will learn "abstract reasoning skills" will not work either, because that sounds much too vague. There is a big necessity for rolling out a clear roadmap with targets in the student's curriculum and sticking with that map in order to keep them motivated.

Furthermore, there is competition from other fields of science. Especially nowadays, with the computer and IT-business luring people from universities, students foresee a better future by pursuing a practical master degree. If universities do not succeed in waking up the interest in formal methods, computer science may start getting loose from its (mathematical) roots.

Lack of visualization Considering the problems *within* the formal method courses, lack of visualization is the first one. Teaching plain dry mathematics is not only dull, it also is not the most effective route to the mind of the students. The human brain learns through association. Cognitive research has already shown that education through visualization enables a better understanding of matter, because mental models are constructed intuitively and more solid.

This would also explain the apparent resistance to these courses. An aura of complexity, mathematical sophistication and unfamiliarity surrounds formal methods courses. This aura is kept alive by the thought that insights are something that someone has from birth, and cannot be learned to a high degree. However, insights can be *enabled*, not only gained. Repetitively letting students "solve equations" narrows their focus and does not connect to the goal of what they are doing it for. Illustrative material will help their brains to enable insights through visualization, much like how a picture says more than a thousand words.

Application of knowledge The last problem is a generally known one, which occurs at the moment the students have gained the desired knowledge on formal methods. Since generalization is one of the most important focusses of scientific teaching, the connection to real world problems might be lost. A dedicated task of formal method teaching should be the establishment of a link between theory and practice, not the practice itself.

When enabling insights, teachers must beware that these insights are not isolated in the minds of their students, but are made accessible. One way of doing this, is to make sure that visualized concepts are being linked to situations where that type of knowledge is applicable, if possible. This effort would especially avoid freshmen developing an anxiety for mathematics and would also contribute to help students see the use of formal methods.

2 Solution

The solution outline we propose and which we can recommend through experience is the following:

Take a real-life problem and a known solution to this, or a sub solution of it. Examples are known network protocols such as Ethernet or as in our case Bluetooth. Let the students make an abstract model of this, that can be used as input to a known model checker. The next obvious step is then to let them use this model checker to verify properties of the model, that together prove the correctness of the protocol.

We will now look into our project more specifically, which will follow the outline presented above.

The purpose of our project was to do a formal verification of part of the Bluetooth protocol. This wireless network protocol consists of several relatively separate phases, which gave us the possibility to examine part of it and still be able to draw very useful conclusions. We have examined the inquiry part. This is usually the first phase in a Bluetooth communication. The purpose of this phase is to discover other devices and find out some basic facts about them.

How we would tackle the problem was mainly up to us. We have decided to alternate individual study with discussions of the problems we had encountered. In case there was a problem we were not able to solve ourselves, we could always ask for assistance. But in practice, because of the setup we have used, we appeared to be able to solve most issues ourselves.

We have split up the content of the project in three parts: investigation of the protocol, modeling the protocol and finally using our model to prove several theorems about Bluetooth communication.

Investigating a protocol description does not appear to be something one does for fun. At first hand, we shared this opinion. However the crucial issue appeared to be in how to investigate the protocol. Doing a broad and superficial survey can be hard and will usually result in forgetting it some days later. In contrast to this, we have investigated part of the protocol in very much detail and made a very narrow survey. This did not only limit the number of pages we had to read, but it also gave us the possibility to understand and check most of the details of the protocol.

We had divided the Inquiry phase among the five of us, discussing all parts one of us might not have understood. Obviously, this resulted in a good understanding of the protocol, but it also resulted in finding some unclarities and some possible inconsistencies in the protocol document. Which makes investigating it quite satisfactory.

Having gained enough knowledge of the protocol, we could move on to the next phase of our project: modeling Bluetooth. The model checker of our choice was UPPAAL. UPPAAL was able to give us a good interface and a useful model checker. But in contrast to many of the other tools we have seen so far, UPPAAL also provided us with a very useful simulator in which one can view and understand traces and use these to further correct the model, or to draw conclusions.

Now, understanding the protocol was one thing, modeling, an entire other. Some of the problems we had heard in theory appeared to be obstructing the design of our model more than we had expected. And especially many of the idealistically stated solutions appeared to have some unforeseen drawbacks. Nevertheless, given some time, we were able to find the solutions to most of these problems, learning many ins and outs of the used theories on our way. Mainly the fact that we have solved the issues ourselves gave us great insights in what the 'idealistically stated solutions' were really about.

The last part of the project consisted of using our model to verify correctness properties of the Bluetooth protocol. Unfortunately this took UPPAAL a lot more time than we had accounted for. The only solution was to abstract our model from some of the hardware details, to make its complexity suitable enough to do verifications of some of the properties we had come up with. But the difficulty was to keep it realistic enough to still be able to draw conclusions about Bluetooth in real-life. Our effort in trying to reduce the complexity eventually resulted in a model that was only slightly simplified, but well suitable to verify our properties. At this point we were able to verify our correctness properties of this part of the Bluetooth protocol. Which we consider to be a very satisfactory result of the project. It is not something that has not been done before in this way and yet proved to be very useful.

3 Project report

To give a more illustrative explanation of what we have done, we have included three (slightly altered) sections of our report that resulted from the project. These will give an introduction to the topic, an informal description of part of Bluetooth we have modeled and the conclusions we have drawn using our model. A more detailed description can be found in this report, which can be found on our webpage [3].

3.1 Introduction to the topic

Bluetooth is a widely used communication protocol these days. It is used in the communication between phones, computers, headsets and many more devices. In the year 1994 the Ericsson company decided it wanted a protocol that could be used to connect mobile phones to other devices. Jaap Haartsen, working for Ericsson, developed the protocol. The techniques were further developed by the Bluetooth Special Interest Group.

One of the things described in the protocol is the way in which two devices that are neither connected nor synchronized can try to find each other. This is called the Inquiry Response Phase, the first phase in the protocol, which should provide a way for the devices to synchronize in order to allow further communication.

We have looked closer at the specification of this phase as described in [2] and created an UPPAAL model to formally verify that after the Inquiry Response Phase indeed the devices will be synchronized.

3.2 Informal description

When two Bluetooth devices want to start communicating they do that using the Inquiry Phases. In these phases one of the devices is assumed to be in master mode querying for other devices. The other device is assumed to be in slave mode. The master keeps sending packages and listening for responses. The slave will listen for a package from the master and respond to the master by sending a return package.

The devices do, however, change frequency during every phase. The frequencies used in Bluetooth are very common frequencies used in wireless phones, remote controls,

garage doors and more. Therefore the devices change ("hop") their frequencies a lot. The devices are unlikely to use the same frequency the first time and the communication attempt will fail. However, the hopping should be done in a way that at a certain point in time the devices will use the same frequency in the same time interval and further synchronization can be achieved using that.

There are quite a few tricks involved in order to get this to work properly. There is the hopping of frequencies, timing issues in sending, receiving and listening and some more.

We want to verify that indeed the devices will eventually synchronize in all cases if we follow the specification. To do this we have constructed a UPPAAL model that represents the relevant bluetooth phases. As illustration we have added the part of this model that is responsible for the device execution. This can be found in figure 1.

3.3 Results and conclusions

We have created a model of which we think is sufficiently close to reality to be used in the verification of some properties of the Inquiry Response Phase.

Modeling the Inquiry Response Phase in UPPAAL worked rather well. It gave us good insight in the phase and some questions surfaced that we could not answer easily. We have even found a strange remark in the specification that to our opinion is incorrect.

Although it is arguable whether the specification is well written, at least we could, with some effort, all agree on what we think the specification specifies.

We verified that always eventually the master device will receive a return packet from the slave for a lot of initial values. This means we have a strong belief that two Bluetooth devices will eventually synchronize.

The UPPAAL model we have created can be downloaded from
`www.cs.ru.nl/ita/publications/papers/fvaan/bluetooth/` The website of the UPPAAL project is
`http://www.uppaal.com`

Verification Results In total, we have tried to prove two properties of the system, representing system liveness and safety. These are:

- $A\Diamond$ `Master.Finished` $\wedge$ `Slave.Finished`
  This property actually expresses that the system always eventually will reach the "finished" state for both devices, i.e. it expresses that always eventually the master device will receive a return packet from the slave. Actually, this property is the desired property the developers of Bluetooth would want to satisfy under all conditions. We have validated this important property for a whole variety of initial clock values. Besides that, we have been able to verify these properties, too, for both ideal Bluetooth clocks as well as clocks that were subject to drift and jitter.
  Some of the validated configurations:

maxwaitbit	deviation (UPPAAL time units)	time (min)
4	1	1
7	1	3
7	3	8

The `maxwaitbit` indicates that when bit number `maxwaitbit` of the Bluetooth clock is or becomes 1 the devices must enter the Bluetooth phase instead of waiting at the initial state. This way we can check a range of initial clock values. Ideally we should verify this for maxwaitbit being the maximum clock bit. But that simply takes too long.
The `deviation` indicates that, for each period of 625 UPPAAL time units, the clock tick may differ this amount of UPPAAL time units. A deviation of 3 indicates therefore a deviation of $\frac{3}{625}$ UPPAAL time units, indicating a maximum deviation of about 7 minutes a day. In practice the clocks used will not be that bad, therefore this means in practice that synchronization is accomplished. We could have used a tighter interval but apart from resources required for the verification this would not affect the result.

– $A\Box$ `not deadlock`
This property actually expresses a system invariant, stating that the system as a whole will never deadlock. This property is very important in getting a confidence that the specification is correctly modeled. Of course if the system can always reach the state `Finished` for all devices, it cannot have deadlocked.
Some of the validated configurations:

maxwaitbit	deviation (UPPAAL time units)	time (min)
0	0	1
2	0	1
4	0	1
7	0	1

With these properties satisfied and no counter examples found we have a strong belief that in our model these properties actually hold for all initial values. Therefore we think in the Bluetooth protocol the devices will also find each other eventually.

4 Results

4.1 How our approach solved the problem

The idea of verifying a real world standard protocol instead of a textbook example is a motivation and an opportunity in the sense that it helps students recognise the values of the teaching in an applied system.

Graphical visualisation, modelling and simulation of a problem, help to understand formal methods through giving different dimensions of the formal methods other than the theoretical formulae.

The almost social atmosphere created by working in relatively small groups when dealing with a problem, enhances the learning process by creating an environment that allows the weak students to learn from the stronger ones. Sometimes a student may not feel free to participate actively in class but when in a small group he/she will feel more at ease to ask even the dumbest question.

A tool like UPPAAL that supports a versatile range of programming language formats, also is a solution in a way because as students try to reflect on their understanding of the formal methods in a piece of code in a less restricted environment, they achieve a deeper understanding of the methods and probably begin to like the exercise. Besides being a rich tool, UPPAAL also reduces the limitions that in many tools are enforced due to poor representation of automata states or boolean conditions.

Further Research and related work Probably we could verify more situations than the ones we did verify thus far. So another group could work on that.

One of the things we didn't pay much attention to is the duration of a transmission in the Inquiry Response Phase. Currently we assume that a transmission, if the receiver listens in time, will arrive completely and without errors or not arrive at all. Time does not elapse while sending or receiving. In reality this is not the case and it might be something to take a closer look at.

Something else we did not look at, but probably will be relatively easy to do using the model is verify properties for more than two instances (master, slave, slave for example).

Our research focused on the Inquiry Response Phase, leaving out other phases. Obviously these could be interesting.

Closely related work can be found in the paper [4] where the probabilistic tool PRISM was used to analyze the Inquiry Phase. [4] focuses on the probabilistic behavior of the Bluetooth communication. Where this paper mainly looks at the expected and worst and best case timing issues, our research mainly focuses on whether the communication will actually be succesful or not.

5 Knowledge and Skills

5.1 Required knowledge and skills

Team work skills Most real world problems (protocols) are systems enormous in size. For students to profitably work on such systems there is a need for teams/groups thus calling for team work skills as an essential tool for this approach.

Mathematical background It requires a relatively good background of mathematics to a certain detail, to visualise and understand formal methods, and in relation to this, students may also need to be familiar with transition systems. And most importantly

they should have a good knowledge of the tool being used for the simulation and modelling. All of this also requires basic programming/modelling skills/knowledge for the students to be able to reflect their understanding of the formal methods into modal code that leads the simulations.

5.2 Desired skills and knowledge outcomes

From the objectives of the course [1] we can select the objectives relevant for this assignment, which will be:

1. Being able to recognize situations in which the applications of formal methods for specification and verification may be useful.
2. Being able to model distributed algorithms and protocols (or more generally: reactive systems) as networks of automata.
3. Being able to formalize desired properties of these algorithms and protocols in terms of automata or temporal logic.
4. Being able to use state-of-the-art proof techniques and computer tools for the analysis of embedded systems and protocols of ”average” complexity.

5.3 Outcome skills and knowledge

Let us finally check whether each objective in section 5.2 is met:

1. We are able to recognize some situations in which the applications of formal methods for specification and verification may be useful.
2. We are able to model a part of the Bluetooth protocol as networks of automata.
3. We are able to formalize desired properties of the Bluetooth protocol in terms of automata or temporal logic.
4. We are able to use UPPAAL for the analysis of a part of the Bluetooth protocol.

So we well meet the objectives for this assignment for the bluetooth protocol, which wasn't a very specific one. So we are probably well able to do the same project for other protocols.

References

1. Analysis of embedded systems. URL `http://www.cs.ru.nl/~fvaan/PV/`.
2. Baseband Specification. In *Bluetooth–Core Specification v2.0 + EDR*, pages 55–210. 2004. URL `http://bluetooth.com/NR/rdonlyres/1F6469BA-6AE7-42B6-B5A1-65148B9DB238/840/Core_v210_EDR.zip`.
3. Supporting formal method teaching with real-life protocols website. URL `www.cs.ru.nl/ita/publications/papers/fvaan/bluetooth/`.
4. M. Duflot, M. Kwiatkowska, G. Norman, and D. Parker. A Formal Analysis of Bluetooth Device Discovery. In *1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04).*, November 2004.

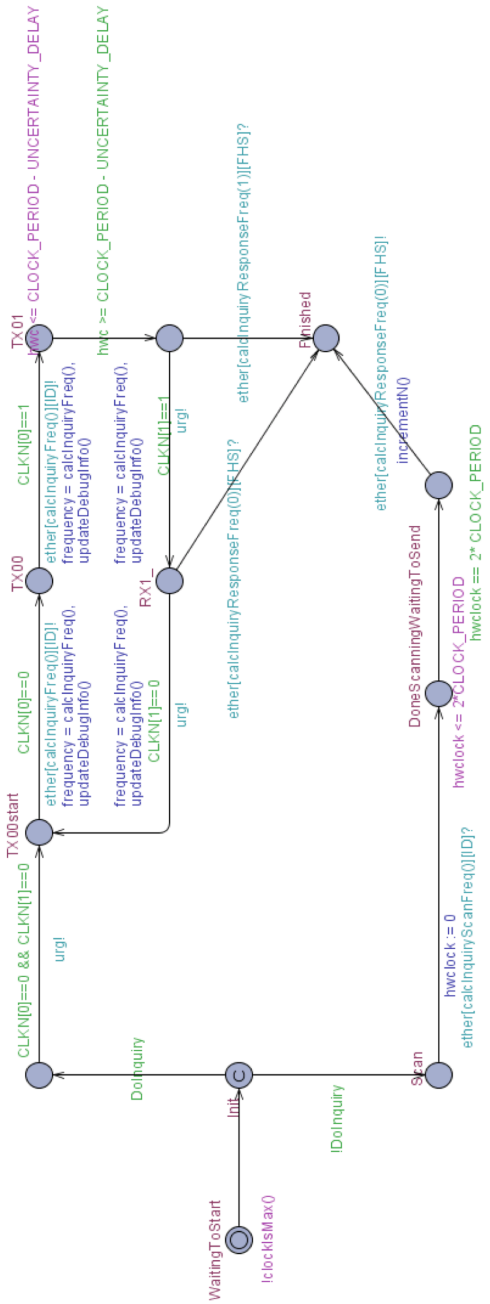


Fig. 1. The full Device template as modeled in UPPAAL

From Design by Contract to Static Analysis of Java Programs: A Teaching Approach*

Christelle Scharff[1] and Sokharith Sok[1,2]

[1] Ivan G. Seidenberg School of Computer Science and Information Systems, Pace University, One Pace Plaza, New York, NY 10038, USA. Emails: cscharff@pace.edu, sokharith.sok@pace.edu

[2] Institute of Technology of Cambodia, BP 86, Blvd Pochentong, Phnom Penh, Cambodia. Email: sokharith.sok@itc.edu.kh.

Abstract. We describe an approach to teach students program correctness as an activity interleaved with the programming process. This approach is based on teaching design by contract followed by Hoare Logic supported by the ICS little engine of proof software. It implies covering redundantly the same examples to enforce the understanding, differences and advantages of both methodologies. Reactions of the students are presented. We also briefly discuss the need of integrated, supported, documented, scalable and usable program correctness tools to spread their use in education and industry.

1 Introduction

Tony Hoare defined the concept of program correctness in the sixties and encouraged developers to think about programs as mathematical objects that can be understood through logic [1,2]. In the seventies, Leslie Lamport wrote a short note entitled "How to tell a Program from an Automobile", where he advocated a lightweight approach to correctness [9]. Programmers should be able to answer two fundamental questions about their program: (a) What is the meaning of the program they developed? (b) Can it be proved that it has the correct meaning? Programmers are not required to write the proofs but should be provided with tools that would help them guarantee that their programs meet given specifications.

Program correctness is generally described with Hoare logic [1,2], a formal system that "provides a set of logical rules in order to reason about the correctness of computer programs with the rigour of mathematical logic" [3]. The central feature of Hoare logic is the Hoare triple $\{P\}\ C\ \{Q\}$. P is a pre-condition, and Q is a post-condition. P and Q are assertions, which are formulae written in predicate logic and claimed to be true during program runtime execution. C is a sequence of statements of the program. $\{P\}\ C\ \{Q\}$ describes how the execution of a piece of code changes the state of the computation. Hoare logic has axioms and inference rules for all the common constructs of modern programming languages (e.g. sequence, assignment, conditional and loop statements).

* This work is supported by the National Science Foundation under grant ITR-0326540.

Design by contract [11] was developed by Bertrand Meyer in the nineties and is built on Hoare Logic; contracts (functional specifications) annotate the program and are checked during program execution. It was first implemented in the Eiffel programming language [12] and is now a common feature in modern programming languages including Java 1.5 through an assertion facility. Agile Java [10] focuses on test-driven development and emphasizes writing assertions, as testing artifacts, before writing code.

As a programming practice, design by contract is accepted in the industry and is more accessible to students. The concept of program correctness is generally taught as an activity independent of the programming process [5]. In this paper, we describe an approach focusing on teaching design by contract followed by program correctness of Java programs using Hoare Logic and ICS (Integrated Canonizer and Solver) [http://www.icansolve.com] [4], a little engine of proof [13] deciding formulae in combinations of theories (including linear arithmetics and arrays). The formal static analysis process is not automated but taught in combination with programming; it implies working directly on the code. This approach implies covering redundantly the same examples to foster the understanding, differences and advantages of both methodologies, and the development of a repository of examples that will be available at: http://www.csis.pace.edu/∼scharff/LEP. We tested our approach in a programming languages and implementation course in fall 2005 at Pace University.

This paper is organized in the following way. We describe our teaching approach in section 2 with an example in section 3. We present students' reactions in section 4. In section 5 we discuss the need of integrated, supported, documented, scalable and usable program correctness tools to widespread their use in education and industry.

2 Approach

Our light-weight approach to teaching program correctness is based on teaching design by contract, followed by Hoare Logic supported by the ICS little engine of proof software. The same illustrating examples are covered redundantly when talking about run-time and formal static analysis of programs.

When tackling a particular programming problem, students are encouraged to think about the normal courses, alternative courses, exceptions, error cases and boundary conditions of the code they have to develop. They have to determine conditions on the parameters and returned result (if applicable). For example, the *factorial* function takes an integer greater than 0 as a parameter, and returns an integer only if its parameter is smaller than 13.

Students determine the pre-conditions, post-conditions, and loop (and class) invariants. They also establish diverse properties on the program fragments they have to write. The discovery of the assertions is a very difficult activity.

Following the design by contract approach, students write the corresponding code and assertions in Java 1.5 in the Eclipse integrated development environment [http://www.eclipse.org]. Pre-conditions are written by throwing exceptions in cases they are not verified. Post-

conditions and loop invariants are written by annotating the code with Java *assert* statements[3].

Once familiar with design by contract, students are initiated to static analysis of Java code as it is implemented in Eclipse; Eclipse detects (statically) some common errors in code (e.g. assignment with no effect, unused variables and methods, non static access to a static member). The goal of this phase is to enforce students' understanding of what static code analysis is and its scope.

Students are then introduced to correctness with Hoare Logic, its differences and advantages (e.g. for critical applications and long-compilation/execution time programs) with respect to design by contract. Writing pre-conditions, post-conditions and invariants requires the students to be familiar with predicate logic. The syntax to write the assertions also depends on the tool that is used to check their validity. For each code fragment with a pre-condition and post-condition, the post-condition is propagated upward through the fragment and newly generated assertions annotate the code. In particular there is a generated pre-condition. This phase is currently not assisted by a tool; students are doing the work on the Java code by hand.

We choose to check the validity of the assertions using the ICS little engine of proof software. ICS [4] is a software developed by Harald Ruess at Stanford Research Institute that efficiently decides quantifier-free formulae in combinations of theories (uninterpreted functions, linear arithmetic, non-linear arithmetic, products, co-products, arrays, fixed-sized bitvectors, propositional sets, and functional abstraction and application). ICS is a Linux command line tool that can be used interactively, in batch mode, or can be accessed in server mode through APIs in C, Lisp, Ocaml and Fortran. ICS has the ability to provide a model or counter-example. For example, in ICS, $x > 1$ & $x > 3$ has $x > 3$ as a model and $x <= 3$ as a counter-example. $i = j$ & $k = l$ & $a[i] = b[k]$ & $j = a[j]$ & $m = b[l]$ & $\sim a[m] = b[k]$ is evaluated to unsatisfiable[4].

In this framework, proving the correctness of a program or program fragment corresponds to proving that if the pre-condition stands before the execution of the code, then the post-condition stands after the execution of the code. This is equivalent to proving that the initial pre-condition implies the generated pre-condition [6]. Proving that $p \rightarrow q$ is valid using ICS is done by (refutationally) proving that $\sim p \rightarrow q$ (or equivalently p & $\sim q$) is unsatisfiable.

3 Example

In this section we consider a "toy" example to illustrate our approach. We want to prove *P*: if we start with x equals to y and execute $x = x + 1$ followed by $y = y + 1$, then x is still equal to y.

The annotated Java code we consider is the following:

[3] Please note that Java 1.5 does not support class invariant. JMSAssert [htpp://www.mmsindia.com/DBCForJava.html] is a tool that brings the class invariant benefits to Java.

[4] In ICS, & represents the conjunction operator and $\sim$ is the negation operator.

```
// Pre-condition
if (x != y){
    throw new illegalArgumentException("x must be equal to y")
}
x = x + 1;
y = y + 1;
// Post-condition
assert (x == y): "After execution of the program X = "
+ x + " must be equal to Y = " + y + ".";
```

When propagating the post-condition upward, we obtain the following annotated code[5]:

```
[x = y]   (Pre-condition)
[x + 1 = y + 1]   (Generated pre-condition)
x = x + 1;
[x = y + 1]       (Generated assertion)
y = y + 1;
[x = y]   (Post-condition)
```

In order to prove P, we must prove that: $x = y \rightarrow x+1 = y+1$. According to the ICS syntax, this is done by proving that: $\sim (x = y \rightarrow x+1 = y+1)$ i.e. $x = y \ \& \ \sim x+1 = y+1$ is unsatisfiable:

```
ics>  sat x=y & ~x+1 =y+1.
:unsat
```

4 Discussion

We used the approach described in section 2 in a junior course of *Programming Languages and Implementation* of fifteen students in fall 2005 at Pace University. This course covers the functional, logical, imperative and object-oriented programming paradigm varieties, program correctness and the compilation process emphasized by the implementation of a scanner and parser. Program correctness was covered in class during four sessions of two hours. Prior to this course students took Programming I & II, Data Structures I & II and Discrete Mathematics. At the end of the semester, students were assessed on their perception and understanding of run-time and formal static analysis through a homework assignment that asked them to apply both methodologies on a specific example (e.g. stack or sorting) and included the following essay questions:

- As a developer, what do you think are the difficulties with annotating programs with assertions that are checked at run-time?
- As a developer, what do you think are the difficulties of formal static analysis of programs?

[5] Annotations are represented between brackets.

- What would you be more willing to use as a developer: design by contract or formal static analysis? Why?
- In formal static analysis of programs, what do you think are the phases that are the responsibility of the developer and what should be automated?

The first obvious difficulty for the students was to come up with the (right) pre-conditions, post-conditions and, in particular, (loop) invariants. However, students believed that considering pre-conditions, post-conditions and invariants permitted them to ”think” more about the behavior of their code and improve its quality.

All students thought that using Java *assert* statements in the code was easier than the formal static analysis process that required fluency in logic. Their perception was motivated by the fact that design by contract was closer to programming. Assertions were perceived as a means of code documentation and testing. When getting a failed assertion, students examined if the problem had occurred in the program itself or in the assertions, but found the task cumbersome. They mentioned the lack of automation of the static analysis process (especially for the propagation of the post-conditions) and the lack of integration of the process in one tool as a drawback to its wide use by developers. Students could not envision how the static analysis process would scale to prove large programs.

Students had to practice with the ICS research tool and learn its specification and proof language. They felt that the tool would have been more accessible with a dedicated graphical user interface and documentation at their level of understanding. It was difficult for the students to understand the scope of ICS - what it can prove, what it cannot prove and how to express statements to be proved in the ICS-specific language. Tools are crucial for the use of formal static analysis in education and industry.

5 Further Work

In the 2005 IFIP Working Conference on Verified Software: Theories, Tools, Experiments, researchers identified tools as an important enabler to making verified software a reality in the next fifteen to twenty years [7]. In the April 2006 IEEE Computer Magazine article ”Verified Software: A Grand Challenge”, C. Jones, P. O’Hearn and J. Woodcock stated that ”Given the right tools, the use of formal methods could become widespread and transform software engineering” [8]. Such an endeavor would require the community that builds software verification tools to first apply the tool on the tool itself and produce integrated, supported, documented, scalable, usable, and multi-user verification tools. Some popular tools are : ESC/Java2 (Extended Static Checker for Java)[http://www.cs.virginia.edu/cs201j/plugin, http://secure.ucd.ie/products /opensource/ESCJava2] based on JML (Java Modeling Language) [http://www. cs. iastate.edu/ leavens/JML], a language to specify modules behavior, and Daikon [http://pag.csail.mit.edu/daikon], a dynamic invariant detector that reports likely program invariants.

We believe that the same tools should be used in education and industry. There is revived interest in the theorem proving community concerning tools as demonstrated by the UITP (User Interface for Theorem Provers) conference gathering people from the theorem proving and human computer interaction communities.

To support our approach, we are currently developing an Eclipse plug-in for ICS. We envision that this plug-in will provide a rich editor for inputting specification and proof commands following the tool syntax and a series of views for results (satisfiable, unsatisfiable, counter-examples), problems (syntax errors with feedback), proof elements (states, proofs, redo/undo of proof steps) and documentation (tool-related, syntax, semantics, proof). Our long-term goal is to develop an Eclipse plug-in for formal static analysis of Java programs based on Hoare Logic and ICS. We believe that using Eclipse as an environment to support the static analysis process would benefit education and industry. We are also developing a repository of examples based on our approach that will be available at: http://www.csis.pace.edu/∼scharff/LEP.

References

1. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.
2. C. A. R. Hoare. Proof of a Program: Find. *Communications of the ACM*, 14(1):39–45, January 1971.
3. Hoare Logic. Wikipedia. Available at: http://en.wikipedia.org/wiki/Hoare_logic.
4. J. C. Filliâtre, S. Owre, H. Rueß and N. Shankar. ICS: Integrated Canonization and Solving. *Proceedings of the Computer-Aided Verification Conference, CAV '2001*, G. Berry, H. Comon and A. Finkel Editors, Lecture Notes in Computer Science 2102, p. 246–249, 2001.
5. T. S. Gegg-Harrison, G. R. Bunce, R. D. Ganetzky, C. M. Olson, J. D. Wilson. Studying program correctness by constructing contracts. SIGCSE Bull. 35, 3 (Sep. 2003), 129-133.
6. D. Gries. *The Science of Programming*, Springer-Verlag New York, Inc., Secaucus, NJ, 1987.
7. IFIP Verified Software: Theories, Tools, Experiments Conference, International Federation for Information Processing, ETH Zürich, October 10-13, 2005. Report available at: http://vstte.ethz.ch/.
8. C. Jones, P. O'Hearn, J. Woodcock, J. Verified Software: A Grand Challenge. Computer 39, 4 (Apr. 2006), 93-95.
9. L. Lamport. How to Tell a Program from an Automobile. January, 28th 1977. Available at: http://research.microsoft.com/users/lamport/pubs/automobile.pdf.
10. J. Langr. Agile Java: Crafting Code with Test-Driven Development, Prentice Hall, February 2005, ISBN: 0131482394.
11. B. Meyer. Applying "Design by Contract". Computer 25, 10 (Oct. 1992), 40-51.
12. B. Meyer. *Eiffel: The Language*, Prentice Hall, 1993.
13. N. Shankar. Little engines of proof. In L.-H. Eriksson and P. Lindsay, editors, FME 2002.

Tool Support for Learning Büchi Automata and Linear Temporal Logic*

Yih-Kuen Tsay, Yu-Fang Chen, and Kang-Nien Wu

Dept. of Information Management, National Taiwan University, TAIWAN

Abstract. Automata and logics are intimately related, and understanding their relation is instrumental in discovering algorithmic solutions to formal reasoning problems or simply in using those solutions. This applies to Büchi automata and linear temporal logic, which have become fundamental components of the model-checking approach to formal verification of concurrent systems. Translation of a propositional temporal formula into an equivalent Büchi automaton is routinely performed in many model-checking algorithms and tools. Albeit the possibility of mechanical translation, a temporal formula and its equivalent automaton appear to be two very different artifacts and their correspondence is not easy to grasp.
In this paper, we introduce a graphical interactive tool, named GOAL, that can assist the user in understanding the relation between Büchi automata and linear temporal logic, and suggest possible usages and benefits of the tool in courses where model-checking techniques are covered. GOAL builds on the successful JFLAP tool for classic theory of automata and formal languages. One main function of GOAL is translation of a propositional temporal formula into an equivalent Büchi automaton that can be visually manipulated, for example, running the automaton on some input. GOAL also supports various standard operations and tests, including equivalence test, on Büchi automata. We believe that, with an easy access to temporal formulae and their graphically presented equivalent Büchi automata, the student's understanding of the two formalisms and their relation will be greatly enhanced.

1 Introduction

The model-checking approach to formal verification of concurrent systems seeks to automatically verify if the given system represented by an abstract model satisfies its specification. Because of its proven effectiveness and ease of use, model checking has become a viable alternative to simulation and testing in industry. Model checkers are also increasingly exploited by verification tools based on deductive (theorem proving) methods, as the work horses for decidable verification subtasks.

In one school of model checking, a concurrent system is equated semantically with a set of infinite computations and its desired behavioral properties are then specified in terms of those computations. The specification of a behavioral property typically asserts temporal dependency between occurrences of certain events (represented by propositions) and linear temporal logic has thus become a particularly popular class

* This work was supported in part by the National Science Council of Taiwan (R.O.C.) under grant NSC 94-2213-E-002-089.

of languages for specification. Temporal dependency between events may also be expressed with Büchi automata, which are finite automata operating on infinite words.

Indeed, automata and logics are intimately related, as we all have learned from classic theory of computation. Understanding their relation is instrumental in discovering algorithmic solutions to formal reasoning problems or simply in using those solutions. This applies to Büchi automata and linear temporal logic. It has been shown that Büchi automata and a variant of linear temporal logic called quantified propositional temporal logic are expressively equivalent. For the pure propositional temporal logic (PTL), practically feasible algorithms exist for translating a PTL formula (which is usually short as a specification) into an equivalent Büchi automaton.

As Büchi automata are also suitable as abstract system models, many researchers have advocated a unified model-checking approach based on automata. In this approach, the negation of the specification formula is translated into an automaton, representing the bad behaviors. The intersection of the system automaton and the negated-specification automaton is then constructed and checked for emptiness. If the intersection automaton accepts no input (i.e., the system and the negated specification do not have any common behavior), then the system is correct with respect to the original specification formula.

Translation of a PTL formula into an equivalent Büchi automaton is now routinely performed in many model-checking algorithms and tools. Albeit the possibility of mechanical translation, a temporal formula and its equivalent Büchi automaton are two very different artifacts and their correspondence is not easy to grasp. Temporal formulae describe temporal dependency without explicit references to time points and are in general more abstract, while Büchi automata "localize" temporal dependency to relations between states and tend to be of lower level. Nonetheless, their relation can be better understood by going through some translation algorithm with different input temporal formulae or simply by examining more examples of temporal formulae and their equivalent Büchi automata. This learning process, however, is tedious and prone to mistakes for the students, while preparing the material is very time-consuming for the instructor. Tool support is needed.

In this paper, we introduce a graphical interactive tool, named GOAL (**G**raphical Interactive Tool for **O**mega-**A**utomata and Temporal **L**ogic), that has been designed and implemented for this purpose, and suggest possible usages and benefits of the tool in courses where model-checking techniques are covered. GOAL builds on the successful JFLAP tool for classic theory of automata and formal languages. One main function of GOAL is translation of a PTL formula into an equivalent Büchi automaton that can be visually manipulated, for example, running the automaton on some input. The user has an option of viewing the intermediate steps that the translation goes through, in particular, which states of the automaton come from which parts of the input temporal formula. GOAL also supports various standard operations and tests, including equivalence test, on Büchi automata. We believe that, with an easy access to temporal formulae and their graphically presented equivalent Büchi automata, the student's understanding of the two formalisms and their relation will be greatly enhanced.

To the best of our knowledge, GOAL is the first graphical interactive tool designed mainly for teaching and learning Büchi automata and linear temporal logic. It supports

the full range of temporal operators, including all past and future temporal operators defined in Manna and Pnueli's book [11]. There are other tools that provide translation of PTL formulae into Büchi automata, e.g., SPIN [8] and LTL2BA [5]. However, none of them provide facilities for visually manipulating automata and few support past temporal operators. The operations and tests on Büchi automata provided by GOAL are also more comprehensive than those by other tools.

2 Büchi Automata, Linear Temporal Logic, and Model Checking

Büchi Automata. Büchi automata are a variant of ω-automata, namely finite automata operating on infinite words. A Büchi automaton accepts those inputs that can drive it through some accepting state infinitely many times. Büchi automata are closed under intersection and complementation [1,7]. Complementation, unlike in the case of finite words, is highly complex [16,14,9,10,4] and known to have an exponential lower bound [12]. Minimizing the number of states is also a hard problem [3,17]. Generalized Büchi automata have multiple sets of accepting states. A generalized Büchi automaton accepts those inputs that can drive it through each of the accepting sets infinitely many times. Generalized Büchi automata and many other variants of ω-automata are equivalent to Büchi automata in expressive power.

Linear Temporal Logic. Linear temporal logic (LTL) has as semantic models infinite sequences of states, which can also be seen as infinite words over a suitable alphabet. We use Propositional Temporal Logic (PTL) to refer to the pure propositional version of LTL, for which a state is simply a subset of atomic propositions holding in that state. PTL formulae are constructed by applying boolean connectives and temporal operators to atomic propositions drawn from a predefined universe. For instance, the formula $\Box(p \rightarrow \diamondsuit q)$ combines two temporal operators, $\Box$ (always) and $\diamondsuit$ (once), to say that "every p is preceded by q" or equivalently "the first p does not occur before the first q". The formula $\Box(p \rightarrow p \mathscr{U} q)$ says that "once p becomes true, it will remain true continuously until q becomes true, which must eventually occur".

(a) $\Box(p \rightarrow \diamondsuit q)$ (b) $\Box(p \rightarrow p \mathscr{U} q)$

Fig. 1. Two PTL formulae and their respective equivalent Büchi automata, where the darker states are accepting states.

A PTL formula can be translated into an equivalent Büchi automaton (but not vice versa) in the sense that every infinite sequence satisfying the formula corresponds to an infinite word accepted by the automaton [19,5,6]. As an illustration, we exam the

Büchi automata equivalent to the two example temporal formulae. The alphabet for both automata is $\{\mathtt{pq}, \mathtt{p\tilde{}q}, \mathtt{\tilde{}pq}, \mathtt{\tilde{}p\tilde{}q}\}$. The Büchi automaton in Figure 1(a) is equivalent to the formula $\Box(p \rightarrow \diamondsuit q)$. From state q_0, there is no transition for $\mathtt{p\tilde{}q}$, ensuring that "the first p does not occur before the first q". The Büchi automaton in Figure 1(b) is equivalent to the formula $\Box(p \rightarrow p \mathscr{U} q)$. An occurrence of $\mathtt{p\tilde{}q}$ brings the automaton from q_0 to q_1, where no transition is possible for $\mathtt{\tilde{}p\tilde{}q}$. So, once p becomes true, it has to remain true until q becomes true. In addition, as q_1 is not an accepting state, either $\mathtt{pq}$ or $\mathtt{\tilde{}pq}$ must occur, bringing the automaton to the accepting state q_0.

Another variant of LTL called Quantified Propositional Temporal Logic (QPTL) [15] additionally allows quantification over atomic propositions. QPTL are equivalent to Büchi automata in expressive power, though the translation from formula to automaton involves a non-elementary blow-up of number of states [16].

Model Checking. Model checking seeks to automatically verify if a given system satisfies its specification [2]. The system is typically modeled as a Kripke structure, which is essentially a state-transition graph where each state is labeled with those propositions that hold in that state; fairness may be imposed on how the transitions should be taken. When the specification is given by a PTL formula, the model checker determines if every computation (sequence of states) generated by the system satisfies (or is a model of) the formula.

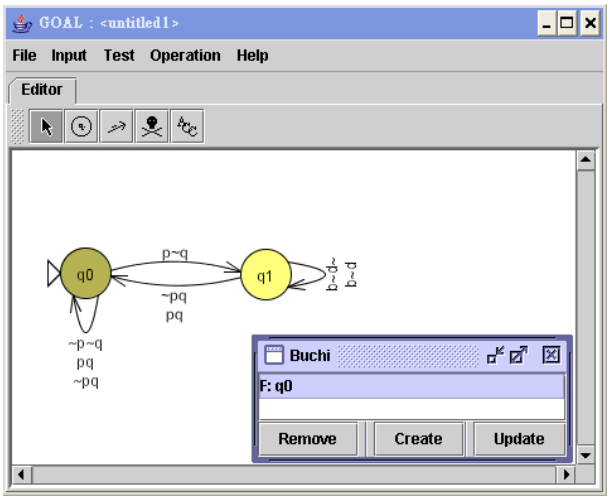
The system may also be modeled as a Büchi automaton; in fact, every Kripke structure (with or without the usual fairness conditions) corresponds to some Büchi automaton. As the PTL formula can also be translated into a Büchi automaton, this results in a uniform treatment of both the system and its specification [18]. Suppose A is the automaton modeling the system and B_φ the automaton representing the specification φ. Let $L(A)$ and $L(B_\varphi)$ denote respectively the languages of the two automata. The problem of model checking translates into that of language containment $L(A) \subseteq L(B_\varphi)$. Let $\overline{L(B_\varphi)}$ denote the complement of $L(B_\varphi)$. The problem is then equivalent to checking if $L(A) \cap \overline{L(B_\varphi)} = \emptyset$. As Büchi automata are closed under intersection and complementation, this reduces to the emptiness problem of Büchi automata.

However, complementation of a Büchi automaton is expensive. A better alternative is to first negate the given PTL formula φ and obtain the equivalent automaton $B_{\neg\varphi}$ such that $L(B_{\neg\varphi}) = \overline{L(B_\varphi)}$. Now, to check if $L(A) \cap L(B_{\neg\varphi}) = \emptyset$, one only needs to construct the intersection of A and $B_{\neg\varphi}$ and complementation is avoided.

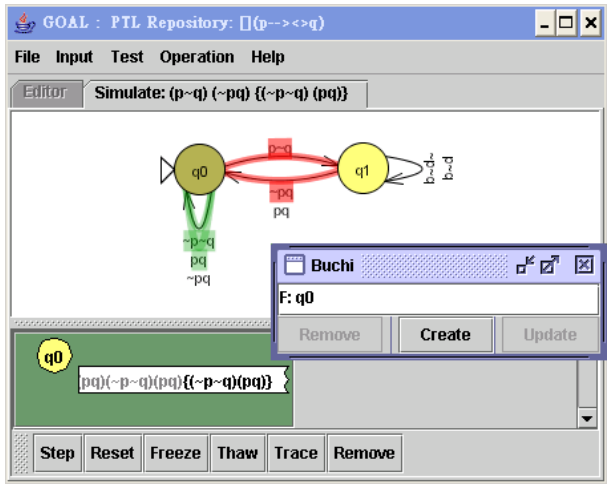
3 The GOAL Tool

In this section, we describe the functions of GOAL (**G**raphical Interactive Tool for **O**mega-**A**utomata and Temporal **L**ogic) and their implementation. The current version of GOAL provides the following functions:

– **Drawing and Running Büchi Automata**: The user can easily point-and-click and drag-and-drop to create a Büchi automaton; the automata in Figure 1 were drawn using GOAL. After an automaton is created, the user can run it through some input to get a feel of what kind of inputs the automaton accepts, as shown in Figure 2.

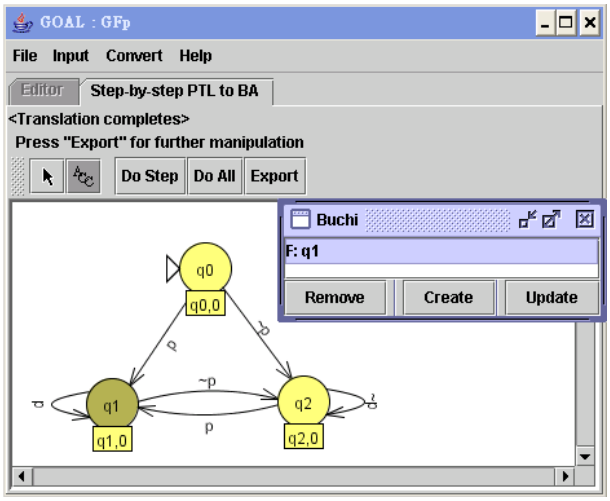


(a) A Büchi automaton drawn by the user

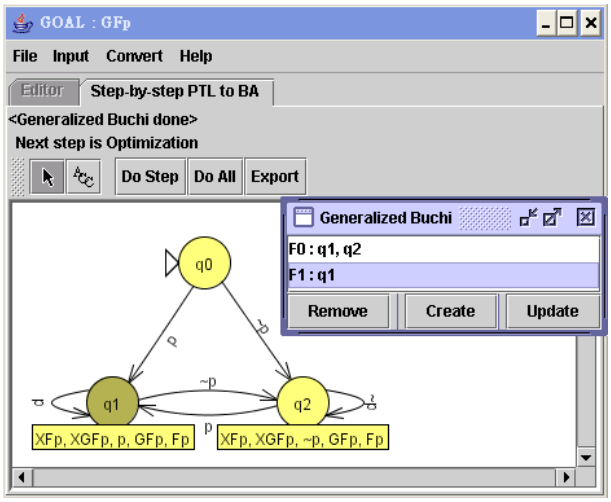


(b) Running the Büchi automaton through an input

Fig. 2. Screen shots of the GOAL tool. The inset window in each screen shot shows the set of accepting states. In Part (b), the pair of "{" and "}" in the input indicates an infinite repetition.



(a) A Büchi automaton translated from GFp (or $\Box\Diamond p$)



(b) The intermediate generalized Büchi automaton

Fig. 3. More screen shots of the GOAL tool. The PTL formula GFp is translated into an equivalent Büchi Automaton. If the translation is followed step by step, the user can also see the intermediate generalized Büchi automaton.

– **From PTL Formulae to Büchi Automata**: After the user has typed in a PTL formula and chosen a suitable translation option, the system responds by displaying an equivalent Büchi automaton, as shown in Figure 3(a) with "GFp" or equivalently "$\Box\Diamond p$" as input. With another option, the system first displays a generalized Büchi automaton and then, upon the user's request, will convert it into a Büchi automaton; Figure 3(b) shows such an intermediate result. The supported temporal operators and their input formats are as follows:

Operator	○	□	◇	$\mathscr{U}$	$\mathscr{W}$	$\mathscr{R}$	⊖	⊜	⊟	⟐	$\mathscr{S}$	$\mathscr{B}$
Format 1	()	[]	<>	U	W	R	(-)	(~)	[-]	<->	S	B
Format 2	X	G	F	U	W	R	Y	Z	H	O	S	B

– **The PTL Repository**: The repository stores a collection of commonly seen patterns of PTL formulae and their respective equivalent Büchi automata, which were drawn by human using GOAL itself and are smaller than machine-translated ones.
– **Boolean Operations on Büchi Automata**: The three standard boolean operations—union, intersection, and complementation are supported.
– **Tests on Büchi Automata**: Emptiness, (language) containment, and equivalence tests are supported. In the emptiness test, if the given Büchi Automaton is non-empty, the system highlights the path that corresponds to an accepted input. The equivalence test of two Büchi Automata is built on top of the containment test which in turns relies on the intersection and complementation operations and the emptiness test.

The automata and the graph modules of GOAL were adapted from those of JFLAP [13]. The most complicated algorithms in GOAL are those for translating temporal formulae to automata and for complementing automata. Our translation algorithm is an adaptation of the tableau construction described in Chapter 5 of Manna and Pnueli's book [11]. For automata complementation, we adopted the algorithm by Safra [14]. Even with inputs of a moderate size, these algorithms may produce very large automata, which are difficult to display and usually impossible to understand intuitively. However, this is not a very serious problem. As GOAL is positioned mainly as an instructional tool, we assume that it will be used with inputs of short temporal formulae or small automata.

A few words are in order about our translation algorithm. Though it generates more states than others do, the algorithm has two advantages: it handles past temporal operators and is relatively simple (which is good for educational purposes). The steps can be easily divided and their intentions clearly illustrated. Some published translation algorithms are indirect, e.g., the translation in [5] used a very weak alternating automaton as the intermediary, while some combine multiple steps into one, e.g., the translation in [6] constructed states and established transitions in the same step. To reduce the number of states, we implemented several methods for state reduction, for example, removing redundant states detected by simulation [3].

4 GOAL in Classroom and More

As the implementation of its main functions has just recently been completed, we have yet to use the GOAL tool in an actual classroom setting. However, an analogy can be

drawn from the use of JFLAP [13], a visual interactive tool for teaching and learning classic theory of automata and formal languages that inspired GOAL. The first author has used JFLAP for several years in his Theory of Computation course for junior and senior undergraduate students. Both the students and the instructor have enjoyed the illuminating visualization that the tool provides. It helps to be able to *see* how an automaton, particularly a *nondeterministic* one, runs on an input. A convenient tool for drawing automata also encourages the students to do more exercises. It was a delight to find that a visual tool has breathed life into an important foundational computer science course that would otherwise be dull to most students. Moreover, as the diagrams can be exported with a PDF printing support, the tool has also saved the instructor's time when preparing handout material and the students' when writing up their homework. We believe the same will apply for GOAL.

The GOAL tool should be useful as teaching and learning support for courses on model checking, formal verification, or even advanced automata theory where ω-automata and temporal logic are essential topics. Our immediate plan is to use GOAL in a course of this Fall titled Software Development Methods, which aims at improving students' ability in designing quality software. The course covers three topics: software modeling, design patterns, and formal verification. In the verification part, we will cover model checking techniques and tools where GOAL can help. Though the emphasis is not on translation algorithms, the student will be asked to write the same specifications with Büchi automata and temporal formulae. With the help of GOAL (particularly the equivalence test), they will be able to quickly validate their answers. They can try out a few inputs on a Büchi automaton to get a better understanding of what its language is. For the more aspiring students, GOAL can provide them with guidance on how a Büchi automaton is obtained systematically from a PTL formula (though not necessarily in an optimal way).

Lastly, we would like to mention that the authors and other members in their group have already started to benefit from developing and using GOAL. Among other things, we learned to appreciate the difficulty and importance of minimizing the number of states of a Büchi automaton during the implementation. No known algorithms guarantee minimality on the number of states. Still any heuristics that help reduce the number of states are valuable, as automata with a smaller number of states are usually easier to understand intuitively (and to verify automatically). Indeed, research and teaching do go hand in hand.

Acknowledgment. We thank Susan H. Rodger, the creator of JFLAP, at Duke University for granting us the permission to use and modify the JFLAP source code. Without the automata and the graph modules of JFLAP (which we had to tweak a little bit), the GOAL tool would not be possible in a relatively short time. We also thank Wen-Chin Chan who helped us with the adaptation of JFLAP's graph module.

References

1. J.R. Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Standford University Press, 1962.

2. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.
3. K. Etessami and G. Holzmann. Optimizing Büchi automata. In *CONCUR 2000, LNCS 1877*, pages 153–167. Springer, 2000.
4. E. Friedgut, O. Kupferman, and M.Y. Vardi. Büchi complementation made tighter. In *Proceedings of the 2nd International Symposium on Automated Technology for Verification and Analysis, LNCS 3299*, pages 64–78. Springer, 2004.
5. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translations. In *Proceedings of the 13th International Conference on Computer-Aided Verification, LNCS 2102*, pages 53–65. Springer, 2001.
6. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, 1995.
7. E. Grädel, W. Thomas, and T. Wilke. *Automata, Logics, and Infinite Games (LNCS 2500)*. Springer, 2002.
8. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
9. N. Klarlund. Progress measures for complementation of ω-automata with application to temporal logic. In *Proceedings of the 32nd IEEE Conference on Foundations of Computer Science*, pages 358–367, 1991.
10. O. Kupferman and M. Vardi. Weak alternating automata are not that weak. *ACM Transactions on Computational Logic*, 2(3):408–429, 2001.
11. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safty*. Springer, 1995.
12. M. Michel. Complementation is more difficult with automata on infinite words. In *CNET, Paris*, 1988.
13. S. Rodger and T. Finley. JFLAP. http://www.jflap.org/.
14. S. Safra. On the complexity of ω-automta. In *Proceedings of the 29th IEEE Conference on Foundations of Computer Science*, pages 319–327, 1988.
15. A.P. Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD thesis, Harvard University, 1983.
16. A.P. Sistla, M. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.
17. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Proceedings of the 12th International Conference on Computer-Aided Verification, LNCS 1855*, pages 248–263. Springer, 2000.
18. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the 1st IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.
19. P. Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 28(110):119–136, 1985.

Enhancing student understanding of formal method through prototyping

Andreea Barbu and Fabrice Mourlin

LACL (France)
dreea@gmx.net and Fabrice.Mourlin@wanadoo.fr

Abstract. Formal methods refer to a variety of mathematical modeling techniques, which are used to model the behavior of a computer system and to verify that the system satisfies design, safety and functional properties. In this paper, we record the experience of teaching Formal Specification to master degree Computer Science students, using the higher order π-calculus language. We present a methodology for teaching formal specification based on a collaboration tool called HOPiTool. This tool is the input gate to build a rapid prototype of mobile agent application. We argue that students, through a rigorous integrated development environment to formal specification, acquire knowledge, skills and abilities that are essential in their professional lives as Computer Scientists.

Keywords: Mobile computing, Teaching and Learning, Formal Methods, Formal Specification.

1 Introduction

Formal methods are considered as a basis for most research frameworks at master degree level. However, its approach often seems difficult to understand for students, because of the complexity of formal methods. The use of formal method is clearly presented at the beginning of the scholar year by the need of formal specification in the context of mobile computing. This formal facet is also explained by the use of property proof. Yet all of these reasons are merely considered a pretext to teach mathematics in computer science lessons. In fact, our students have a pragmatic profile and their experience is essentially based on design and programming studies. Also, the use of semi-formal methods is more established and well accepted by master degree students.

Since 2004, we decided to introduce formal methods in teaching mobility. Our objective was to introduce formal methods not only as general knowledge which could be useful for everyone, but as a wizard which would be able to lead the programming of a mobile agent application. The direct application was immediately understood by our students and the motivation of everyone was obvious: not only can the formal method be useful in approaching a technical subject, but has also piqued the interest of companies such as those in the telecom industry.

Our teaching orientation was geared towards the use of a specification platform (called Higher- Order π-calculus tool HOPiTool) which we have developed on purpose. This high-level tool allows students to generate mobile agent application skeletons. This tool has also become the main interest in a first approach and, because it is the

vehicle of a specific formal calculus, the Milner's formal language was understood and assimilated by all beginners in the group of students. Some features of our tool can be compared to Mobile WorkBench Tool (for a large subset of π-calculus) [6] but the context and the functionalities (interaction, and code templating operations) have not the same application.

This paper explains the architecture of our teaching approach. First we describe the application domain of the formal methods we use; secondly we place all actors of our scenario and depict a specific lesson. Then, we list the immediate results and, step by step how our technical tool plays the role of an exchange platform between users and experts. Finally, we sum up with the projection of our work on some other subjects and contexts.

2 Architecture of our teaching approach

Previously, the formal methods course unit was lecture-based (three hour-long lectures per week), with the support of weekly tutorials in the first term and (subsequently) weekly practical sessions in the second term. The lectures introduced the following broad topics:

- Industrial use of formal methods [1];
- Mathematical notation (i.e. logic, sets, relations, functions, sequences, etc.)
- Property proof from model, and formal theory

Because of this situation, and the increasing number of students attaining poor marks for the formal methods unit, it was decided to change our approach. In 2002, we built a platform dedicated to formal specification teaching. Our tool was initially intended as a formal specification editor and certifier, rather than a teaching tool. However the initial use showed promise for another application.

The core choice of the approach had to do with which formal language to adopt and teach. This choice was directed by the potential applications; the most direct applications were the most whacking ones.

Because of our abilities in mobile computing and nomadic computing, we chose the higher order π-calculus language created by Robin Milner [2].

In theoretical computer science, the π-calculus is a notation originally developed by Robin Milner, Joachim Parrow and David Walker as advancement over Calculus of Communicating Systems in order to provide mobility in modeling concurrency [3]. In the family of process calculi that have been used to model concurrent programming, the π-calculus plays a role similar to that of the λ-calculus concerning sequential programming.

The expressiveness of this formal calculus allows us to consider mobile feature as basic structural feature of our specifications. A formal π-calculus specification is, thus, easy to interpret as an operational description of a real mobile application on a concrete device. This factor was a valuable asset with our students - especially those having a technical background.

Mobile computing applications were also a key argument in our presentation. This kind of direct application is interesting for large industry, with companies such as Motorola, which sprung up to support the growing need for mobile devices. Some other

domains are taken into account such as ubiquitous computing. Students are receptive to these real issues and discover by the end of the academic year that companies are very interested in people who are familiar with these subjects.

At the beginning of the development map, we determined some basic features for our tool:

- The user must obtain an initial prototype when his formal specification is correct (respect a set of rules),
- The prototype is automatically executable on the current network, even if it is restricted to one node,
- The platform has to be collaborative - meaning that the work of a user can be observable by any one else with permission to do so,
- A specification repository should offer basic 'bricks' for beginners to start with,
- At the end of a user project, its owner can only consider a formal specification as a formal view of the project, as opposed to the initial prototype, which is an executable one.

Since then each lecture became divided into two parts: an explanation of the lecture concerning a specific concept (about one hour), followed by a concrete and direct application of this concept through practical work using the HOPiTool.

For example, the first lecture is about the deployment of an agent in a graph and its consequences on the communication scheme. The second part of the lecture is about the specification of a system based on a set of sentries which control telnet protocol and forward information when a connection occurs. A sentry is an agent which observes a specific protocol or has a precise task. The section which follows highlights the role of the practical aspect in our global teaching strategy.

3 An essential direct application of formal method

This template of lecture is the basis for all lectures in the course. The direct application allows the teacher to evaluate not only the level of a student but also the homogeneity of the group. This note is explained via the Figure 1 below. After the explanation of the teacher, each student of the group receives the requirements for the formal specification he has to write. Then a controller agent is installed on the teacher's platform. This installation step means that a directory of specifications (for the controller) is duplicated Each student can consult this specification as a reader. It serves as a reference that the telnet agents should exchange data with it.

Then each student can write his own specification in his workspace. This workspace is hidden from the other students with only the teacher having access as a reader. HOPiTool helps the student not only with checking syntax, but also checking whether an agent specification can be evaluated in a more complex system where some agents (a controller and its student agent) share some channels or gate vocabulary.

The following agent generation step creates a local prototype from a student specification, in the same workspace. Of course, this stage can detect some ambiguities and even some errors. This semantic step has to be configured (by the teacher at the beginning of the use: some network features, etc.).

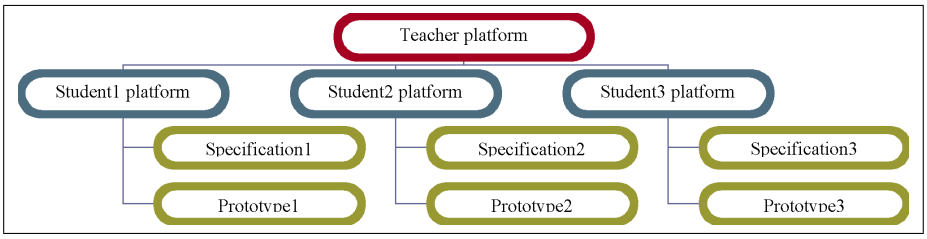


Fig. 1. A snapshot of a working period

Simulation and observation follows. Before launching his prototype, the student must notify the teacher that he is ready to start his own agent (or his own set of agents).

The event provides information about the skill of the students, even if the subject is similar to the first part of the lecture. Secondly, the teacher has to launch his controller prototype. Then, the simulation can elapse. Depending on the construction of each part, a result can be observed by both the student and the teacher.

Finally, a global simulation can be prepared if the workspace of each student is made available to the others. In this last step the teacher can assess students' understanding of the notions involved in the prototyping.

To sum up this section, we note that a direct application is controlled and driven by the teacher. This scheme is also available in another context: when several students work in collaboration with each other. In this case, the role of HOPiTool platform is quite different, the teacher platform does not support an agent and its role is that of an observer of the activity within a student group. This is particularly useful to understand and evaluate the origin of problems in school projects.

As HOPiTool is always available on the student's platform, each student's work (specification of prototype) which is not finished during a lecture must be realized before the following lecture. However, the same measurements can be assessed (duration of work, relation with some other student's work). These conditions motivate each student to achieve his goals.

The next section presents in detail a specific lecture and how formal method and mobile prototype are combined.

4 Description of a specific lecture

The timing of each lecture needs to be precisely defined as the measurement could be wrong or merely approximate. In this section we will present a fragment of a lecture with the subject being how to export an agent.

The first part of the lecture is dedicated to the expression of the agent exportation and, especially, the signature of the exported agent and how another agent can receive or import an agent. The signature is explained and the scope of names which are looked up by the agent (called B on figure 2). Robin Milner uses unification algorithm and some extension are added for the higher- order expression. These properties are particularly

difficult for students to grasp, who do not have a background in logics. A first reading of such specification is complex and students made mistake between pattern matching and unification.

The figure below highlights a scenario: an agent A exports another agent S like it could publish a service. On another node, an agent called B wishes to import this agent like it could be subscribed to a remote service.

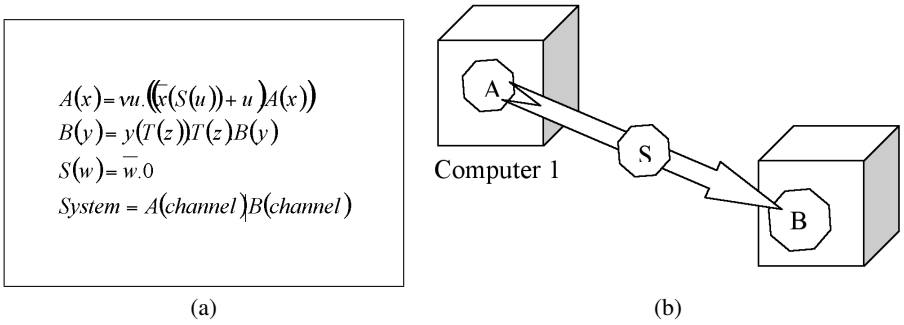


Fig. 2. Higher-order communication description (a) Higher-order π-calculus specification, (b) Box diagram

The second part of the lecture involves a case study on SLP (Service Location Protocol) [4]. The students build a formal specification of SLP protocol where the mobile features are highlighted. Of course, these aspects involve the use of agent exportation. From a reference of the SLP protocol each student defines two kinds of agents: a directory agent and a service agent. The teacher specifies a user agent. These names come from the SLP IETF reference document. The next figure show both parts of the specification and the gates which are used to exchange them.

$$SA(Srv\,\mathrm{Re}\,g, SrvAck) = Srv\,\mathrm{Re}\,g(Service(pr\,\mathrm{int}, msg).SrvAck.SA(Srv\,\mathrm{Re}\,g, SrvAck)$$
$$DA(Srv\,\mathrm{Re}\,g, SrvRqst, SrvAck) = (Srv\,\mathrm{Re}\,g.(S(name, f)input(S(name, f).SrvAck)))$$
$$|(SrvRqst(S(name, f), SrvRply)name(SrvRply))DA(Srv\,\mathrm{Re}\,g, SrvRqst, SrvAck)$$

Fig. 3. Agent specification UA is written by the teacher and SA, DA are written by the student

Several auxiliary agents are defined by the students (to keep all the exchanges of services). This part of the work is hidden here but is essential for the overall specification. The first half of the second part is used for the specification and the controls, upon which the generation code can take place. The entire prototype is written in Java

and Jini (Java Intelligent network Interface); we use the results of some previous research work in our research team about mobile agents [5]. This library maps agents to classes in a way such that the exchange of agents is described as a Jini service. Even if the students are not familiar with the library, they can consider the results of the code generation as a potential validation of their previous work. Thus, one class per agent is generated and an ant file is created with all the useful targets for the compilation, the deployment and the execution of the prototype. The first execution happens between the teacher UA prototype and the student prototype. When this level is satisfied, a more complex validation can be done with a distinct service per student and a more complex UA agent. The interpretation of the results is immediate and the link between higher order π-calculus and the mobile code becomes apparent. The next section presents the consequences of our approach and how HOPiTool is enhanced by the development of additional modules.

5 Immediate feedback

The overall course schedule is today quite different from what it used to be. Two years ago, the plan followed the difficulties of the subjects, now it is implicit in the step by step control of teaching material made possible by HOPiTool. The core of the course may appear to be centered on the HOPiTool platform, yet the tool itself serves only as a tool, a pretext for other direct applications. Two main applications are realized: in the first year, a set of case studies are built. They represent a library or a reference for the beginners.

Next, the formal approach is enhanced with a test module. It allows the student to write specifications faster while improving quality. This module was a project initiated by a research group, but implemented during a master's degree project. The goal is to generate some test suites which respect the formal specification written by a student. These tests can be composed into a hierarchy of test suites. The test suites contain test cases and even other test suites. When the prototype is generated, the tests can be run automatically and can check their own results. When a student runs tests, he gets simple and immediate visual feedback as to whether the tests pass or fail. There is no need to manually comb through a report of test results.

The structure of a student workspace has evolved as shown in figure 4. The test cases can be run automatically or manually depending on the level of the student. The test generation is a pedagogical approach; it completes the understanding of the specification. Several criteria can be added, for instance, to insure that all branches of the specification are explored. These test cases arise from the specification, yet their Java derivations illustrate the role of the higher order π-calculus language. This adds to the belief that formal languages are a key brick in software development.

6 Conclusion

In our experience, the use of a pedagogical tool led to a new organization of the entire course. Because it is not only a help for every one but also something more complex, the structure of each lecture changes. Now even the theoretical courses have practical

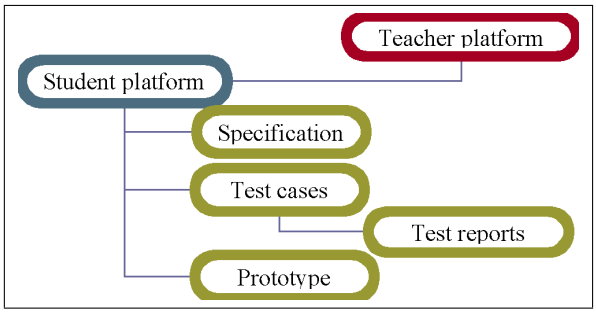


Fig. 4. New structure of the student workspace

exercises. The structuring of a lecture consists of one hour for theory and two hours for practice. The main result is a different behavior in the students; the formal subject becomes more appealing.

Finally, the evaluation of the module has changed since the HOPiTool platform becomes a framework for the exams. The validation of the students' exam is also simplified with the use of test cases. The teachers provide a reference after the exam with statistics. Also, tracking the students' progress becomes easier than before and scores are improved.

References

1. M. G. Hinchey and J. P. Bowen, editors: "Industrial-Strength, Formal Methods in Practice". FACIT Series, Springer-Verlag, 1999.
2. R. Milner, J. Parrow, D. Walker: "A calculus of mobile processes", part I and II, Information and computation, Number 100, 1989.
3. D. Sangiorgi and D. Walker: "The π-calculus: A Theory of Mobile Processes", Cambridge University Press, ISBN 0521781779, 2000.
4. J. Kempf, and P. St. Pierre: "Service Location Protocol for Enterprise Networks", Wiley, New York, 1999.
5. M. Bernichi and F. Mourlin: "A New Behavioural Pattern for Mobile Code" In ESM 2005, University of Porto, Porto, Portugal, 24-26 October 2005.
6. Victor, B., Moeller, F., 1994. The Mobility Workbench - A tool for the π-calculus. In: D. Dill (Ed.), Proceedings CAV'94: Computer-Aided Verification. Lecture Notes in Computer Science, Vol.818, Springer Verlag, Berlin, 428-440.

Evaluating a Formal Methods Technique via Student Assessed Exercises

Alastair F. Donaldson* and Alice Miller

Department of Computing Science
University of Glasgow
Glasgow, Scotland.
{ally,alice}@dcs.gla.ac.uk

Abstract. We present a case study in evaluating a formal methods technique, using student assessed exercise solutions as sample input to SymmExtractor, a symmetry detection tool for the SPIN model checker. We discuss the ethical procedure which must be followed when using student programs for research, and present the results of our evaluation.

1 Introduction

The mathematical nature of formal methods means that designers of a technique or tool may give little thought to its widespread usability, concentrating chiefly on theoretical soundness and elegance with respect to a limited application domain. Although the rigour of a formal approach depends on these theoretical aspects, usability is clearly important if techniques and tools are to be adopted by industry, educators, or other researchers.

Formal methods students are suitable participants for evaluation of formal methods techniques — their interest is balanced between the practical aspects of a particular tool and the theory which underlies its construction, and they will reject a technique which has limited application even if it is theoretically elegant. Further, such evaluation can be mutually beneficial: students can learn about practical formal methods by using certain techniques and tools, and simultaneously aid research into the usability of prototype extensions.

The contribution of this paper is a case study evaluating an automatic symmetry detection tool, SymmExtractor. The evaluation is based on a set of example solutions to an assessed exercise from the *Modelling Reactive Systems* final year course at the University of Glasgow. We discuss the ethical issues involved in using student programs for research, present the design of our evaluation, and propose some changes to SymmExtractor based on the evaluation results. The benefit of this evaluation was one way: students allowed us to use their solutions in our research. We conclude by proposing a symbiotic approach in which students' education in formal methods can also benefit from participation in evaluation.

* Supported by the Carnegie Trust for the Universities of Scotland.

2 Symmetry in Model Checking

Model checking [5] is a popular automated formal reasoning technique whereby temporal logic properties of a concurrent system can be checked via an abstract, finite state model of the system. A widely used model checker suitable for protocol verification is SPIN, which allows reasoning over specifications written in Promela [10]. The application of model checking is limited due to the state space explosion problem – as the number of components in a system increases, the state space of its associated model grows combinatorially, becoming too large to reason over. Symmetry reduction techniques aim to combat this problem by exploiting replication in the topology of the system. Such replication may induce automorphisms, or *symmetries* of the underlying state space. These automorphisms form a group, any subgroup of which can be used to partition the state space into equivalence classes. Certain temporal properties can be checked over a *quotient* state space, consisting of one state per equivalence class. This can potentially result in large savings in memory and verification time [3,7].

For details of symmetry reduction in model checking, see a recent survey [12]. We now describe two tools for analysing symmetry in Promela models. The first, SymmExtractor, infers symmetries of the state space underlying a Promela specification via static analysis of the specification. The second, SPIN-to-GRAPE, allows generators for the group of all symmetries of a small state space to be computed, and was used extensively in the design and evaluation of SymmExtractor.

2.1 The SymmExtractor and SPIN-to-GRAPETools

For symmetry reduction to be a useful technique it is vital to be able to detect symmetries of a state space without actually building the state space. There are three main approaches to symmetry detection: restricting the input language so that the state space associated with a specification is *guaranteed* to be symmetric [15]; extending the language with keywords which can be used to *specify* symmetry [11]; and inferring symmetries of the state space by analysing the communication structure of a specification [3,6]. The third approach is less restrictive than the first, and avoids the manual effort of the second, thus potentially allows symmetry reduction to be a "push button" technique.

SymmExtractor [6] performs automatic symmetry detection for Promela by extracting the *static channel diagram* (SCD) of a Promela specification. The SCD is a graphical representation of potential communication between components of the specification. Generators for the group of symmetries of the SCD, $Aut(SCD)$, is computed, and the computational algebra system GAP [9] is used to determine generators for a subgroup of $Aut(SCD)$ which induces a symmetry group of the underlying state space, and thus can be used for symmetry-reduced model checking. SymmExtractor is incorporated in TopSPIN [7], which uses the detected symmetries to for state space reduction when model checking with SPIN.

SymmExtractor requires that a Promela specification obeys certain restrictions. These include: the use of an `init` process in which all processes are instantiated simultaneously; restrictions on the use of channel variables; and a constrained set of allowable operations on variables which take as their values process identifiers. These restrictions aim to make automatic symmetry detection tractable and straightforward to implement,

without unduly modifying the way that Promela programs are specified. The evaluation presented in this paper aims to identify mismatches between these restrictions and natural modelling styles used by students in a set of sample specifications.

For small models, it can be illuminating to construct the state space as a directed graph and explicitly compute its group of automorphisms. The SPIN-to-GRAPE tool [8] outputs the state space underlying a Promela specification as a directed graph suitable for input to GRAPE [16], a graph-theoretic add on to GAP. GRAPE can then be used to compute the automorphism group of the state space and the corresponding quotient state space.

SPIN-to-GRAPE can be used to explicitly analyse the symmetry group of a state space with up to several thousand states. The tool was used extensively for testing purposes during the development of SymmExtractor and TopSPIN [7] (a symmetry reduction package for SPIN which incorporates SymmExtractor). In Section 4.2 we discuss the role of SPIN-to-GRAPE in the evaluation of SymmExtractor.

3 Modelling Task – a Telephone Exchange

Modelling Reactive Systems (MRS) is a final year 20-lecture formal methods course at the University of Glasgow. The primary focus of the course is on the theory and practice of model checking, and students use SPIN in practical sessions. The main prerequisite for MRS is a discrete mathematics course for computing science, which covers the basics of set theory, predicate logic, relational algebra and methods of proof. In addition, students are required to have passed first year mathematics courses on calculus and algebra, as well as multiple computing science courses on programming, data structures and algorithms. Almost 20% of the assessment for MRS is via a practical exercise which involves specifying a reactive system using Promela, then reasoning about the specification with SPIN.

The MRS practical exercise for 2004/2005 involved producing three versions of a specification for a two user telephone system. Intuitively, a Promela specification of a two-user telephone exchange should exhibit one non-trivial symmetry which switches the local states of the users (and their associated channels) throughout all global states. Thus solutions to this modelling task provide a good set of Promela examples with which to evaluate the restrictions imposed by SymmExtractor, discussed in Section 2.1. Further, the associated state spaces are small enough for SPIN-to-GRAPE to compute all state space symmetries present in a given specification, which can be compared with those detected by SymmExtractor.

4 Evaluation

4.1 Ethical Approval

To ensure that our user study is ethical, we have followed the *Glasgow Ethics Code* check-list [13]. This is a 12-point check-list distilled from the ethical standard of the British Psychological Society [2], and focuses on the issues which are most relevant to computing science projects. Compliance with most of the points on the check-list was straightforward. The following points required some care:

- **All participants explicitly stated that they agreed to take part** Students who allowed us to use their solutions in the study were provided with an information sheet detailing the aims of the study, and asked to sign a consent form (provided as Appendix A and adapted from a standard example [14]). The intended usage of students' solutions is detailed in Section 4.2.
- **The researcher conducting the experiment is not in a position of authority or influence over any of the participants** As the solutions formed part of the course assessment, it was important that the the consent of students was not sought until after solutions had been assessed and returned. This assured students that their decision to take part in the study could have no effect on their score for the exercise, and encouraged them to answer the assessed questions in exactly the same way as they would have otherwise.

A further ethical concern is that the assessed exercise should be designed to meet the intended learning outcomes of the course and not to meet research aims (unless these overlap). In addition, since assessment has been shown to narrow students' focus [1], care must be taken to ensure that an assessment biased towards the research interests of the course director does not restrict breadth of learning. In our case the exercise had been set to meet the course aims before we designed our evaluation.

The study was approved by the ethics committee of the Faculty of Information and Mathematical Sciences at the University of Glasgow (ref. *FIMS00203*). We obtained signed consent forms from 17 students from a class of 35.

4.2 Methods

For each specification in the sample set we gathered the following data by a combination of automatic and manual analysis:

1. Size of the unreduced state space (computed using SPIN)
2. State space symmetries computed by SPIN-to-GRAPE, and size of the resulting quotient state space
3. Symmetry breaking features of the specification, and modifications required to restore symmetry (documented by experimenter)
4. Violations of restrictions imposed by SymmExtractor (as reported by the tool) and modifications required to satisfy restrctions (documented by experimenter)
5. Symmetries detected by SymmExtractor
6. Size of the quotient state space computed by TopSPIN.

Symmetry breaking features are aspects of the specification which destroy the intuitive symmetry discussed in Section 3. When SPIN-to-GRAPE showed absence of this expected symmetry in a given specification, the experimenter manually examined the specification to identify symmetry breaking features. We classify the modifications of 4 above as *minor* if they could be avoided by a straightforward extension of SymmExtractor, *medium* if they would be unnecessary if SymmExtractor could capture symmetry between global variables, or *major* if they could only be avoided by significant development of the theory on which SymmExtractor is based. The quotient state space is computed using TopSPIN to ensure that it matches the quotient structure constructed with respect to the symmetries computed by SPIN-to-GRAPE.

4.3 Results

We refer to the individual components of a three part solution as specifications. Of the 51 specifications analysed, just over half did not exhibit the expected symmetry due to symmetry breaking features. In most cases this was because `run` statements were not surrounded by an `atomic` block; for other examples the telephone users were initialised asymmetrically (e.g. *handset* variables for users 1 and 2 were set to *up* and *down* respectively, destroying symmetry between users). In all cases it was possible to restore symmetry by trivial modifications, with a negligible effect on the global state space. With these modifications, SymmExtractor was able to detect symmetry immediately from 23 of the resulting specifications. A further 13 required modifications which we classified as *minor* – these included replacing locally instantiated channels with globally instantiated channels, and removing channel instantiation statements from record declarations. Another 7 specifications required *medium* modifications (as described above). The final 8 specifications required *major* modifications. These modifications have identified a problem with the usability of the tool, which involves the way that arrays indexed by process identifiers are accessed.

We have extended the SymmExtractor documentation with a short set of modelling guidelines based on the problems encountered when applying the tool to this set of examples. It is clear that the tool would be more useful if it could handle symmetry between global variables, and an approach to this extension is sketched in [6]. The main challenge which the evaluation results have presented is to find techniques to automatically determine the relationship between numeric identifiers passed as parameters to processes by the user (and used to access arrays), and the runtime id values which SPIN assigns to processes.

5 Conclusions and Future Work

We have presented a case study in formal methods evaluation, using solutions to a student assignment as input to the SymmExtractor symmetry detection tool. The evaluation has identified some necessary improvements to the tool, as well as some modelling styles which destroy potential symmetries of a model. We have also discussed the ethical procedure which we followed before using student programs.

We are currently evaluating SymmExtractor further using another set of student programs, which model a railway signalling system. In future we hope to carry out this kind of evaluation *during* a formal methods course, so that students can learn by critically analysing new add-ons (like our symmetry reduction package) to existing tools. The students were unaware, when designing their solutions to this assessed exercise, that symmetry detection and reduction techniques would later be applied to their programs. However, symmetry reduction is part of the MRS course. Therefore it would be useful to see the effect on their programming style had they been asked to build specifications suitable for symmetry reduction. As development of SymmExtractor and TopSPIN continues this is very much a future research direction.

References

1. J. Bowden, G. Masters and P. Ramsden. Influence of assessment demands on first year students' approaches to learning. *Research and Development in Higher Education: A Forgotten Species?*, pages 397–407. Higher Education Research and Development Society of Australia, 1987.
2. British Psychological Society code of conduct. http://www.bps.org.uk/
3. E.M. Clarke, E.A. Emerson, S. Jha, and A.P. Sistla. Symmetry reductions in model checking. In *CAV'98*, LNCS 1427, pages 147–158. Springer, 1998.
4. E.M. Clarke, R. Enders, T. Filkhorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1–2):77–104, 1996.
5. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.
6. A.F. Donaldson and A. Miller. Automatic symmetry detection for model checking using computational group theory. In *FM'05*, LNCS 3582, pages 418–496. Springer, 2005.
7. A.F. Donaldson and A. Miller. A computational group theoretic symmetry reduction package for the SPIN model checker. In *AMAST'06*, LNCS 4019, pages 374–380. Springer, 2006.
8. A.F. Donaldson, A. Miller and M. Calder. SPIN-to-GRAPE: a tool for analysing symmetry in Promela models. *Electronic Notes in Theoretical Computer Science*, 139(1):3–23, 2005.
9. The Gap Group. *GAP–Groups Algorithms and Programming, Version 4.2*. Aachen, St. Andrews, 1999. http://www-gap.dcs.st-and.ac.uk/~gap.
10. G. J. Holzmann. *The SPIN model checker: primer and reference manual*. Addison Wesley, 2003.
11. C.N. Ip and D.L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2): 41–75, 1996.
12. A. Miller, A. Donaldson and M. Calder. Symmetry in temporal logic model checking. *Computing Surveys*, 2006. To appear.
13. H.C. Purchase. Student compliance with ethical guidelines: The Glasgow Ethics Code. In *Proc. Higher Education Academy 6th Annual Conference on Information and Computer Sciences*, pages 82–85, 2005.
14. H.C. Purchase. Example participant consent form. http://www.dcs.gla.ac.uk/∼/ethics/
15. A.P. Sistla, V. Gyuris, and E.A. Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 9(2):113–166, 2000.
16. L.H. Soicher. Computing with graphs and groups. In *Topics in Algebraic Graph Theory*, pages 250–266. Cambridge University Press, 2004.

Appendix

Participant Consent Form: Symmetry in Promela Models

The aim of this experiment is to investigate structural symmetry arising in typical Promela models of distributed systems.

The experiment will involve allowing the experimenter to analyse your assessed exercise submission for last year's Modelling Reactive Systems 4 course. The analysis is concerned with the structure of the state space underlying your solutions, *not* with the semantic correctness of the solutions.

All results will be held in strict confidence, ensuring the privacy of all participants. No personal participant information will be stored within the data. Data will be stored online in a password protected computer account.

A feedback email message will be sent to all participants, after the data has been analysed.

Please note that it is the Promela language, not you, that is being evaluated. You may withdraw from the experiment at any time without prejudice, and any data already recorded will be discarded.

If you have any further questions regarding this experiment, please contact:

Alastair Donaldson
Computing Science Department
Lilybank Gardens
ally@dcs.gla.ac.uk

I have read the information sheet, and agree to voluntarily take part in this experiment:

Name: _____ Email: _____

Signature: _____ Date: _____

This study adheres to the BPS ethical guidelines, and has been approved by the FIMS ethics committee of The University of Glasgow (ref: FIMS00203). Whilst you are free to discuss your participation in this study with the researcher (contactable on 330 4236 ext. 0049), if you would like to speak to someone not involved in the study you may contact the chairs of the FIMS Ethics Committee: s.garrod,s.schweinberger@psy.gla.ac.uk.

Teaching with the Computerised Package *Language, Proof, and Logic* (LPL)

Roussanka Loukanova

Computational Linguistics
Dept. of Linguistics and Philology, Uppsala University

Abstract. In the first sections of this paper, I discuss briefly education in foundational subjects involving Mathematical Logic. Then, I share my experience with a courseware for introductory and intermediate undergraduate courses in Mathematical Logic, which incorporates a pedagogical approach of combining:
(1) a textbook, which introduces logic methods of reasoning and proofs in informal, but correct way, which is then paralleled by formalization
(2) using computer software for better comprehension of abstract and theoretical concepts.
(3) practice with model checkers and automated provers.
I will demonstrate the programs in the *Language, Proof, and Logic* (LPL) package with several exercises on models and formal proofs.

1 Importance of Education in Formal Methods

Teaching Formal Methods typically, but not always, is conducted by courses such as Math Logic, Discrete Math, Computability, etc. It has an old history, the beginning of which is the beginning of the Philosophy and Sciences. Courses in logic have been indispensable from the academic education in various major fields of sciences up to their modern proliferation. Nowadays, knowledge and skills in mathematical logic are essential for various subjects such as mathematics, computer science, philosophy, general and computational linguistics, informatics, etc. Various courses in logic are given for undergraduate and graduate students in many of the educational programs of these subjects. This has led the lecturers to select and include in their courses specific topics of formal methods, depending on the demands of the particular subjects. A special attention has been given to education in Math Logic in the context of the undergraduate curriculum. With the advance of the information technologies, we can not overestimate the importance of teaching formal methods in the entire university education, including for forming formal reasoning in humanities and arts. Despite of that, there is a tendency for replacing existing requirements for courses in logic with other, unrelated, courses. Even, in the above mentioned disciplines, the current trend is for reducing the logical content of the relevant courses, and especially for eliminating proofs. This problem exists even in computer science and computational linguistics, where the techniques of proofs are the layout of many, if not the most, formal and computational approaches to contemporary applications.

Why logic is, and in general formal methods are, important to education in mathematics, computer science, philosophy, computational linguistics, bioinformatics, medicine,

and so on? The original research on logic and formal methods continues to be as actual as in its origin times: the methods of reasoning and deriving new knowledge in a consistent, sound way, without faults. This by itself is sufficient for including it in the educational programs. But with the contemporary advent of computers and information technologies there are more arguments for including formal methods in the curricula.

Firstly, the computers themselves (both, hardware and software) are a direct practical output of developments in logic, incl. its sub-fields of computability.

Secondly, all of the above mentioned subjects require skills for using computers and computer systems by understanding the foundational basics. This involves a flaw of current software tools for logic and other computability methods, e.g., formal grammar tools, finite automata, Turing machines. The flaw even goes deeper: Logic and Computability software tools are very implementation-bound. Similar problem exists for the computational grammar tools for processing human languages, where the parsing techniques of software tools are typically based on variations of Context-Free parsing, while the grammar theory implemented is typically diverging from the classical Chomsky Hierarchy of Grammars (see Loukanova, HT 2006, Computational Grammar II). Software tools for classical math do not suffer from this problem (e.g., Lagrange would be able to use a tool like Maple with at most one minute of computer introduction).

And thirdly, and most importantly, understanding the formal foundations is necessary for the education of developers of computers, computer systems and applications. Education of professionals for the job market related to these subjects requires providing knowledge and skills for work with formal systems and models and relating them to real world tasks.

Some universities include logic courses for humanity and arts students, but more often, the formal reasoning requirements are either optionally, or entirely, covered by pure math courses, such as calculus or probability. For example, Indiana University students in Humanity and Arts can take a course in Finite Math or Calculus. As another example, Notre Dame requires at least two semesters mathematics, where the courses depend on the major of students. For students in science, business and engineering, this includes Calculus; for mathematics students, this can include some sub-field of Logic, such as Computability and Automata Theory on topics of what can and what can not be computed, and if a task is computable, how and with what resources.

2 Some of the Goals of Teaching Formal Methods

Educational programs in formal methods vary among subjects, educational programs and traditions of schools. The argument for providing Mathematical Logic for humanity and arts is to provide students with formal methods of sound reasoning and abilities of analyzing arguments. For that, basic courses in classic logic suffice. But for students in computer science and computational linguistics, it is important that education provides foundational background for syntax and semantics of programming languages and human languages. The emphasis on the latter is lesser in computer science. For the education in computer science, these courses would, typically, differ in topics, goals and methods, from those for computational linguistics, and far more for humanities.

Graduate (PhD) programs in natural sciences, such as medicine and biology, should offer advanced courses in formal methods: These would provide foundations for interdisciplinary research and developing advanced industry technologies for building sound bio models and simulation, and introducing formal verification of models of complex bio systems.

Thus, in all varieties of educational programs, courses on formal methods vary in goals and topics, but they also share common features which are specific for the subject of formal methods, incl. logic. The many years work on education in logic is reflected, for instance, by the *Guidelines on Logic Education* of the Committee on Logic Education of the ASL, The Bulletin of Symbolic Logic 1, 1995.

The goals of teaching in Mathematical Logic include, but are not limited to the common goals of higher education, such as:

- independent work on the course
- preparation for meeting and taking real and serious tasks related to the subject
- promoting discovery up to striving for challenging tasks on the subject
- abilities to use the learned methods and techniques for applications in other subjects and real life tasks

General common goals of teaching in Logic:

- Some of the direct goals include covering
  - the fundamentals of model theory
  - syntax-semantics relationships
  - symbolic manipulations
  - methods and techniques of proofs
- Long term goals related to the every-day real life:
  - understanding foundational logical principles of reasoning and argumentation
  - to provide basis of relating formal reasoning in symbolic languages to everyday human language, or to professionally specialized usage of human language
  - managing information flow: techniques of processing available information represented in any form, spoken and written, in pictures, diagrams, etc., and being able to derive new information in reliable ways (a task which is receiving a special status with the contemporary advances of information technology)
- Specific subject related goals:
  - to provide reliable knowledge about the formal and theoretical apparatus of the selected topics
  - learning topic specific techniques and methods
  - to promote ability to generalize specific techniques in varying and new tasks
  - to provide basis for further independent study and research in the subject of formal methods
  - abilities to apply formal methods in real life applications and tasks for meeting the demands of sciences and industry in the context of current information and technology advances

3 Typical Topics Covered by Logic Courses

The topics covered by a general course on formal methods can be summarized, with some variations depending on specific needs and context as follows (see also the *Guidelines on Logic Education* of ASL):

Mathematics students get specific training in the various fields and sub-fields of mathematics. They learn various, specific for these fields, techniques for geometric, numerical, functional, etc., manipulations and rigorous proofs. The goals of (sequences of) courses in logic for mathematicians is to provide knowledge of the formal background of these techniques and Foundations of Mathematics.

Logic and computability provide the foundations of Computer Science in all of its essentials (from the hardware to software, via the operating systems and programming, including specialties on syntax, semantics and compilers of programming languages, algorithms, data structures, cryptology and security techniques, data bases and software applications). It should be already a standard (which is still far from the reality at some universities) that computer science departments teach logic and computability at all levels of the study, by courses with titles such as, Theoretical Foundations of Computer Science, Logic, Computability, Finite Mathematics, Discrete Math, Theory of Automata and Formal Languages, etc. The specific topics covered vary depending on the level of the courses and other factors.

Computational Linguistics is an interdisciplinary field between theoretical linguistics, computer science, logic, mathematics, artificial intelligence, cognitive science and neuroscience. It covers the contemporary developments for computerized processing of human languages. To provide foundational knowledge, the goals and topics of the courses on formal methods for Computational Linguistics are very close to those for Computer Science, but with a stronger emphasis on Model Theory, Higher Order Logics, Type Logic Grammars, Computational Semantics (by focusing on compositionality), Formal Grammars and Languages, Automata Theory, Parsing Methods specialized for human languages, Methods of Resolution and Unification. These courses provide foundations of formal and computational approaches to grammar, in its broad coverage: computerization of morphology, lexicon, syntax and semantics.

In the last years, there has been a dramatic move to developing computerized applications of Logic (i.e., the development of the software itself involves logic) for teaching formal methods at all levels, from the undergraduate introductory logic up to advanced graduate courses in mathematics, computer science, computational linguistics, artificial intelligence, cognitive science and other subjects. A list of some computerized logic systems and courseware, including a very brief description, and Internet resource addresses, is given by the Committee on Logic Education of ASL at `http://www.phil.ucalgary.ca/asl-cle/`.

4 Teaching and Learning Logic with the package *Language, Proof, and Logic*

In this section, I will share my teaching experience[1] in meeting the learning criteria by using the package *Language, Proof, and Logic*.

4.1 The Courseware Package

The *Language, Proof, and Logic* package comes with a textbook and a CD which includes a complete manuscript of the textbook in searchable pdf format and the following programs:

Tarski's World software for learning the syntax and semantics of first order languages via a particular first-order language and its finite models. Students can write statements about geometric objects and some relations between them. By using the software, they can build visual models of sets of statements and check their truth and satisfiability. The software has a friendly tool called `Game` with which students can follow up their mistakes and understand them in a very pedagogical way.

Fitch is a natural deduction proof environment for building and checking first-order proofs. Students can try and check up their intuitions in writing formal proofs by "backward" reasoning, and with the help of *Fitch* they can check for wrong steps, discard them and backtrack to follow correct ones.

Boole is a program that facilitates the construction and checking truth tables. With it, they can learn, in a very efficient way, such notions as tautology, tautological consequence, etc. By using *Boole*, students are able to build up in a very fast and efficient way even large truth tables that are error-free. With it, they are able to solve problems that, if done on paper by hand, would be very tedious and error-prone.

Submit allows students to submit, with a personal ID, via the Internet, solutions of exercises to *Grade Grinder* which is an automatic grading service. *Grade Grinder* sends (in just few minutes) email reports to students about how they did. Students can submit unlimited number of times for improving their solutions (without involving a lecturer). When satisfied by their results, they can request *Grade Grinder* to sent a report to the lecturer for course credits. The lecturer can see the details of the final solutions by getting email reports and by Internet logging to the grading server.

The software programs facilitate enormously learning and promote understanding of the material by students, especially those who encounter the formal and abstract methods for the first time. In addition to this, the programs have beautiful visual interfaces and are easy to use.

The LPL package has Internet web site with software updates and information about the software, examples and other resources.

4.2 Knowledge Provided by the Course Package

The package is an introduction to logic, which can be used in introductory or intermediate level undergraduate courses. It is appropriate for logic courses in philosophy,

[1] I also have experience with Turing's World, Hyperproof, MIZAR, PVS, and other software for computational grammars.

computer science, mathematics, general and computational linguistics, artificial intelligence, cognitive science, bioinformatics, (pre)-medical and other schools. In some of these disciplines, including computer science, mathematics and computational linguistics, more depth is needed. Formal definitions and results (meta-theorems) with proofs can be provided either by using additional textbooks and lecture materials.

The aim of the package is learning the syntax and semantics of first order language and logic in a way that is friendly and pleasurable for students. However, to avoid the risks of transient "entertainment elements" in education, which wear off rapidly, up to becoming nuisance, the package should be used in a combination with more advanced introduction in logic, especially for Computer Science and Computational Linguistics.

The models of first order language is presented in a clear way by demonstrating relationships between human language (English), a language of logic and models of real world objects. Tarski's World program can be used for building models of sets of sentences. The textbook includes plenty of exercises with Tarski's World for translating English sentences into first order language. These exercises facilitate learning the syntax and semantics of both, natural and logic languages. By testing truth conditions with Tarski's World, students learn logic and, also, get closer awareness of unexpected subtleties of the human language syntax and its semantics. The experiences with these type of exercises is especially important for students in computational linguistics. In natural (human) language processing, ambiguity of natural language is one of the fundamental difficulties and topic of applications and research. The textbook includes numerous exercises with Tarski's World, which help students to comprehend complex relations between syntax and semantics of human languages and their ambiguity. These exercises prepare students for the introductory and advanced courses in Computational Semantics, where various scope ambiguities are covered (for example, quantifiers and attitudes).

The courseware investigates the notion of logical consequence in first order logic and its relation to the natural logical consequence in human languages. Throughout the textbook, students learn logic rules of reasoning and proofs in two styles, informal and formal. The textbook introduces several methods of proofs, in particular, natural deduction system, resolution, counterexamples, mathematical induction and others. Natural deduction is practiced by using the Fitch software in the package. The method of counterexample for demonstrating that a claim is not a consequence of others, is introduced by providing explanations and exercises with Tarski's World software.

The students can read more advanced, optional sections about major results in first order logic, which are introduced in a clear, intuitive way. They learn techniques for applications of logic procedures such as unification and resolution in logic programming.

What students learn with the logic software in the package is oriented towards traditional formal logic, but it is a friendly first experience, which prepares them for other advanced logic software and applications of logic in computer science and computational linguistics. Such applications include verification systems, automatic provers, model checkers, formal and computational syntax and semantics of natural language, etc.

4.3 Pedagogical Approach

The entire package is specially developed by achieving the effects of a unique pedagogical approach: The textbook introduces students to logic notions and concepts in a step-by-step way. It gives clear and intuitive descriptions by including examples and exercises, many of which are solved in details with self-study instructions. The logic concepts are pedagogically grouped and each group is introduced at first with clear intuitive explanations and problem examples with informal solutions. The informal introduction is then followed by a formal introduction. In this way, students learn techniques for informal, but correct reasoning and proofs. They are able to comprehend far easier and deeper the formal techniques and results.

Many concepts can be learn by a combination of reading the text and performing exercises with the software applications. The exercises in each section are preceded by examples of detailed solutions marked as `Try it`. For better learning of complex concepts and results, the textbook also includes many exercises that are for work on paper by a pencil, in the classic way for abstract theories.

The textbook can be used with or without software applications. A lecturer does not need to use the software in the lectures. The pedagogical value of the book is complete even if used as a classical reading material.

4.4 The Grading System

By using the programs, students are able to do an impressive amount of exercises, which normally, on paper by hand, would be extraordinarily time consuming, if possible at all.

Solutions and proofs, if done by hand (or typeset in some text-editor), are prone to mistakes which can go unnoticeable by students (and sometimes by lecturers). Typically, in the traditional way, students submit solutions with some mistakes and then, after grading, do not have much opportunities to correct themselves for a better grade. With the software help, students have the opportunity to search after mistakes to improve their solutions. In this way, they learn by deeper understanding the concepts in the process of correcting their own mistakes. In addition, the achieved results provide them with unique satisfaction and promote further learning.

Something more, with the automatic grading system, a lecturer can assign an impressive number of exercises for homework and exams, which otherwise, would be extraordinary, up to impossible, amount of grading work. The textbooks includes also exercises for solving and grading in a traditional way.

5 Teaching Experiments and Experience with LPL

I have used LPL in the following courses:

- Indiana University (IU), course *Mathematical and Logic Methods for Cognitive Science* (1998)
  Jon Barwise and I developed the course for the Cognitive Science program of IU. We used Tarski's World, Fitch and Turing's World in the labs. Students had read scheduled material prior to lectures. We conducted lectures "on the blackboard,"

by giving definitions, theorems and proofs in response to student's questions. The learning outcomes were very high.

- Indiana University (IU), course *Discrete Mathematics*, Computer Science, (1998-1999).
  I used the LPL programs in the labs to the course, while other textbooks were used for lectures.
- I used the LPL courseware package for teaching Seminars on Topics in Computational Semantics at the University of Minnesota, Minneapolis. The course was for students majoring in Computer Science and Computational Linguistics. Learning outputs were very high.
- I used the LPL programs in the labs as part of a course Natural Language Processing (NLP) for Computer Science students at Uppsala University (2002, 2003). The learning outcomes were very high.
- I have been using the LPL courseware package for The Natural Language Engineering (Computational Linguistics) program of Uppsala University since 2002 in the courses on Logic (see Loukanova, VT 2003, Foundations of Computer Science I and Loukanova, HT 2006, Foundations of Computer Science II) and Computational Semantics (see Loukanova, VT 2006, Computational Semantics II). To provide more in dept coverage of the theoretical background in Logic needed for formal and computational methods in NLP, I prepared additional lecture notes. However, the background level of the students, and the limitations of the course time, do not permit including theoretical results and proofs.

6 Conclusions about the LPL Courseware

By my teaching experience, the courseware package incorporates unique pedagogical methods, which can be summarized as follows:

- Providing intuitive and clear descriptions supplemented with many real life examples
- Teaching informal but correct reasoning by giving proofs in natural and less technical language
- Informal proofs are followed by formal proofs. Such parallel presentations promote deep learning of technical concepts and theory.
- Promoting independent study, which targets and achieves deep understanding and discovery
- Engaging two learning modes: *Read It* (by reading the textbook, and other materials); and *Do It* (by practicing with the logic applications)
- The independent study is highly promoted by the advanced computer software and on-line Internet resources.
- The grading software provides objective and verified testing.

My experience with LPL courseware has convinced me that it has to be used together with additional teaching materials for covering in depth formal definitions and theoretical results with proofs. There is a serious conflict between the time limitations

of the courses in Logic and the educational needs of Computational Linguistics, Natural Language Processing (NLP)[2] and Computer Science[3] for a solid background in formal methods. One way for filling up these needs, seems to be developing materials on formal methods that are appropriate for more independent learning by students, with supervision and consultations by a lecturer, instead of the classical lecturing.

I will provide this discussion of LPL software package by demonstrating its programs with exercises.

7 References

1. Association for Symbolic Logic, Committee on Logic Education: http://www.phil.ucalgary.ca/asl-cle/
2. *Guidelines on Logic Education.* The Bulletin of Symbolic Logic 1, 1995, 4-8. Association for Symbolic Logic. 1079-8986/95/0101-0002. http://www.phil.ucalgary.ca/asl-cle/guidelines.pdf
3. *Language, Proof, and Logic.* Jon Barwise and John Etchemendy, et al., 1999, CSLI. Stanford. (http://www-csli.stanford.edu/LPL/)
4. Loukanova, R., VT 2003, Foundations of Computer Science I (Logic) http://stp.ling.uu.se/~rloukano/classes/2003s/logik/index.html
5. Loukanova, R., HT 2006, Foundations of Computer Science II (Mathematical Linguistics) http://stp.ling.uu.se/~rloukano/classes/06a/math_ling/syllabus.html
6. Loukanova, R., VT 2006, Computational Semantics II, http://stp.ling.uu.se/~rloukano/classes/06s/compsem_2/syllabus.html
7. Loukanova, R., HT 2006, Computational Grammar II, http://stp.ling.uu.se/~rloukano/classes/06a/compgr_2/syllabus.html

[2] These two areas are overlapping, but different.

[3] NLP is dependent on, and in many universities is part of, the Computer Science curriculum

Author Index